Oracle® Spatial and Graph

RDF Semantic Graph Developer's Guide

12*c* Release 1 (12.1)

E51611-08

June 2017

Provides usage and reference information about Oracle Database support for semantic technologies, including storage, inference, and query capabilities for data and ontologies based on Resource Description Framework (RDF), RDF Schema (RDFS), and Web Ontology Language (OWL).

ORACLE

Oracle Spatial and Graph RDF Semantic Graph Developer's Guide, 12*c* Release 1 (12.1)

E51611-08

Copyright © 2005, 2017, Oracle and/or its affiliates. All rights reserved.

Primary Author: Chuck Murray

Contributors: Eugene Inseok Chong, Souri Das, Vladimir Kolovski, Matt Perry, Karl Rieb, Jags Srinivasan, Seema Sundara, Zhe (Alan) Wu, Aravind Yalamanchi

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface xix

Audience xix
Documentation Accessibility xix
Related Documents xix
Conventions xx

Changes in This Release for Oracle Spatial and Graph RDF Semantic Graph Developer's Guide xxi

Changes in Oracle Database 12c Release 1 (12.1.0.2) xxi
Support for SPARQL 1.1 Federated Query xxi
Combining Native Triple Data with Virtual RDB2RDF Triple Data xxi
Changes in Oracle Database 12c Release 1 (12.1.0.1) xxi
New Features xxi
Deprecated Features xxiii
Desupported Features xxiii
Changes for RDF Semantic Graph Support for Apache Jena xxiii
Support for Retrieving User-Friendly Java Objects from SEM_MATCH or SQL-Based Query Results xxiv
Protege Plugin for Oracle Database and Oracle NoSQL Database Enterprise Edition Release 2.0.39 xxiv
Support for Customized Data Source Name xxiv
Support for SPARQL Queries with Translated SQL Text Larger Than 29000 Bytes xxiv
Support for Oracle Database in Read-Only Mode xxv
Less Verbose Joseki Output xxv
Java APIs for Managing the Table Responsible for Terminating Long-Running SPARQL Queries xxv
Support for Apache Tomcat and JBoss Application Server xxv
Support for Fuseki 1.0.1 xxv

Part I Conceptual and Usage Information

1 RDF Semantic Graph Overview

1.1 Introduction to Oracle Semantic Technologies Support 1-2
1.2 Semantic Data Modeling 1-3
1.3 Semantic Data in the Database 1-3
1.3.1 Metadata for Models 1-3
1.3.2 Statements 1-5
1.3.3 Subjects and Objects 1-8
1.3.4 Blank Nodes 1-8
1.3.5 Properties 1-8
1.3.6 Inferencing: Rules and Rulebases 1-8
1.3.7 Entailments (Rules Indexes) 1-11
1.3.8 Virtual Models 1-13
1.3.9 Named Graphs 1-16
1.3.10 Semantic Data Security Considerations 1-17
1.4 Semantic Metadata Tables and Views 1-18
1.5 Semantic Data Types, Constructors, and Methods 1-20
1.5.1 Constructors for Inserting Triples 1-21
1.6 Using the SEM_MATCH Table Function to Query Semantic Data 1-22
1.6.1 Performing Queries with Incomplete or Invalid Entailments 1-29
1.6.2 Graph Patterns: Support for Curly Brace Syntax, and OPTIONAL, FILTER, UNION, and GRAPH Keywords 1-30
1.6.3 Graph Patterns: Support for SPARQL ASK Syntax 1-39
1.6.4 Graph Patterns: Support for SPARQL CONSTRUCT Syntax 1-39
1.6.5 Graph Patterns: Support for SPARQL DESCRIBE Syntax 1-44
1.6.6 Graph Patterns: Support for SPARQL SELECT Syntax 1-46
1.6.7 Graph Patterns: Support for SPARQL 1.1 Constructs 1-50
1.6.8 Graph Patterns: Support for SPARQL 1.1 Federated Query 1-61
1.6.9 Inline Query Optimizer Hints 1-65
1.6.10 Full-Text Search 1-65
1.6.11 Spatial Support 1-66
1.6.12 Best Practices for Query Performance 1-69
1.6.13 Special Considerations When Using SEM_MATCH 1-72
1.7 Loading and Exporting Semantic Data 1-74
1.7.1 Bulk Loading Semantic Data Using a Staging Table 1-74
1.7.2 Batch Loading N-Triple Format Semantic Data Using the Java API 1-77
1.7.3 Loading Semantic Data Using INSERT Statements 1-79
1.7.4 Exporting Semantic Data 1-79
1.7.5 Exporting or Importing a Semantic Network Using Oracle Data Pump 1-82
1.7.6 Purging Unused Values 1-83
1.8 Using Semantic Network Indexes 1-86

1.8.1 MDSYS.SEM_NETWORK_INDEX_INFO View ... 1-87
1.9 Using Data Type Indexes ... 1-88
1.10 Managing Statistics for Semantic Models and the Semantic Network ... 1-89
1.10.1 Saving Statistics at a Model Level ... 1-90
1.10.2 Restoring Statistics at a Model Level ... 1-91
1.10.3 Saving Statistics at the Network Level ... 1-91
1.10.4 Restoring Statistics at the Network Level ... 1-92
1.10.5 Setting Statistics at a Model Level ... 1-92
1.10.6 Deleting Statistics at a Model Level ... 1-92
1.11 Quick Start for Using Semantic Data ... 1-92
1.12 Semantic Data Examples (PL/SQL and Java) ... 1-93
1.12.1 Example: Journal Article Information ... 1-94
1.12.2 Example: Family Information ... 1-95
1.13 Software Naming Changes Since Release 11.1 ... 1-102
1.14 For More Information About RDF Semantic Graph ... 1-103

2 OWL Concepts

2.1 Ontologies ... 2-1
2.1.1 Example: Cancer Ontology ... 2-1
2.1.2 Supported OWL Subsets ... 2-2
2.2 Using OWL Inferencing ... 2-5
2.2.1 Creating a Simple OWL Ontology ... 2-5
2.2.2 Performing Native OWL inferencing ... 2-6
2.2.3 Performing OWL and User-Defined Rules Inferencing ... 2-6
2.2.4 Generating OWL inferencing Proofs ... 2-8
2.2.5 Validating OWL Models and Entailments ... 2-9
2.2.6 Using SEM_APIS.CREATE_ENTAILMENT for RDFS Inference ... 2-10
2.2.7 Enhancing Inference Performance ... 2-10
2.2.8 Optimizing owl:sameAs Inference ... 2-11
2.2.9 Performing Incremental Inference ... 2-13
2.2.10 Using Parallel Inference ... 2-14
2.2.11 Using Named Graph Based Inferencing (Global and Local) ... 2-15
2.2.12 Performing Selective Inferencing (Advanced Information) ... 2-18
2.3 Using Semantic Operators to Query Relational Data ... 2-19
2.3.1 Using the SEM_RELATED Operator ... 2-19
2.3.2 Using the SEM_DISTANCE Ancillary Operator ... 2-20
2.3.3 Creating a Semantic Index of Type MDSYS.SEM_INDEXTYPE ... 2-22
2.3.4 Using SEM_RELATED and SEM_DISTANCE When the Indexed Column Is Not the First Parameter ... 2-23
2.3.5 Using URIPREFIX When Values Are Not Stored as URIs ... 2-24

3 Simple Knowledge Organization System (SKOS) Support

3.1 Supported and Unsupported SKOS Semantics ... 3-1

3.1.1 Supported SKOS Semantics ... 3-2
3.1.2 Unsupported SKOS Semantics ... 3-3
3.2 Performing Inference on SKOS Models ... 3-3
3.2.1 Validating SKOS Models and Entailments ... 3-3
3.2.2 Property Chain Handling ... 3-4

4 Semantic Indexing for Documents

4.1 Information Extractors for Semantically Indexing Documents ... 4-2
4.2 Extractor Policies ... 4-4
4.3 Semantically Indexing Documents ... 4-5
4.4 SEM_CONTAINS and Ancillary Operators ... 4-6
4.4.1 SEM_CONTAINS_SELECT Ancillary Operator ... 4-7
4.4.2 SEM_CONTAINS_COUNT Ancillary Operator ... 4-7
4.5 Searching for Documents Using SPARQL Query Patterns ... 4-7
4.6 Bindings for SPARQL Variables in Matching Subgraphs in a Document (SEM_CONTAINS_SELECT Ancillary Operator) ... 4-8
4.7 Improving the Quality of Document Search Operations ... 4-9
4.8 Indexing External Documents ... 4-10
4.9 Configuring the Calais Extractor type ... 4-12
4.10 Working with General Architecture for Text Engineering (GATE) ... 4-12
4.11 Creating a New Extractor Type ... 4-13
4.12 Creating a Local Semantic Index on a Range-Partitioned Table ... 4-14
4.13 Altering a Semantic Index ... 4-15
4.13.1 Rebuilding Content for All Existing Policies in a Semantic Index ... 4-15
4.13.2 Rebuilding to Add Content for a New Policy to a Semantic Index ... 4-15
4.13.3 Rebuilding Content for an Existing Policy from a Semantic Index ... 4-15
4.13.4 Rebuilding to Drop Content for an Existing Policy from a Semantic Index ... 4-16
4.14 Passing Extractor-Specific Parameters in CREATE INDEX and ALTER INDEX ... 4-16
4.15 Performing Document-Centric Inference ... 4-16
4.16 Metadata Views for Semantic Indexing ... 4-17
4.16.1 MDSYS.RDFCTX_POLICIES View ... 4-17
4.16.2 RDFCTX_INDEX_POLICIES View ... 4-18
4.16.3 RDFCTX_INDEX_EXCEPTIONS View ... 4-18
4.17 Default Style Sheet for GATE Extractor Output ... 4-19

5 Fine-Grained Access Control for RDF Data

5.1 Triple-Level Security ... 5-2
5.1.1 Fine-Grained Security for Inferred Data and Ladder-Based Inference (LBI) ... 5-3
5.1.2 Extended Example: Applying OLS Triple-Level Security on Semantic Data ... 5-5
5.2 Resource-Level Security ... 5-11
5.2.1 Securing RDF Subjects ... 5-12
5.2.2 Securing RDF Predicates ... 5-13
5.2.3 Securing RDF Objects ... 5-13

5.2.4 Generating Labels for Inferred Triples 5-13
5.2.5 Using Labels Based on Application Logic 5-16
5.2.6 RDFOLS_SECURE_RESOURCE View 5-18

6 Workspace Manager Support for RDF Data

6.1 Enabling Workspace Manager Support for RDF Data 6-2
6.1.1 Removing Workspace Manager Support for RDF Data 6-2
6.2 Version-Enabling an RDF Model 6-2
6.3 Inferring from Version-Enabled RDF Models 6-3
6.4 Merging and Refreshing Workspaces in Version-Enabled RDF Models 6-4
6.5 Special Considerations When Using Workspace Manager Support for RDF Data 6-5
6.6 Usage Flow Example: Versioning Semantic Models 6-5

7 RDF Semantic Graph Support for Apache Jena

7.1 Setting Up the Software Environment 7-2
7.1.1 If You Used a Previous Version of the Support for Apache Jena 7-3
7.2 Setting Up the SPARQL Service 7-4
7.2.1 Creating the Required Data Source Using WebLogic Server 7-6
7.2.2 Configuring the Joseki-Based SPARQL Service 7-6
7.2.3 Configuring the Fuseki-Based SPARQL Service 7-9
7.2.4 Terminating Long-Running SPARQL Queries 7-10
7.2.5 N-Triples Encoding for Non-ASCII Characters 7-10
7.3 Setting Up the RDF Semantic Graph Environment 7-10
7.4 SEM_MATCH and RDF Semantic Graph Support for Apache Jena Queries Compared 7-11
7.5 Retrieving User-Friendly Java Objects from SEM_MATCH or SQL-Based Query Results. 7-12
7.6 Optimized Handling of SPARQL Queries 7-15
7.6.1 Compilation of SPARQL Queries to a Single SEM_MATCH Call 7-16
7.6.2 Optimized Handling of Property Paths 7-16
7.7 Additions to the SPARQL Syntax to Support Other Features 7-17
7.7.1 SQL Hints 7-18
7.7.2 Using Bind Variables in SPARQL Queries 7-18
7.7.3 Additional WHERE Clause Predicates 7-20
7.7.4 Additional Query Options 7-20
7.7.5 Midtier Resource Caching 7-23
7.8 Functions Supported in SPARQL Queries through RDF Semantic Graph Support for Apache Jena 7-24
7.8.1 Functions in the ARQ Function Library 7-24
7.8.2 Native Oracle Database Functions for Projected Variables 7-24
7.8.3 User-Defined Functions 7-25
7.9 SPARQL Update Support 7-29
7.10 Analytical Functions for RDF Data 7-31
7.10.1 Generating Contextual Information about a Path in a Graph 7-37
7.11 Support for Server-Side APIs 7-38

7.11.1 Virtual Models Support ... 7-38
7.11.2 Connection Pooling Support ... 7-40
7.11.3 Semantic Model PL/SQL Interfaces ... 7-41
7.11.4 Inference Options ... 7-41
7.11.5 PelletInfGraph Class Support Deprecated ... 7-44
7.12 Bulk Loading Using RDF Semantic Graph Support for Apache Jena ... 7-44
7.12.1 Using prepareBulk in Parallel (Multithreaded) Mode ... 7-47
7.12.2 Handling Illegal Syntax During Data Loading ... 7-49
7.13 Automatic Variable Renaming ... 7-50
7.14 JavaScript Object Notation (JSON) Format Support ... 7-51
7.15 Other Recommendations and Guidelines ... 7-53
7.15.1 BOUND or !BOUND Instead of EXISTS or NOT EXISTS ... 7-53
7.15.2 SPARQL 1.1 SELECT Expressions ... 7-53
7.15.3 Syntax Involving Bnodes (Blank Nodes) ... 7-53
7.15.4 Limit in the SERVICE Clause ... 7-54
7.15.5 OracleGraphWrapperForOntModel Class for Better Performance ... 7-54
7.16 Example Queries Using RDF Semantic Graph Support for Apache Jena ... 7-56
7.16.1 Test.java: Query Family Relationships ... 7-56
7.16.2 Test6.java: Load OWL Ontology and Perform OWLPrime inference ... 7-57
7.16.3 Test7.java: Bulk Load OWL Ontology and Perform OWLPrime inference ... 7-59
7.16.4 Test8.java: SPARQL OPTIONAL Query ... 7-60
7.16.5 Test9.java: SPARQL Query with LIMIT and OFFSET ... 7-61
7.16.6 Test10.java: SPARQL Query with TIMEOUT and DOP ... 7-63
7.16.7 Test11.java: Query Involving Named Graphs ... 7-64
7.16.8 Test12.java: SPARQL ASK Query ... 7-65
7.16.9 Test13.java: SPARQL DESCRIBE Query ... 7-66
7.16.10 Test14.java: SPARQL CONSTRUCT Query ... 7-67
7.16.11 Test15.java: Query Multiple Models and Specify "Allow Duplicates" ... 7-69
7.16.12 Test16.java: SPARQL Update ... 7-70
7.16.13 Test17.java: SPARQL Query with ARQ Built-In Functions ... 7-71
7.16.14 Test18.java: SELECT Cast Query ... 7-72
7.16.15 Test19.java: Instantiate Oracle Database Using OracleConnection ... 7-73
7.16.16 Test20.java: Oracle Database Connection Pooling ... 7-74
7.17 SPARQL Gateway and Semantic Data ... 7-75
7.17.1 SPARQL Gateway Features and Benefits Overview ... 7-76
7.17.2 Installing and Configuring SPARQL Gateway ... 7-76
7.17.3 Using SPARQL Gateway with Semantic Data ... 7-78
7.17.4 Customizing the Default XSLT File ... 7-82
7.17.5 Using the SPARQL Gateway Java API ... 7-82
7.17.6 Using the SPARQL Gateway Graphical Web Interface ... 7-85
7.17.7 Using SPARQL Gateway as an XML Data Source to OBIEE ... 7-91
7.18 Deploying Joseki in Apache Tomcat or JBoss ... 7-94
7.18.1 Deploying Joseki in Apache Tomcat 6.0.29 or 7.0.42 ... 7-94

7.18.2 Deploying Joseki in JBoss 7.1.1 7-96

8 RDF Semantic Graph Support for OpenRDF Sesame

8.1 RDF Semantic Graph Support for OpenRDF Sesame Overview 8-2
8.2 Setup and Configuration for RDF Semantic Graph Support for OpenRDF Sesame 8-3
8.2.1 Setting Up the Software Environment 8-3
8.2.2 Setting Up the SPARQL Service 8-4
8.2.3 Setting Up the RDF Semantic Graph Environment 8-14
8.3 SEM_MATCH and RDF Semantic Graph Support for OpenRDF Sesame Queries Compared 8-15
8.4 Optimized Handling of SPARQL Queries 8-16
8.4.1 Compilation of SPARQL Queries to a Single SEM_MATCH Call 8-16
8.5 Recommendations for Best Performance 8-16
8.5.1 Statement Storage and Implications for RDF Semantic Graph Support for OpenRDF Sesame Queries 8-17
8.6 Additions to the SPARQL Syntax to Support Other Features 8-17
8.6.1 SQL Hints 8-18
8.6.2 Additional WHERE Clause Predicates 8-18
8.6.3 Additional Query Options 8-18
8.7 Support for Server-Side APIs 8-19
8.7.1 Virtual Models Support 8-19
8.7.2 Connection Pooling Support 8-20
8.7.3 Semantic Model PL/SQL Interfaces 8-21
8.7.4 Inference Options 8-22
8.8 Oracle-Specific Extensions to Sesame APIs 8-22
8.8.1 Statement Uniqueness 8-22
8.8.2 Indexes and Interoperability with the RDF Semantic Graph Support for Apache Jena 8-23
8.8.3 Inference 8-24
8.8.4 Performing Analysis Operations 8-24
8.9 Using the Sesame Console with RDF Semantic Graph Support for OpenRDF Sesame 8-24
8.10 Example Queries Using RDF Semantic Graph support for OpenRDF Sesame 8-27
8.10.1 Example1.java: Basic Operations 8-28
8.10.2 Example2.java: Add a Data File (in TRIG format) 8-30
8.10.3 Example3.java: Simple Query 8-31
8.10.4 Example4.java: Simple Bulk Load 8-33
8.10.5 Example5.java: Bulk Load RDF/XML and Application Table Index Maintenance 8-35
8.10.6 Example6.java: Bulk Load With StatusListener to Handle Loading with Bad Data 8-37
8.10.7 Example7.java: Load Data from Sesame Store into Oracle Database 8-39
8.10.8 Example8.java: SPARQL ASK Query 8-41
8.10.9 Example9.java: SPARQL CONSTRUCT and DESCRIBE 8-42
8.10.10 Example10.java: Inference 8-44
8.10.11 Example11.java: Named Graph Query 8-47

8.10.12 Example12.java: Indexes on Application Table [Advanced] 8-48
8.10.13 Example13.java: Uniqueness Constraint on Application Table [Advanced] 8-50
8.10.14 Example14.java: Query Timeout and Parallel Execution [Advanced] 8-52
8.10.15 Example15.java: Get COUNT of Matches [Advanced] 8-53
8.10.16 Example16.java: Specify Bind Variable for Constant in Query Pattern [Advanced] 8-55
8.10.17 Example17.java: Specify Bind Variable for Constant in Different Position in Query Pattern [Advanced] 8-57
8.10.18 Example18.java: Build URIs from Internal Numeric IDs [Advanced] 8-58

9 User-Defined Inferencing and Querying

9.1 User-Defined Inferencing 9-1
9.1.1 Problem Solved and Benefit Provided by User-Defined Inferencing 9-1
9.1.2 API Support for User-Defined Inferencing 9-2
9.1.3 User-Defined Inference Extension Function Examples 9-4
9.2 User-Defined Functions and Aggregates 9-28
9.2.1 Data Types for User-Defined Functions and Aggregates 9-28
9.2.2 API Support for User-Defined Functions 9-29
9.2.3 API Support for User-Defined Aggregates 9-32

10 RDF Views: Relational Data as RDF

10.1 Why Use RDF Views on Relational Data? 10-1
10.2 API Support for RDF Views 10-1
10.2.1 Creating an RDF View with Direct Mapping 10-2
10.2.2 Creating an RDF View with an R2RML Mapping 10-3
10.2.3 Dropping an RDF View 10-5
10.2.4 Exporting Virtual Content of an RDF View into a Staging Table 10-5
10.3 Example: Using an RDF View with Direct Mapping 10-5
10.4 Combining Native RDF Data with Virtual RDB2RDF Data 10-8

Part II Reference and Supplementary Information

11 SEM_APIS Package Subprograms

11.1 SEM_APIS.ADD_DATATYPE_INDEX 11-1
11.2 SEM_APIS.ADD_SEM_INDEX 11-2
11.3 SEM_APIS.ALTER_DATATYPE_INDEX 11-3
11.4 SEM_APIS.ALTER_ENTAILMENT 11-3
11.5 SEM_APIS.ALTER_MODEL 11-4
11.6 SEM_APIS.ALTER_SEM_INDEX_ON_ENTAILMENT 11-5
11.7 SEM_APIS.ALTER_SEM_INDEX_ON_MODEL 11-6
11.8 SEM_APIS.ANALYZE_ENTAILMENT 11-7
11.9 SEM_APIS.ANALYZE_MODEL 11-9
11.10 SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE 11-11

11.11 SEM_APIS.CLEANUP_FAILED ... 11-13
11.12 SEM_APIS.COMPOSE_RDF_TERM ... 11-13
11.13 SEM_APIS.CREATE_ENTAILMENT ... 11-15
11.14 SEM_APIS.CREATE_RDFVIEW_MODEL ... 11-25
11.15 SEM_APIS.CREATE_RULEBASE ... 11-27
11.16 SEM_APIS.CREATE_SEM_MODEL ... 11-27
11.17 SEM_APIS.CREATE_SEM_NETWORK ... 11-28
11.18 SEM_APIS.CREATE_SOURCE_EXTERNAL_TABLE ... 11-29
11.19 SEM_APIS.CREATE_VIRTUAL_MODEL ... 11-31
11.20 SEM_APIS.DELETE_ENTAILMENT_STATS ... 11-33
11.21 SEM_APIS.DELETE_MODEL_STATS ... 11-33
11.22 SEM_APIS.DISABLE_CHANGE_TRACKING ... 11-34
11.23 SEM_APIS.DISABLE_INC_INFERENCE ... 11-35
11.24 SEM_APIS.DROP_DATATYPE_INDEX ... 11-35
11.25 SEM_APIS.DROP_ENTAILMENT ... 11-36
11.26 SEM_APIS.DROP_RDFVIEW_MODEL ... 11-37
11.27 SEM_APIS.DROP_RULEBASE ... 11-38
11.28 SEM_APIS.DROP_SEM_INDEX ... 11-38
11.29 SEM_APIS.DROP_SEM_MODEL ... 11-39
11.30 SEM_APIS.DROP_SEM_NETWORK ... 11-39
11.31 SEM_APIS.DROP_USER_INFERENCE_OBJS ... 11-40
11.32 SEM_APIS.DROP_VIRTUAL_MODEL ... 11-40
11.33 SEM_APIS.ENABLE_CHANGE_TRACKING ... 11-41
11.34 SEM_APIS.ENABLE_INC_INFERENCE ... 11-42
11.35 SEM_APIS.ESCAPE_CLOB_TERM ... 11-42
11.36 SEM_APIS.ESCAPE_CLOB_VALUE ... 11-43
11.37 SEM_APIS.ESCAPE_RDF_TERM ... 11-44
11.38 SEM_APIS.ESCAPE_RDF_VALUE ... 11-45
11.39 SEM_APIS.EXPORT_ENTAILMENT_STATS ... 11-46
11.40 SEM_APIS.EXPORT_MODEL_STATS ... 11-46
11.41 SEM_APIS.EXPORT_RDFVIEW_MODEL ... 11-47
11.42 SEM_APIS.GET_CHANGE_TRACKING_INFO ... 11-48
11.43 SEM_APIS.GET_INC_INF_INFO ... 11-49
11.44 SEM_APIS.GET_MODEL_ID ... 11-50
11.45 SEM_APIS.GET_MODEL_NAME ... 11-51
11.46 SEM_APIS.GET_TRIPLE_ID ... 11-51
11.47 SEM_APIS.GETV$DATETIMETZVAL ... 11-52
11.48 SEM_APIS.GETV$DATETZVAL ... 11-53
11.49 SEM_APIS.GETV$NUMERICVAL ... 11-54
11.50 SEM_APIS.GETV$STRINGVAL ... 11-55
11.51 SEM_APIS.GETV$TIMETZVAL ... 11-56
11.52 SEM_APIS.IMPORT_ENTAILMENT_STATS ... 11-57
11.53 SEM_APIS.IMPORT_MODEL_STATS ... 11-58

11.54 SEM_APIS.IS_TRIPLE 11-59
11.55 SEM_APIS.LOAD_INTO_STAGING_TABLE 11-60
11.56 SEM_APIS.LOOKUP_ENTAILMENT 11-61
11.57 SEM_APIS.MERGE_MODELS 11-62
11.58 SEM_APIS.PRIVILEGE_ON_APP_TABLES 11-63
11.59 SEM_APIS.PURGE_UNUSED_VALUES 11-64
11.60 SEM_APIS.REFRESH_SEM_NETWORK_INDEX_INFO 11-65
11.61 SEM_APIS.REMOVE_DUPLICATES 11-65
11.62 SEM_APIS.RENAME_ENTAILMENT 11-66
11.63 SEM_APIS.RENAME_MODEL 11-67
11.64 SEM_APIS.SET_ENTAILMENT_STATS 11-68
11.65 SEM_APIS.SET_MODEL_STATS 11-68
11.66 SEM_APIS.SWAP_NAMES 11-69
11.67 SEM_APIS.UNESCAPE_CLOB_TERM 11-70
11.68 SEM_APIS.UNESCAPE_CLOB_VALUE 11-71
11.69 SEM_APIS.UNESCAPE_RDF_TERM 11-72
11.70 SEM_APIS.UNESCAPE_RDF_VALUE 11-72
11.71 SEM_APIS.VALIDATE_ENTAILMENT 11-73
11.72 SEM_APIS.VALIDATE_MODEL 11-74
11.73 SEM_APIS.VALUE_NAME_PREFIX 11-75
11.74 SEM_APIS.VALUE_NAME_SUFFIX 11-76

12 SEM_OLS Package Subprograms

12.1 SEM_OLS.APPLY_POLICY_TO_APP_TAB 12-1
12.2 SEM_OLS.REMOVE_POLICY_FROM_APP_TAB 12-2

13 SEM_PERF Package Subprograms

13.1 SEM_PERF.DELETE_NETWORK_STATS 13-1
13.2 SEM_PERF.EXPORT_NETWORK_STATS 13-2
13.3 SEM_PERF.GATHER_STATS 13-3
13.4 SEM_PERF.IMPORT_NETWORK_STATS 13-4

14 SEM_RDFCTX Package Subprograms

14.1 SEM_RDFCTX.ADD_DEPENDENT_POLICY 14-1
14.2 SEM_RDFCTX.CREATE_POLICY 14-2
14.3 SEM_RDFCTX.DROP_POLICY 14-3
14.4 SEM_RDFCTX.MAINTAIN_TRIPLES 14-4
14.5 SEM_RDFCTX.SET_DEFAULT_POLICY 14-5
14.6 SEM_RDFCTX.SET_EXTRACTOR_PARAM 14-6

15 SEM_RDFSA Package Subprograms

15.1 SEM_RDFSA.ADD_VPD_CONSTRAINT 15-1
15.2 SEM_RDFSA.APPLY_OLS_POLICY 15-3

15.3 SEM_RDFSA.APPLY_VPD_POLICY 15-5
15.4 SEM_RDFSA.CREATE_VPD_POLICY 15-6
15.5 SEM_RDFSA.DELETE_VPD_CONSTRAINT 15-8
15.6 SEM_RDFSA.DISABLE_OLS_POLICY 15-8
15.7 SEM_RDFSA.DROP_VPD_POLICY 15-9
15.8 SEM_RDFSA.ENABLE_OLS_POLICY 15-10
15.9 SEM_RDFSA.MAINT_VPD_METADATA 15-10
15.10 SEM_RDFSA.REMOVE_OLS_POLICY 15-12
15.11 SEM_RDFSA.REMOVE_VPD_POLICY 15-12
15.12 SEM_RDFSA.RESET_MODEL_LABELS 15-13
15.13 SEM_RDFSA.SET_PREDICATE_LABEL 15-14
15.14 SEM_RDFSA.SET_RDFS_LABEL 15-15
15.15 SEM_RDFSA.SET_RESOURCE_LABEL 15-16
15.16 SEM_RDFSA.SET_RULE_LABEL 15-18

A Enabling, Downgrading, or Removing RDF Semantic Graph Support

A.1 Enabling RDF Semantic Graph Support A-1
A.1.1 Enabling RDF Semantic Graph Support in a New Database Installation A-1
A.1.2 Upgrading RDF Semantic Graph Support from Release 11.1 or 11.2 ..... A-4
A.1.3 Spatial and Partitioning Requirements A-6
A.2 Downgrading RDF Semantic Graph Support to a Previous Release A-6
A.2.1 Downgrading to Release 11.2 Semantic Technologies Support A-6
A.3 Removing RDF Semantic Graph Support A-7

B SEM_MATCH Support for Spatial Queries

B.1 ogcf:boundary B-2
B.2 ogcf:buffer B-3
B.3 ogcf:convexHull B-4
B.4 ogcf:difference B-5
B.5 ogcf:distance B-6
B.6 ogcf:envelope B-7
B.7 ogcf:getSRID B-7
B.8 ogcf:intersection B-8
B.9 ogcf:relate B-9
B.10 ogcf:sfContains B-10
B.11 ogcf:sfCrosses B-11
B.12 ogcf:sfDisjoint B-12
B.13 ogcf:sfEquals B-13
B.14 ogcf:sfIntersects B-14
B.15 ogcf:sfOverlaps B-15
B.16 ogcf:sfTouches B-16
B.17 ogcf:sfWithin B-17
B.18 ogcf:symDifference B-18

B.19 ogcf:union B-19
B.20 orageo:area B-20
B.21 orageo:buffer B-21
B.22 orageo:centroid B-22
B.23 orageo:convexHull B-22
B.24 orageo:difference B-23
B.25 orageo:distance B-24
B.26 orageo:intersection B-25
B.27 orageo:length B-26
B.28 orageo:mbr B-27
B.29 orageo:nearestNeighbor B-28
B.30 orageo:relate B-29
B.31 orageo:union B-31
B.32 orageo:withinDistance B-32
B.33 orageo:xor B-33

C Virtual Private Database Support in RDF Semantic Graph

C.1 Virtual Private Database (VPD) for RDF Data C-2
C.1.1 VPD Policy for RDF Data C-3
C.1.2 RDF Metadata for Enforcing VPD Policies C-5
C.1.3 Data Access Constraints C-7
C.1.4 RDFVPD_POLICIES View C-10
C.1.5 RDFVPD_MODELS View C-11
C.1.6 RDFVPD_POLICY_CONSTRAINTS View C-11
C.1.7 RDFVPD_PREDICATE_MDATA View C-11
C.1.8 RDFVPD_RESOURCE_REL View C-12

Glossary

Index

List of Figures

1-1 Oracle Semantic Capabilities 1-2
1-2 Inferencing 1-9
1-3 Family Tree for RDF Example 1-96
2-1 Cancer Ontology Example 2-2
6-1 Physical Versioning of Entailment (Rules Index) 6-3
7-1 Visual Representation of Analytical Function Output 7-38
7-2 Graphical Interface Main Page (index.html) 7-86
7-3 SPARQL Query Main Page Response 7-87
7-4 Graphical Interface Navigation and Browsing Page (browse.jsp) 7-88
7-5 Browsing and Navigation Page: Response 7-88
7-6 Query and Response from Clicking URI Link 7-89
7-7 XSLT Management Page 7-90
7-8 SPARQL Management Page 7-91
7-9 Import Metadata - Select Data Source 7-92
7-10 Import Metadata - Select Metadata Types 7-93
7-11 Import Metadata - Select Metadata Objects 7-94

List of Tables

1-1 MDSYS.SEM_MODEL$ View Columns ... 1-4
1-2 MDSYS.SEMM_model-name View Columns ... 1-4
1-3 MDSYS.RDF_VALUE$ Table Columns ... 1-5
1-4 MDSYS.SEMR_rulebase-name View Columns ... 1-10
1-5 MDSYS.SEM_RULEBASE_INFO View Columns ... 1-10
1-6 MDSYS.SEM_RULES_INDEX_INFO View Columns ... 1-12
1-7 MDSYS.SEM_RULES_INDEX_DATASETS View Columns ... 1-12
1-8 MDSYS.SEM_MODEL$ View Column Explanations for Virtual Models ... 1-14
1-9 MDSYS.SEM_VMODEL_INFO View Columns ... 1-14
1-10 MDSYS.SEM_VMODEL_DATASETS View Columns ... 1-15
1-11 Semantic Metadata Tables and Views ... 1-18
1-12 Built-in Functions Available for FILTER Clause ... 1-30
1-13 SEM_MATCH graphs and named_graphs Values, and Resulting Dataset Configurations ... 1-37
1-14 Built-in Aggregates ... 1-52
1-15 Property Path Syntax Constructs ... 1-59
1-16 MDSYS.SEM_NETWORK_INDEX_INFO View Columns (Partial List) ... 1-87
1-17 Data Types for Data Type Indexing ... 1-88
1-18 MDSYS.SEM_DTYPE_INDEX_INFO View Columns ... 1-89
1-19 Semantic Technology Software Objects: Old and New Names ... 1-102
2-1 PATIENTS Table Example Data ... 2-2
2-2 RDFS/OWL Vocabulary Constructs Included in Each Supported Rulebase ... 2-4
2-3 MDSYS.SEMC_entailment_name View Columns ... 2-12
4-1 MDSYS.RDFCTX_POLICIES View Columns ... 4-17
4-2 MDSYS.RDFCTX_INDEX_POLICIES View Columns ... 4-18
4-3 MDSYS.RDFCTX_INDEX_EXCEPTIONS View Columns ... 4-18
5-1 MDSYS.RDFOLS_SECURE_RESOURCE View Columns ... 5-18
7-1 Functions and Return Values for my_strlen Example ... 7-26
7-2 PL/SQL Subprograms and Corresponding RDF Semantic Graph support for Apache Jena Java Class and Methods ... 7-41
8-1 PL/SQL Subprograms and Corresponding Support for Sesame Java Class and Methods ... 8-22
11-1 Inferencing Keywords for inf_components_in Parameter ... 11-18
11-2 SEM_RDFSA Package Constants for label_gen Parameter ... 11-23
15-1 SEM_RDFSA Package Constants for rdfsa_options Parameter ... 15-4
C-1 MDSYS.RDFVPD_POLICIES View Columns ... C-10
C-2 MDSYS.RDFVPD_MODELS View Columns ... C-11
C-3 MDSYS.RDFVPD_POLICY_CONSTRAINTS View Columns ... C-11
C-4 MDSYS.RDFVPD_PREDICATE_MDATA View Columns ... C-12
C-5 MDSYS.RDFVPD_RESOURCE_REL View Columns ... C-12

Preface

Oracle Spatial and Graph RDF Semantic Graph Developer's Guide provides usage and reference information about Oracle Database Enterprise Edition support for semantic technologies, including storage, inference, and query capabilities for data and ontologies based on Resource Description Framework (RDF), RDF Schema (RDFS), and Web Ontology Language (OWL). The RDF Semantic Graph feature is licensed with the Oracle Spatial and Graph option to Oracle Database Enterprise Edition, and it requires the Oracle Partitioning option to Oracle Database Enterprise Edition.

Note:

You must perform certain actions and meet prerequisites before you can use any types, synonyms, or PL/SQL packages related to RDF Semantic Graph support. These actions and prerequisites are explained in Enabling RDF Semantic Graph Support (page A-1).

Audience

This guide is intended for those who need to use semantic technology to store, manage, and query semantic data in the database.

You should be familiar with at least the main concepts and techniques for the Resource Description Framework (RDF) and the Web Ontology Language (OWL).

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For an excellent explanation of RDF concepts, see the World Wide Web Consortium (W3C) *RDF Primer* at `http://www.w3.org/TR/rdf-primer/`.

For information about OWL, see the *OWL Web Ontology Language Reference* at `http://www.w3.org/TR/owl-ref/`.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Changes in This Release for Oracle Spatial and Graph RDF Semantic Graph Developer's Guide

This preface contains:

- Changes in Oracle Database 12c Release 1 (12.1.0.2) (page xxi)
- Changes in Oracle Database 12c Release 1 (12.1.0.1) (page xxi)
- Changes for RDF Semantic Graph Support for Apache Jena for Apache Jena 2.7.2 (page xxiii)

Changes in Oracle Database 12*c* Release 1 (12.1.0.2)

The following are changes in *Oracle Spatial and Graph RDF Semantic Graph Developer's Guide* for Oracle Database 12*c* Release 1 (12.1.0.2).

Support for SPARQL 1.1 Federated Query

SEM_MATCH supports SPARQL 1.1 Federated Query. The SERVICE construct can be used to retrieve results from a specified SPARQL endpoint URL. With this capability, you can combine local RDF data (native RDF data or RDF views of relational data) with other, possibly remote, RDF data served by a W3C standards-compliant SPARQL endpoint.

For information about support for SPARQL 1.1 Federated Query, see Graph Patterns: Support for SPARQL 1.1 Federated Query (page 1-61).

Combining Native Triple Data with Virtual RDB2RDF Triple Data

A new section (Combining Native RDF Data with Virtual RDB2RDF Data (page 10-8)) explains how you can combine native triple data with virtual RDB2RDF triple data in a single SEM_MATCH query by means of the SERVICE keyword.

Changes in Oracle Database 12*c* Release 1 (12.1.0.1)

The following are changes in *Oracle Spatial and Graph RDF Semantic Graph Developer's Guide* for Oracle Database 12*c* Release 1 (12.1.0.1).

New Features

The following features are new in this release:

- Enhanced Support for SPARQL 1.1 Constructs (page xxii)
- Enhanced Support for Virtual Models (page xxii)
- Support for User-Defined Inferencing and Querying (page xxii)
- OWL 2 EL Support (page xxiii)
- OGC GeoSPARQL Support (page xxiii)
- Ladder-Based Inference (page xxiii)
- RDF Views (page xxiii)
- Data Pump Support for Exporting and Importing a Semantic Network (page xxiii)

Enhanced Support for SPARQL 1.1 Constructs

SEM_MATCH supports the following SPARQL 1.1 constructs, as explained in Graph Patterns: Support for SPARQL 1.1 Constructs (page 1-50):

- An expanded set of functions
- Expressions in the SELECT Clause (page 1-50)
- Subqueries (page 1-51)
- Grouping and Aggregation (page 1-52)
- Negation (page 1-55)
- Value Assignment (page 1-56)
- Property Paths (page 1-59)

Enhanced Support for Virtual Models

Virtual models can now be created with arbitrary combinations of models and/or entailments, as explained in Virtual Models (page 1-13). Previously, virtual models were required to have at least one model and were limited to at most one entailment.

Virtual models can now be replaced without first being dropped. A new `REPLACE=T` option for the SEM_APIS.CREATE_VIRTUAL_MODEL (page 11-31) procedure lets you maintain access privileges for a virtual model while changing its definition. (The `REPLACE=T` option is analogous to using CREATE OR REPLACE VIEW with a view.)

Support for User-Defined Inferencing and Querying

New RDF Semantic Graph extension architectures enable the addition of user-defined capabilities:

- The inference extension architecture enables you to add user-defined inferencing to the presupplied inferencing support.
- The query extension architecture enables you to add user-defined functions and aggregates to be used in SPARQL queries, both through the SEM_MATCH table function and through the support for Apache Jena and OpenRDF Sesame.

For information about these features, see User-Defined Inferencing and Querying (page 9-1).

OWL 2 EL Support

The OWL 2 EL profile is supported by the addition of the system-defined rulebase `OWL2EL`, as explained in Supported OWL Subsets (page 2-2).

OGC GeoSPARQL Support

The OGC GeoSPARQL standard for representing and querying spatial data is now supported, as explained in OGC GeoSPARQL Support (page 1-67).

Ladder-Based Inference

Ladder-based inference is available as a convenient option for fine-grained triple-level security, as explained in Fine-Grained Security for Inferred Data and Ladder-Based Inference (LBI) (page 5-3).

RDF Views

You can create and use RDF views over relational data. Mapping relational data to RDF triples enables you to perform semantic operations conveniently, and without having to store RDF triples corresponding to the relational data. For information, see RDF Views: Relational Data as RDF (page 10-1).

Data Pump Support for Exporting and Importing a Semantic Network

Effective with Oracle Database Release 12.1, you can export and import a semantic network using the full database export and import features of the Oracle Data Pump utility, as explained in Exporting or Importing a Semantic Network Using Oracle Data Pump (page 1-82).

Deprecated Features

Support for the following in RDF Semantic Graph is deprecated in this release, and may be desupported in a future release:

- Virtual Private Database (VPD) (see Virtual Private Database Support in RDF Semantic Graph (page C-1))
- Version-enabled models (see Workspace Manager Support for RDF Data (page 6-1))

Desupported Features

Some features previously described in this document are desupported in Oracle Database 12*c* Release 1 (12.1). See *Oracle Database Upgrade Guide* for a list of desupported features.

Changes for RDF Semantic Graph Support for Apache Jena

RDF Semantic Graph support for Apache Jena supports Apache Jena 2.11.1, including jena-core-2.11.1 and jena-arq-2.11.1. This support includes the following features (most of which were included when support was added for Apache Jena 2.7.2):

- Support for Retrieving User-Friendly Java Objects from SEM_MATCH or SQL-Based Query Results (page xxiv)
- Protege Plugin for Oracle Database and Oracle NoSQL Database Enterprise Edition Release 2.0.39 (page xxiv)

- Support for Customized Data Source Name (page xxiv)
- Support for SPARQL Queries with Translated SQL Text Larger Than 29000 Bytes (page xxiv)
- Support for Oracle Database in Read-Only Mode (page xxv)
- Less Verbose Joseki Output (page xxv)
- Java APIs for Managing the Table Responsible for Terminating Long-Running SPARQL Queries (page xxv)
- Support for Apache Tomcat and JBoss Application Server (page xxv)
- Support for Fuseki 1.0.1 (page xxv)

For information about using the support for Apache Jena, see RDF Semantic Graph Support for Apache Jena (page 7-1).

Support for Retrieving User-Friendly Java Objects from SEM_MATCH or SQL-Based Query Results

With support for retrieving user-friendly Java objects from SEM_MATCH or SQL-based graph query results, you no longer need to parse and understand the subtle details embedded in projected columns like $RDFVTYP, $RDFLTYP, $RDFLANG, and $RDFCLOB; or their corresponding columns in MDSYS.RDF_VALUE$ including VALUE_TYPE, LITERAL_TYPE, LANGUAGE_TYPE, LONG_VALUE, and VALUE_NAME.

For an explanation and examples, see Retrieving User-Friendly Java Objects from SEM_MATCH or SQL-Based Query Results (page 7-12).

Protege Plugin for Oracle Database and Oracle NoSQL Database Enterprise Edition Release 2.0.39

This plugin allows an easy integration of the Protege 4.1 visual ontology editing functions with the robust semantic data management capabilities provided by Oracle Database and Oracle NoSQL Database.

The plugin .jar file and installation document are under the `protege_plugin/` directory of the release zip file.

Support for Customized Data Source Name

In previous releases, a fixed data source name *OracleSemDS* was required in the Joseki deployment. This release allows you to customize the data source name through the `oracle:dataSourceName` setting in the Joseki configuration.

You can also customize the data source name used for establishing database connections to terminate long-running SPARQL queries using query ID (see Terminating Long-Running SPARQL Queries (page 7-10) for details). To change the default data source name, edit the following JVM property:

```
-Doracle.spatial.rdf.client.jena.dsNameForQueryMgt=OracleSemDS
```

Support for SPARQL Queries with Translated SQL Text Larger Than 29000 Bytes

The S2S feature has been enhanced such that it now supports very long SPARQL queries that have translated SQL texts that are hundreds of thousands of bytes long.

However, Oracle still recommends the use of simpler (and shorter) SPARQL queries whenever possible, because simpler SPARQL queries are easier for users and application developers to understand, and they tend to be more efficient for Oracle Database to execute.

Support for Oracle Database in Read-Only Mode

The Joseki web service endpoint can now work with an Oracle Database opened in read-only mode. Queries can be answered without problems.

S2S must be used for queries to work with a read-only Oracle Database.

Less Verbose Joseki Output

This release reduces the amount of Joseki logging output. You can, however, set the trace level to 1 or higher to get more trace output for debugging purpose. For example:

```
-Doracle.spatial.rdf.client.jena.josekiTraceLevel=1
```

Java APIs for Managing the Table Responsible for Terminating Long-Running SPARQL Queries

When a request to terminate a SPARQL query is sent using the following servlet:

```
http://<hostname>:7001/joseki/querymgt?abortqid=8761
```

the request (query ID and a timestamp) is recorded in a table named ORACLE_ORARDF_QUERY_MGT_TAB in user's schema.

The following methods have been added to the class `oracle.spatial.rdf.client.jena.OracleQueryProgressMonitor` to allow users to query, add, and remove entries in the ORACLE_ORARDF_QUERY_MGT_TAB table. Details of these methods are explained in the Javadoc (`javadoc.zip` is under the `javadoc/` directory of the release zip file):

```
addQuery
deleteAllQueries
deleteQuery
listQueries
```

Support for Apache Tomcat and JBoss Application Server

Support is provided for Joseki deployment in Apache Tomcat and JBoss, in addition to Oracle WebLogic Server. For information, see Deploying Joseki in Apache Tomcat or JBoss (page 7-94).

Support for Fuseki 1.0.1

Support is provided for using Fuseki 1.0.1 to serve RDF data over HTTP.

For information, see Configuring the Fuseki-Based SPARQL Service (page 7-9).

Part I

Conceptual and Usage Information

This document has the following parts:

- Part I provides conceptual and usage information about RDF Semantic Graph.
- Part II provides reference information about RDF Semantic Graph subprograms; it also provides supplementary information in appendixes and a glossary.

Part I contains the following chapters:

- RDF Semantic Graph Overview (page 1-1)
- OWL Concepts (page 2-1)
- Simple Knowledge Organization System (SKOS) Support (page 3-1)
- Semantic Indexing for Documents (page 4-1)
- Fine-Grained Access Control for RDF Data (page 5-1)
- Workspace Manager Support for RDF Data (page 6-1)
- RDF Semantic Graph Support for Apache Jena (page 7-1)
- RDF Semantic Graph support for OpenRDF Sesame (page 8-1)
- User-Defined Inferencing and Querying (page 9-1)
- RDF Views: Relational Data as RDF (page 10-1)

1

RDF Semantic Graph Overview

This chapter describes the support in Oracle Spatial and Graph for semantic technologies, specifically Resource Description Framework (RDF) and a subset of the Web Ontology Language (OWL). These capabilities are referred to as the RDF Semantic Graph feature of Oracle Spatial and Graph. This chapter assumes that you are familiar with the major concepts associated with RDF and OWL, such as {subject, predicate, object} triples, URIs, blank nodes, plain and typed literals, and ontologies. This chapter does not explain these concepts in detail, but focuses instead on how the concepts are implemented in Oracle.

- For an excellent explanation of RDF concepts, see the World Wide Web Consortium (W3C) *RDF Primer* at `http://www.w3.org/TR/rdf-primer/`.
- For information about OWL, see the *OWL Web Ontology Language Reference* at `http://www.w3.org/TR/owl-ref/`.

The PL/SQL subprograms for working with semantic data are in the SEM_APIS package, which is documented in SEM_APIS Package Subprograms (page 11-1).

The RDF and OWL support are features of Oracle Spatial and Graph, which must be installed for these features to be used. However, the use of RDF and OWL is not restricted to spatial data.

This chapter contains the following major sections:

- Introduction to Oracle Semantic Technologies Support (page 1-2)
- Semantic Data Modeling (page 1-3)
- Semantic Data in the Database (page 1-3)
- Semantic Metadata Tables and Views (page 1-18)
- Semantic Data Types_ Constructors_ and Methods (page 1-20)
- Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22)
- Loading and Exporting Semantic Data (page 1-74)
- Using Semantic Network Indexes (page 1-86)
- Using Data Type Indexes (page 1-88)
- Managing Statistics for Semantic Models and the Semantic Network (page 1-89)
- Quick Start for Using Semantic Data (page 1-92)
- Semantic Data Examples (PL/SQL and Java) (page 1-93)
- Software Naming Changes Since Release 11.1 (page 1-102)

For information about OWL concepts and the Oracle Database support for OWL capabilities, see OWL Concepts (page 2-1).

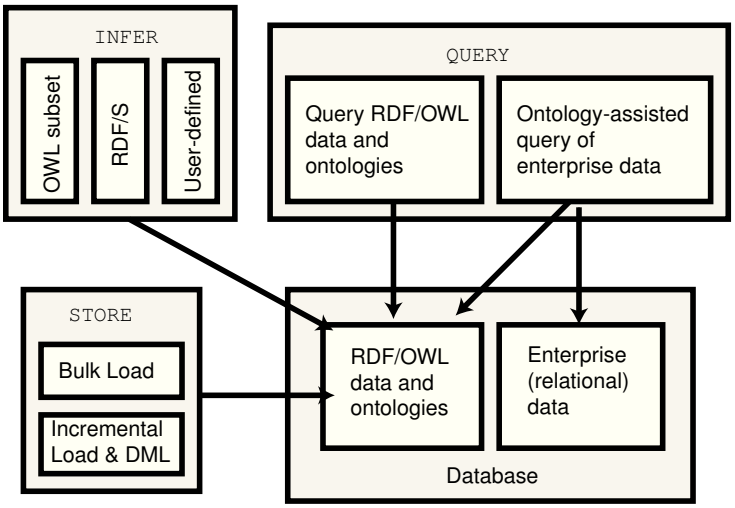
Note:

Before performing any operations described in this guide, you must enable RDF Semantic Graph support in the database and meet other prerequisites, as explained in Enabling RDF Semantic Graph Support (page A-1).

1.1 Introduction to Oracle Semantic Technologies Support

Oracle Database enables you to store semantic data and ontologies, to query semantic data and to perform ontology-assisted query of enterprise relational data, and to use supplied or user-defined inferencing to expand the power of querying on semantic data. Figure 1-1 (page 1-2) shows how these capabilities interact.

Figure 1-1 Oracle Semantic Capabilities



As shown in Figure 1-1 (page 1-2), the database contains semantic data and ontologies (RDF/OWL models), as well as traditional relational data. To load semantic data, bulk loading is the most efficient approach, although you can load data incrementally using transactional INSERT statements.

Note:

If you want to use existing semantic data from a release before Oracle Database 11.1, the data must be upgraded as described in Enabling RDF Semantic Graph Support (page A-1).

You can query semantic data and ontologies, and you can also perform ontology-assisted queries of semantic and traditional relational data to find semantic relationships. To perform ontology-assisted queries, use the SEM_RELATED operator, which is described in Using Semantic Operators to Query Relational Data (page 2-19).

You can expand the power of queries on semantic data by using inferencing, which uses rules in rulebases. Inferencing enables you to make logical deductions based on

the data and the rules. For information about using rules and rulebases for inferencing, see Inferencing: Rules and Rulebases (page 1-8).

1.2 Semantic Data Modeling

In addition to its formal semantics, semantic data has a simple data structure that is effectively modeled using a directed graph. The metadata statements are represented as triples: nodes are used to represent two parts of the triple, and the third part is represented by a directed link that describes the relationship between the nodes. The triples are stored in a semantic data network. In addition, information is maintained about specific semantic data models created by database users. A user-created **model** has a model name, and refers to triples stored in a specified table column.

Statements are expressed in triples: {subject or resource, predicate or property, object or value}. In this manual, {subject, property, object} is used to describe a triple, and the terms *statement* and *triple* may sometimes be used interchangeably. Each triple is a complete and unique fact about a specific domain, and can be represented by a link in a directed graph.

1.3 Semantic Data in the Database

There is one universe for all semantic data stored in the database. All triples are parsed and stored in the system as entries in tables under the MDSYS schema. A triple {subject, property, object} is treated as one database object. As a result, a single document containing multiple triples results in multiple database objects.

All the subjects and objects of triples are mapped to nodes in a semantic data network, and properties are mapped to network links that have their start node and end node as subject and object, respectively. The possible node types are blank nodes, URIs, plain literals, and typed literals.

The following requirements apply to the specifications of URIs and the storage of semantic data in the database:

- A subject must be a URI or a blank node.
- A property must be a URI.
- An object can be any type, such as a URI, a blank node, or a literal. (However, null values and null strings are not supported.)

1.3.1 Metadata for Models

The MDSYS.SEM_MODEL$ view contains information about all models defined in the database. When you create a model using the SEM_APIS.CREATE_SEM_MODEL (page 11-27) procedure, you specify a name for the model, as well as a table and column to hold references to the semantic data, and the system automatically generates a model ID.

Oracle maintains the MDSYS.SEM_MODEL$ view automatically when you create and drop models. Users should never modify this view directly. For example, do not use SQL INSERT, UPDATE, or DELETE statements with this view.

The MDSYS.SEM_MODEL$ view contains the columns shown in Table 1-1 (page 1-4).

Table 1-1 MDSYS.SEM_MODEL$ View Columns

Column Name	Data Type	Description
OWNER	VARCHAR2(30)	Schema of the owner of the model.
MODEL_ID	NUMBER	Unique model ID number, automatically generated.
MODEL_NAME	VARCHAR2(25)	Name of the model.
TABLE_NAME	VARCHAR2(30)	Name of the table to hold references to semantic data for the model.
COLUMN_NAME	VARCHAR2(30)	Name of the column of type SDO_RDF_TRIPLE_S in the table to hold references to semantic data for the model.
MODEL_TABLESPACE_NAME	VARCHAR2(30)	Name of the tablespace to be used for storing the triples for this model.
MODEL_TYPE	VARCHAR2(40)	A value indicating the type of RDF model: `M` for regular model; `V` for virtual model; `X` for model created to store the contents of the semantic index; or `D` for model created on relational data.

When you create a model, a view for the triples associated with the model is also created under the MDSYS schema. This view has a name in the format SEMM_*model-name*, and it is visible only to the owner of the model and to users with suitable privileges. Each MDSYS.SEMM_*model-name* view contains a row for each triple (stored as a link in a network), and it has the columns shown in Table 1-2 (page 1-4).

Table 1-2 MDSYS.SEMM_model-name View Columns

Column Name	Data Type	Description
P_VALUE_ID	NUMBER	The VALUE_ID for the text value of the predicate of the triple. Part of the primary key.
START_NODE_ID	NUMBER	The VALUE_ID for the text value of the subject of the triple. Also part of the primary key.
CANON_END_NODE_ID	NUMBER	The VALUE_ID for the text value of the canonical form of the object of the triple. Also part of the primary key.
END_NODE_ID	NUMBER	The VALUE_ID for the text value of the object of the triple
MODEL_ID	NUMBER	The ID for the RDF model to which the triple belongs.
COST	NUMBER	(Reserved for future use)
CTXT1	NUMBER	(Reserved column; can be used for fine-grained access control)
CTXT2	VARCHAR2(4000)	(Reserved for future use)

Table 1-2 (Cont.) MDSYS.SEMM_model-name View Columns

Column Name	Data Type	Description
DISTANCE	NUMBER	(Reserved for future use)
EXPLAIN	VARCHAR2(4000)	(Reserved for future use)
PATH	VARCHAR2(4000)	(Reserved for future use)
G_ID	NUMBER	The VALUE_ID for the text value of the graph name for the triple. Null indicates the default graph (see Named Graphs (page 1-16)).
LINK_ID	VARCHAR2(71)	Unique triple identifier value. (It is currently a computed column, and its definition may change in a future release.)

Note:

In Table 1-2 (page 1-4), for columns P_VALUE_ID, START_NODE_ID, END_NODE_ID, CANON_END_NODE_ID, and G_ID, the actual ID values are computed from the corresponding lexical values. However, a lexical value may not always map to the same ID value.

1.3.2 Statements

The MDSYS.RDF_VALUE$ table contains information about the subjects, properties, and objects used to represent RDF statements. It uniquely stores the text values (URIs or literals) for these three pieces of information, using a separate row for each part of each triple.

Oracle maintains the MDSYS.RDF_VALUE$ table automatically. Users should never modify this view directly. For example, do not use SQL INSERT, UPDATE, or DELETE statements with this view.

The RDF_VALUE$ table contains the columns shown in Table 1-3 (page 1-5).

Table 1-3 MDSYS.RDF_VALUE$ Table Columns

Column Name	Data Type	Description
VALUE_ID	NUMBER	Unique value ID number, automatically generated.
VALUE_TYPE	VARCHAR2(10)	The type of text information stored in the VALUE_NAME column. Possible values: `UR` for URI, `BN` for blank node, `PL` for plain literal, `PL@` for plain literal with a language tag, `PLL` for plain long literal, `PLL@` for plain long literal with a language tag, `TL` for typed literal, or `TLL` for typed long literal. A long literal is a literal with more than 4000 bytes.

Table 1-3 (Cont.) MDSYS.RDF_VALUE$ Table Columns

Column Name	Data Type	Description
VNAME_PREFIX	VARCHAR2(4000)	If the length of the lexical value is 4000 bytes or less, this column stores a prefix of a portion of the lexical value. The SEM_APIS.VALUE_NAME_PREFIX (page 11-75) function can be used for prefix computation. For example, the prefix for the portion of the lexical value `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` without the angle brackets is `http://www.w3.org/1999/02/22-rdf-syntax-ns#`.
VNAME_SUFFIX	VARCHAR2(512)	If the length of the lexical value is 4000 bytes or less, this column stores a suffix of a portion of the lexical value. The SEM_APIS.VALUE_NAME_SUFFIX (page 11-76) function can be used for suffix computation. For the lexical value mentioned in the description of the VNAME_PREFIX column, the suffix is `type`.
LITERAL_TYPE	VARCHAR2(4000)	For typed literals, the type information; otherwise, null. For example, for a row representing a creation date of 1999-08-16, the VALUE_TYPE column can contain `TL`, and the LITERAL_TYPE column can contain `http://www.w3.org/2001/XMLSchema#date`.
LANGUAGE_TYPE	VARCHAR2(80)	Language tag (for example, `fr` for French) for a literal with a language tag (that is, if VALUE_TYPE is `PL@` or `PLL@`). Otherwise, this column has a null value.
CANON_ID	NUMBER	The ID for the canonical lexical value for the current lexical value. (The use of this column may change in a future release.)
COLLISION_EXT	VARCHAR2(64)	Used for collision handling for the lexical value. (The use of this column may change in a future release.)
CANON_COLLISION_EXT	VARCHAR2(64)	Used for collision handling for the canonical lexical value. (The use of this column may change in a future release.)
LONG_VALUE	CLOB	The character string if the length of the lexical value is greater than 4000 bytes. Otherwise, this column has a null value.
VALUE_NAME	VARCHAR2(4000)	This is a computed column. If length of the lexical value is 4000 bytes or less, the value of this column is the concatenation of the values of VNAME_PREFIX column and the VNAME_SUFFIX column.

1.3.2.1 Triple Uniqueness and Data Types for Literals

Duplicate triples are not stored in the database. To check if a triple is a duplicate of an existing triple, the subject, property, and object of the incoming triple are checked against triple values in the specified model. If the incoming subject, property, and object are all URIs, an exact match of their values determines a duplicate. However, if the object of incoming triple is a literal, an exact match of the subject and property, and

a value (canonical) match of the object, determine a duplicate. For example, the following two triples are duplicates:

```
<eg:a> <eg:b> <"123"^^http://www.w3.org/2001/XMLSchema#int>
<eg:a> <eg:b> <"123"^^http://www.w3.org/2001/XMLSchema#unsignedByte>
```

The second triple is treated as a duplicate of the first, because `"123"^^<http://www.w3.org/2001/XMLSchema#int>` has an equivalent value (is canonically equivalent) to `"123"^^<http://www.w3.org/2001/XMLSchema#unsignedByte>`. Two entities are canonically equivalent if they can be reduced to the same value.

To use a non-RDF example, `A*(B-C)`, `A*B-C*A`, `(B-C)*A`, and `-A*C+A*B` all convert into the same canonical form.

Note:

Although duplicate triples and quads are not stored in the underlying table partition for the MDSYS.RDFM_<*model*> view, it is possible to have duplicate rows in an application table. For example, if a triple is inserted multiple times into an application table, it will appear once in the MDSYS.RDFM_<*model*> view, but will occupy multiple rows in the application table.

Value-based matching of lexical forms is supported for the following data types:

- STRING: plain literal, xsd:string and some of its XML Schema subtypes
- NUMERIC: xsd:decimal and its XML Schema subtypes, xsd:float, and xsd:double. (Support is not provided for float/double INF, -INF, and NaN values.)
- DATETIME: xsd:datetime, with support for time zone. (Without time zone there are still multiple representations for a single value, for example, `"2004-02-18T15:12:54"` and `"2004-02-18T15:12:54.0000"`.)
- DATE: xsd:date, with or without time zone
- OTHER: Everything else. (No attempt is made to match different representations).

Canonicalization is performed when the time zone is present for literals of type xsd:time and xsd:dateTime.

The following namespace definition is used: `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`

The first occurrence of a literal in the RDF_VALUE$ table is taken as the canonical form and given the VALUE_TYPE value of `CPL`, `CPL@`, `CTL`, `CPLL`, `CPLL@`, or `CTLL` as appropriate; that is, a `C` for canonical is prefixed to the actual value type. If a literal with the same canonical form (but a different lexical representation) as a previously inserted literal is inserted into the RDF_VALUE$ table, the VALUE_TYPE value assigned to the new insert is `PL`, `PL@`, `TL`, `PLL`, `PLL@`, or `TLL` as appropriate.

Canonically equivalent text values having different lexical representations are thus stored in the RDF_VALUE$ table; however, canonically equivalent triples are not stored in the database.

1.3.3 Subjects and Objects

RDF subjects and objects are mapped to nodes in a semantic data network. Subject nodes are the start nodes of links, and object nodes are the end nodes of links. Non-literal nodes (that is, URIs and blank nodes) can be used as both subject and object nodes. Literals can be used only as object nodes.

1.3.4 Blank Nodes

Blank nodes can be used as subject and object nodes in the semantic network. Blank node identifiers are different from URIs in that they are scoped within a semantic model. Thus, although multiple occurrences of the same blank node identifier within a single semantic model necessarily refer to the same resource, occurrences of the same blank node identifier in two different semantic models do not refer to the same resource.

In an Oracle semantic network, this behavior is modeled by requiring that blank nodes are always reused (that is, are used to represent the same resource if the same blank node identifier is used) within a semantic model, and never reused between two different models. Thus, when inserting triples involving blank nodes into a model, you must use the SDO_RDF_TRIPLE_S constructor that supports reuse of blank nodes.

1.3.5 Properties

Properties are mapped to links that have their start node and end node as subjects and objects, respectively. Therefore, a link represents a complete triple.

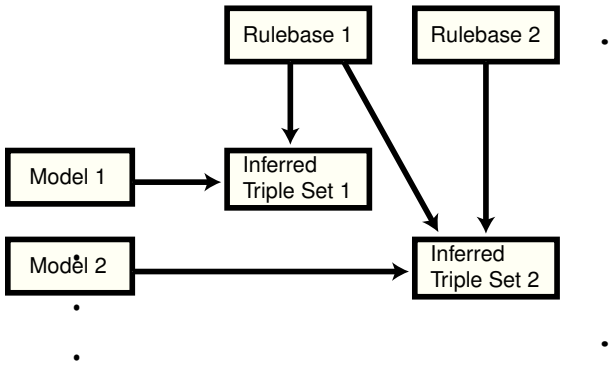
When a triple is inserted into a model, the subject, property, and object text values are checked to see if they already exist in the database. If they already exist (due to previous statements in other models), no new entries are made; if they do not exist, three new rows are inserted into the RDF_VALUE$ table (described in Statements (page 1-5)).

1.3.6 Inferencing: Rules and Rulebases

Inferencing is the ability to make logical deductions based on rules. Inferencing enables you to construct queries that perform semantic matching based on meaningful relationships among pieces of data, as opposed to just syntactic matching based on string or other values. Inferencing involves the use of rules, either supplied by Oracle or user-defined, placed in rulebases.

Figure 1-2 (page 1-9) shows triple sets being inferred from model data and the application of rules in one or more rulebases. In this illustration, the database can have any number of semantic models, rulebases, and inferred triple sets, and an inferred triple set can be derived using rules in one or more rulebases.

Figure 1-2 Inferencing



A **rule** is an object that can be applied to draw inferences from semantic data. A rule is identified by a name and consists of:

- An IF side pattern for the antecedents
- An optional filter condition that further restricts the subgraphs matched by the IF side pattern
- A THEN side pattern for the consequents

For example, the rule that *a chairperson of a conference is also a reviewer of the conference* could be represented as follows:

```
('chairpersonRule', -- rule name
 '(?r :ChairPersonOf ?c)', -- IF side pattern
 NULL,  -- filter condition
 '(?r :ReviewerOf ?c)', -- THEN side pattern
 SEM_ALIASES (SEM_ALIAS('', 'http://some.org/test/'))
)
```

In this case, the rule does not have a filter condition, so that component of the representation is NULL. For best performance, use a single-triple pattern on the THEN side of the rule. If a rule has multiple triple patterns on the THEN side, you can easily break it into multiple rules, each with a single-triple pattern, on the THEN side.

A **rulebase** is an object that contains rules. The following Oracle-supplied rulebases are provided:

- RDFS
- RDF (a subset of RDFS)
- OWLSIF (empty)
- RDFS++ (empty)
- OWL2RL (empty)
- OWLPrime (empty)
- SKOSCORE (empty)

The RDFS and RDF rulebases are created when you call the SEM_APIS.CREATE_SEM_NETWORK (page 11-28) procedure to add RDF support to the database. The RDFS rulebase implements the RDFS entailment rules, as described in the World Wide Web Consortium (W3C) *RDF Semantics* document at `http://`

www.w3.org/TR/rdf-mt/. The RDF rulebase represents the RDF entailment rules, which are a subset of the RDFS entailment rules. You can see the contents of these rulebases by examining the MDSYS.SEMR_RDFS and MDSYS.SEMR_RDF views.

You can also create user-defined rulebases using the SEM_APIS.CREATE_RULEBASE (page 11-27) procedure. User-defined rulebases enable you to provide additional specialized inferencing capabilities.

For each rulebase, a system table is created to hold rules in the rulebase, along with a system view with a name in the format MDSYS.SEMR_*rulebase-name* (for example, MDSYS.SEMR_FAMILY_RB for a rulebase named `FAMILY_RB`). You must use this view to insert, delete, and modify rules in the rulebase. Each MDSYS.SEMR_*rulebase-name* view has the columns shown in Table 1-4 (page 1-10).

Table 1-4 MDSYS.SEMR_rulebase-name View Columns

Column Name	Data Type	Description
RULE_NAME	VARCHAR2(30)	Name of the rule
ANTECEDENTS	VARCHAR2(4000)	IF side pattern for the antecedents
FILTER	VARCHAR2(4000)	Filter condition that further restricts the subgraphs matched by the IF side pattern. Null indicates no filter condition is to be applied.
CONSEQUENTS	VARCHAR2(4000)	THEN side pattern for the consequents
ALIASES	SEM_ALIASES	One or more namespaces to be used. (The SEM_ALIASES data type is described in Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).)

Information about all rulebases is maintained in the MDSYS.SEM_RULEBASE_INFO view, which has the columns shown in Table 1-5 (page 1-10) and one row for each rulebase.

Table 1-5 MDSYS.SEM_RULEBASE_INFO View Columns

Column Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the rulebase
RULEBASE_NAME	VARCHAR2(25)	Name of the rulebase
RULEBASE_VIEW_NAME	VARCHAR2(30)	Name of the view that you must use for any SQL statements that insert, delete, or modify rules in the rulebase
STATUS	VARCHAR2(30)	Contains `VALID` if the rulebase is valid, `INPROGRESS` if the rulebase is being created, or `FAILED` if a system failure occurred during the creation of the rulebase.

Example 1-1 Inserting a Rule into a Rulebase

Example 1-1 (page 1-10) creates a rulebase named `family_rb`, and then inserts a rule named `grandparent_rule` into the `family_rb` rulebase. This rule says that if a

person is the parent of a child who is the parent of a child, that person is a grandparent of (that is, has the `grandParentOf` relationship with respect to) his or her child's child. It also specifies a namespace to be used. (This example is an excerpt from Example 1-90 (page 1-96) in Example: Family Information (page 1-95).)

```
EXECUTE SEM_APIS.CREATE_RULEBASE('family_rb');

INSERT INTO mdsys.semr_family_rb VALUES(
  'grandparent_rule',
  '(?x :parentOf ?y) (?y :parentOf ?z)',
  NULL,
  '(?x :grandParentOf ?z)',
  SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')));
```

Note that the kind of grandparent rule shown in Example 1-1 (page 1-10) can be implemented using the OWL 2 property chain construct. For information about property chain handling, see Property Chain Handling (page 3-4).

Example 1-2 Using Rulebases for Inferencing

You can specify one or more rulebases when calling the SEM_MATCH table function (described in Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22)), to control the behavior of queries against semantic data. Example 1-2 (page 1-11) refers to the `family_rb` rulebase and to the `grandParentOf` relationship created in Example 1-1 (page 1-10), to find all grandfathers (grandparents who are male) and their grandchildren. (This example is an excerpt from Example 1-90 (page 1-96) in Example: Family Information (page 1-95).)

```
-- Select all grandfathers and their grandchildren from the family model.
-- Use inferencing from both the RDFS and family_rb rulebases.
SELECT x, y
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male}',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

For information about support for native OWL inferencing, see Using OWL Inferencing (page 2-5).

1.3.7 Entailments (Rules Indexes)

An **entailment** (rules index) is an object containing precomputed triples that can be inferred from applying a specified set of rulebases to a specified set of models. If a SEM_MATCH query refers to any rulebases, an entailment must exist for each rulebase-model combination in the query.

To create an entailment, use the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure. To drop (delete) an entailment, use the SEM_APIS.DROP_ENTAILMENT (page 11-36) procedure.

When you create an entailment, a view for the triples associated with the entailment is also created under the MDSYS schema. This view has a name in the format SEMI_*entailment-name*, and it is visible only to the owner of the entailment and to users with suitable privileges. Each MDSYS.SEMI_*entailment-name* view contains a row for each triple (stored as a link in a network), and it has the same columns as the SEMM_*model-name* view, which is described in Table 1-2 (page 1-4) in Metadata for Models (page 1-3).

Information about all entailments is maintained in the MDSYS.SEM_RULES_INDEX_INFO view, which has the columns shown in Table 1-6 (page 1-12) and one row for each entailment.

Table 1-6 MDSYS.SEM_RULES_INDEX_INFO View Columns

Column Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the entailment
INDEX_NAME	VARCHAR2(25)	Name of the entailment
INDEX_VIEW_NAME	VARCHAR2(30)	Name of the view that you must use for any SQL statements that insert, delete, or modify rules in the entailment
STATUS	VARCHAR2(30)	Contains `VALID` if the entailment is valid, `INVALID` if the entailment is not valid, `INCOMPLETE` if the entailment is incomplete (similar to `INVALID` but requiring less time to re-create), `INPROGRESS` if the entailment is being created, or `FAILED` if a system failure occurred during the creation of the entailment.
MODEL_COUNT	NUMBER	Number of models included in the entailment
RULEBASE_COUNT	NUMBER	Number of rulebases included in the entailment

Information about all database objects, such as models and rulebases, related to entailments is maintained in the MDSYS.SEM_RULES_INDEX_DATASETS view. This view has the columns shown in Table 1-7 (page 1-12) and one row for each unique combination of values of all the columns.

Table 1-7 MDSYS.SEM_RULES_INDEX_DATASETS View Columns

Column Name	Data Type	Description
INDEX_NAME	VARCHAR2(25)	Name of the entailment
DATA_TYPE	VARCHAR2(8)	Type of data included in the entailment. Examples: `MODEL` and `RULEBASE`
DATA_NAME	VARCHAR2(25)	Name of the object of the type in the DATA_TYPE column

Example 1-3 (page 1-12) creates an entailment named `family_rb_rix_family`, using the `family` model and the `RDFS` and `family_rb` rulebases. (This example is an excerpt from Example 1-90 (page 1-96) in Example: Family Information (page 1-95).)

Example 1-3 Creating an Entailment

```
BEGIN
  SEM_APIS.CREATE_ENTAILMENT(
    'rdfs_rix_family',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'));
END;
/
```

1.3.8 Virtual Models

A virtual model is a logical graph that can be used in a SEM_MATCH query. A virtual model is the result of a UNION or UNION ALL operation on one or more models and/or entailments.

Using a virtual model can provide several benefits:

- It can simplify management of access privileges for semantic data. For example, assume that you have created three semantic models and one entailment based on the three models and the OWLPrime rulebase. Without a virtual model, you must individually grant and revoke access privileges for each model and the entailment. However, if you create a virtual model that contains the three models and the entailment, you will only need to grant and revoke access privileges for the single virtual model.
- It can facilitate rapid updates to semantic models. For example, assume that virtual model VM1 contains model M1 and entailment R1 (that is, VM1 = M1 UNION ALL R1), and assume that semantic model M1_UPD is a copy of M1 that has been updated with additional triples and that R1_UPD is an entailment created for M1_UPD. Now, to have user queries over VM1 go to the updated model and entailment, you can redefine virtual model VM1 (that is, VM1 = M1_UPD UNION ALL R1_UPD).
- It can simplify query specification because querying a virtual model is equivalent to querying multiple models in a SEM_MATCH query. For example, assume that models m1, m2, and m3 already exist, and that an entailment has been created for m1, m2 ,and m3 using the OWLPrime rulebase. You could create a virtual model vm1 as follows:

  ```
  EXECUTE sem_apis.create_virtual_model('vm1', sem_models('m1', 'm2', 'm3'),
                                        sem_rulebases('OWLPRIME'));
  ```

  To query the virtual model, use the virtual model name as if it were a model in a SEM_MATCH query. For example, the following query on the virtual model:

  ```
  SELECT * FROM TABLE (sem_match('{…}', sem_models('vm1'), null, …));
  ```

  is equivalent to the following query on all the individual models:

  ```
  SELECT * FROM TABLE (sem_match('{…}', sem_models('m1', 'm2', 'm3'),
                                        sem_rulebases('OWLPRIME'), …));
  ```

  A SEM_MATCH query over a virtual model will query either the *SEMV* or *SEMU* view (*SEMU* by default and *SEMV* if the 'ALLOW_DUP=T' option is specified) rather than querying the UNION or UNION ALL of each model and entailment. For information about these views and options, see the reference section for the SEM_APIS.CREATE_VIRTUAL_MODEL (page 11-31) procedure.

You cannot use Oracle Workspace Manager version-enabling on a model that participates in a virtual model. (Workspace Manager support for RDF data is described in Workspace Manager Support for RDF Data (page 6-1).)

Virtual models use views (described later in this section) and add some metadata entries, but do not significantly increase system storage requirements.

To create a virtual model, use the SEM_APIS.CREATE_VIRTUAL_MODEL (page 11-31) procedure. To drop (delete) a virtual model, use the SEM_APIS.DROP_VIRTUAL_MODEL (page 11-40) procedure. A virtual model is

dropped automatically if any of its component models, rulebases, or entailment are dropped. To replace a virtual model without dropping it, use the SEM_APIS.CREATE_VIRTUAL_MODEL (page 11-31) procedure with the `REPLACE=T` option. Replacing a virtual model allows you to redefine it while maintaining any access privileges.

To query a virtual model, specify the virtual model name in the `models` parameter of the SEM_MATCH table function, as shown in Example 1-4 (page 1-15).

For information about the SEM_MATCH table function, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22), which includes information using certain attributes when querying a virtual model.

When you create a virtual model, an entry is created for it in the MDSYS.SEM_MODEL$ view, which is described in Table 1-1 (page 1-4) in Metadata for Models (page 1-3). However, the values in several of the columns are different for virtual models as opposed to semantic models, as explained in Table 1-8 (page 1-14).

Table 1-8 MDSYS.SEM_MODEL$ View Column Explanations for Virtual Models

Column Name	Data Type	Description
OWNER	VARCHAR2(30)	Schema of the owner of the virtual model
MODEL_ID	NUMBER	Unique model ID number, automatically generated. Will be a negative number, to indicate that this is a virtual model.
MODEL_NAME	VARCHAR2(25)	Name of the virtual model
TABLE_NAME	VARCHAR2(30)	Null for a virtual model
COLUMN_NAME	VARCHAR2(30)	Null for a virtual model
MODEL_TABLESPACE_NAME	VARCHAR2(30)	Null for a virtual model

Information about all virtual models is maintained in the MDSYS.SEM_VMODEL_INFO view, which has the columns shown in Table 1-9 (page 1-14) and one row for each virtual model.

Table 1-9 MDSYS.SEM_VMODEL_INFO View Columns

Column Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the virtual model
VIRTUAL_MODEL_NAME	VARCHAR2(25)	Name of the virtual model
UNIQUE_VIEW_NAME	VARCHAR2(30)	Name of the view that contains unique triples in the virtual model, or null if the view was not created
DUPLICATE_VIEW_NAME	VARCHAR2(30)	Name of the view that contains duplicate triples (if any) in the virtual model

Table 1-9 (Cont.) MDSYS.SEM_VMODEL_INFO View Columns

Column Name	Data Type	Description
STATUS	VARCHAR2(30)	Contains `VALID` if the associated entailment is valid, `INVALID` if the entailment is not valid, `INCOMPLETE` if the entailment is incomplete (similar to `INVALID` but requiring less time to re-create), `INPROGRESS` if the entailment is being created, `FAILED` if a system failure occurred during the creation of the entailment, or `NORIDX` if no entailment is associated with the virtual model. In the case of multiple entailments, the lowest status among all of the component entailments is used as the virtual model's status (`INVALID` < `INCOMPLETE` < `VALID`).
MODEL_COUNT	NUMBER	Number of models in the virtual model
RULEBASE_COUNT	NUMBER	Number of rulebases used for the virtual model
RULES_INDEX_COUNT	NUMBER	Number of entailments in the virtual model

Information about all objects (models, rulebases, and entailments) related to virtual models is maintained in the MDSYS.SEM_VMODEL_DATASETS view. This view has the columns shown in Table 1-10 (page 1-15) and one row for each unique combination of values of all the columns.

Table 1-10 MDSYS.SEM_VMODEL_DATASETS View Columns

Column Name	Data Type	Description
VIRTUAL_MODEL_NAME	VARCHAR2(25)	Name of the virtual model
DATA_TYPE	VARCHAR2(8)	Type of object included in the virtual model. Examples: `MODEL` for a semantic model, `RULEBASE` for a rulebase, or `RULEIDX` for an entailment
DATA_NAME	VARCHAR2(25)	Name of the object of the type in the DATA_TYPE column

Example 1-4 Querying a Virtual Model

```
SELECT COUNT(protein)
  FROM TABLE (SEM_MATCH (
    '{?protein rdf:type :Protein .
      ?protein :citation ?citation .
      ?citation :author "Bairoch A."}',
    SEM_MODELS('UNIPROT_VM'),
    NULL,
    SEM_ALIASES(SEM_ALIAS('', 'http://purl.uniprot.org/core/')),
    NULL,
    NULL,
    'ALLOW_DUP=T'));
```

1.3.9 Named Graphs

RDF Semantic Graph supports the use of named graphs, which are described in the "RDF Dataset" section of the W3C *SPARQL Query Language for RDF* recommendation (`http://www.w3.org/TR/rdf-sparql-query/#rdfDataset`).

This support is provided by extending an RDF triple consisting of the traditional subject, predicate, and object, to include an additional component to represent a **graph name**. The extended RDF triple, despite having four components, will continue to be referred to as an *RDF triple* in this document. In addition, the following terms are sometimes used:

- **N-Triple** is a format that does not allow extended triples. Thus, n-triples can include only triples with three components.
- **N-Quad** is a format that allows both "regular" triples (three components) and extended triples (four components, including the graph name). For more information, see `http://www.w3.org/TR/2013/NOTE-n-quads-20130409/`.

  To load a file containing extended triples (possibly mixed with regular triples) into an Oracle database, the input file must be in N-Quad format.

The graph name component of an RDF triple must either be null or a URI. If it is null, the RDF triple is said to belong to a **default graph**; otherwise it is said to belong to a named graph whose name is designated by the URI.

Additionally, to support named graphs in SDO_RDF_TRIPLE_S object type (described in Semantic Data Types_ Constructors_ and Methods (page 1-20)), a new syntax is provided for specifying a model-graph, that is, a combination of model and graph (if any) together, and the RDF_M_ID attribute holds the identifier for a model-graph: a combination of model ID and value ID for the graph (if any). The name of a model-graph is specified as *model_name*, and if a graph is present, followed by the colon (`:`) separator character and the graph name (which must be a URI and enclosed within angle brackets < >).

For example, in a medical data set the named graph component for each RDF triple might be a URI based on patient identifier, so there could be as many named graphs as there are unique patients, with each named graph consisting of data for a specific patient.

For information about performing specific operations with named graphs, see the following:

- Using constructors and methods: Semantic Data Types_ Constructors_ and Methods (page 1-20)
- Loading: Loading N-Quad Format Data into a Staging Table Using an External Table (page 1-76) and Loading Data into Named Graphs Using INSERT Statements (page 1-79)
- Querying: GRAPH Keyword Support (page 1-37) and Expressions in the SELECT Clause (page 1-50)
- Inferencing: Using Named Graph Based Inferencing (Global and Local) (page 2-15)

1.3.9.1 Data Formats Related to Named Graph Support

TriG (http://wifo5-03.informatik.uni-mannheim.de/bizer/trig/) and N-QUADS (http://www.w3.org/TR/2013/NOTE-n-quads-20130409/) are two popular data formats that provide graph names (or context) to triple data. (As of November 2011, neither format was a standard.) The graph names (context) can be used in a variety of different ways. Typical usage includes, but is not limited to, the grouping of triples for ease of management, localized query, localized inference, and provenance.

Example 1-5 RDF Data Encoded in TriG Format

Example 1-5 (page 1-17) shows an RDF data set encoded in TriG format. It contains a default graph and a named graph.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

# Default graph
{
  <http://my.com/John> dc:publisher <http://publisher/Xyz> .
}

# A named graph
<http://my.com/John> {
  <http://my.com/John> foaf:name "John Doe" .
}
```

When loading the TriG file from Example 1-5 (page 1-17) into a `DatasetGraphOracleSem` object (for example, using Example 7-12 (page 7-46) in Bulk Loading Using RDF Semantic Graph Support for Apache Jena (page 7-44), but replacing the constant `"N-QUADS"` with `"TRIG"`), the triples in the default graph will be loaded into Oracle Database as triples with null graph names, and the triples in the named graphs will be loaded into Oracle Database with the designated graph names.

Example 1-6 N-QUADS Format Representation

N-QUADS format is a simple extension of the existing N-TRIPLES format by adding an optional fourth column (graph name or context). Example 1-6 (page 1-17) shows the N-QUADS format representation of the TriG file from Example 1-5 (page 1-17).

```
<http://my.com/John> <http://purl.org/dc/elements/1.1/publisher> <http://publisher/
Xyz> .
<http://my.com/John> <http://xmlns.com/foaf/0.1/name> "John Doe" <http://my.com/John>
```

When loading an N-QUADS file into a `DatasetGraphOracleSem` object (see Example 7-12 (page 7-46)), lines without the fourth column will be loaded into Oracle Database as triples with null graph names, and lines with a fourth column will be loaded into Oracle Database with the designated graph names.

1.3.10 Semantic Data Security Considerations

The following database security considerations apply to the use of semantic data:

- When a model or entailment is created, the owner gets the SELECT privilege with the GRANT option on the associated view. Users that have the SELECT privilege on these views can perform SEM_MATCH queries against the associated model or entailment.

- When a rulebase is created, the owner gets the SELECT, INSERT, UPDATE, and DELETE privileges on the rulebase, with the GRANT option. Users that have the SELECT privilege on a rulebase can create an entailment that includes the rulebase. The INSERT, UPDATE, and DELETE privileges control which users can modify the rulebase and how they can modify it.
- To perform data manipulation language (DML) operations on a model, a user must have DML privileges for the corresponding base table.
- The creator of the base table corresponding to a model can grant privileges to other users.
- To perform data manipulation language (DML) operations on a rulebase, a user must have the appropriate privileges on the corresponding database view.
- The creator of a model can grant SELECT privileges on the corresponding database view to other users.
- A user can query only those models for which that user has SELECT privileges to the corresponding database views.
- Only the creator of a model or a rulebase can drop it.

1.4 Semantic Metadata Tables and Views

Oracle Database maintains several tables and views in the MDSYS schema to hold metadata related to semantic data. (Some of these tables and views are created by the SEM_APIS.CREATE_SEM_NETWORK (page 11-28) procedure, as explained in Quick Start for Using Semantic Data (page 1-92), and some are created only as needed.) Table 1-11 (page 1-18) lists the tables and views in alphabetical order. (In addition, several tables and views are created for Oracle internal use, and these are accessible only by users with DBA privileges.)

Table 1-11 Semantic Metadata Tables and Views

Name	Contains Information About	Described In
RDF_VALUE$	Subjects, properties, and objects used to represent statements	Statements (page 1-5)
RDFOLS_SECURE_RESOURCE	Resources secured with Oracle Label Security (OLS) policies and the sensitivity labels associated with these resources	RDFOLS_SECURE_RESOURCE View (page 5-18)
RDFVPD_MODELS	RDF models and their associated VPD policies	RDFVPD_MODELS View (page C-11)
RDFVPD_POLICIES	All VPD policies defined in the schema or the policies to which the user has FULL access	RDFVPD_POLICIES View (page C-10)
RDFVPD_POLICY_CONSTRAINTS	Constraints defined in the VPD policy that are accessible to the current user	RDFVPD_PREDICATE_MDATA View (page C-11)

Table 1-11 (Cont.) Semantic Metadata Tables and Views

Name	Contains Information About	Described In
RDFVPD_PREDICATE_MDATA	Predicate metadata associated with a VPD policy	RDFVPD_PREDICATE_MDATA View (page C-11)
RDFVPD_RESOURCE_REL	Subclass, subproperty, and equivalence property relationships that are defined between resources in a VPD policy	RDFVPD_RESOURCE_REL View (page C-12)
SEM_DTYPE_INDEX_INFO	All data type indexes in the network	Using Data Type Indexes (page 1-88)
SEM_MODEL$	All models defined in the database	Metadata for Models (page 1-3)
SEM_NETWORK_INDEX_INFO$	Semantic network indexes	MDSYS.SEM_NETWORK_INDEX_INFO View (page 1-87)
SEM_RULEBASE_INFO	Rulebases	Inferencing: Rules and Rulebases (page 1-8)
SEM_RULES_INDEX_DATASETS	Database objects used in entailments	Entailments (Rules Indexes) (page 1-11)
SEM_RULES_INDEX_INFO	Entailments (rules indexes)	Entailments (Rules Indexes) (page 1-11)
SEM_VMODEL_INFO	Virtual models	Virtual Models (page 1-13)
SEM_VMODEL_DATASETS	Database objects used in virtual models	Virtual Models (page 1-13)
SEMCL_*entailment-name*	`owl:sameAs` clique members and canonical representatives	Optimizing owl:sameAs Inference (page 2-11)
SEMI_*entailment-name*	Triples in the specified entailment	Entailments (Rules Indexes) (page 1-11)

Table 1-11 (Cont.) Semantic Metadata Tables and Views

Name	Contains Information About	Described In
SEMM_*model-name*	Triples in the specified model	Metadata for Models (page 1-3)
SEMR_*rulebase-name*	Rules in the specified rulebase	Inferencing: Rules and Rulebases (page 1-8)
SEMU_*virtual-model-name*	Unique triples in the virtual model	Virtual Models (page 1-13)
SEMV_*virtual-model-name*	Triples in the virtual model	Virtual Models (page 1-13)

1.5 Semantic Data Types, Constructors, and Methods

The SDO_RDF_TRIPLE object type represents semantic data in triple format, and the SDO_RDF_TRIPLE_S object type (the _S for storage) stores persistent semantic data in the database. The SDO_RDF_TRIPLE_S type has references to the data, because the actual semantic data is stored only in the central RDF schema. This type has methods to retrieve the entire triple or part of the triple.

Note:

Blank nodes are always reused within an RDF model and cannot be reused across models

The SDO_RDF_TRIPLE type is used to display triples, whereas the SDO_RDF_TRIPLE_S type is used to store the triples in database tables.

The SDO_RDF_TRIPLE object type has the following attributes:

```
SDO_RDF_TRIPLE (
  subject VARCHAR2(4000),
  property VARCHAR2(4000),
  object VARCHAR2(10000))
```

The SDO_RDF_TRIPLE_S object type has the following attributes:

```
SDO_RDF_TRIPLE_S (
  RDF_C_ID  NUMBER,  -- Canonical object value ID
  RDF_M_ID NUMBER,  -- Model (or Model-Graph) ID
  RDF_S_ID  NUMBER,  -- Subject value ID
  RDF_P_ID NUMBER, -- Property value ID
  RDF_O_ID  NUMBER)  -- Object value ID
```

The SDO_RDF_TRIPLE_S type has the following methods that retrieve the name of the RDF model (or model-graph), a triple, or a part (subject, property, or object) of a triple:

```
GET_MODEL() RETURNS VARCHAR2
GET_TRIPLE() RETURNS SDO_RDF_TRIPLE
GET_SUBJECT() RETURNS VARCHAR2
GET_PROPERTY() RETURNS VARCHAR2
GET_OBJECT() RETURNS CLOB
```

Example 1-7 (page 1-21) shows the SDO_RDF_TRIPLE_S methods.

Example 1-7 SDO_RDF_TRIPLE_S Methods

```
SELECT a.triple.GET_MODEL() AS model_graph, a.triple.GET_TRIPLE() AS triple
  FROM articles_rdf_data a WHERE a.id = 99;

MODEL_GRAPH
--------------------------------------------------------------------------------
TRIPLE(SUBJECT, PROPERTY, OBJECT)
--------------------------------------------------------------------------------
ARTICLES:<http://examples.com/ns#Graph1>
SDO_RDF_TRIPLE('<http://nature.example.com/Article101>', '<http://purl.org/dc/
elements/1.1/creator>', '"John Smith"')

SELECT a.triple.GET_TRIPLE() AS triple
  FROM articles_rdf_data a WHERE a.id = 1;

TRIPLE(SUBJECT, PROPERTY, OBJECT)
--------------------------------------------------------------------------------
SDO_RDF_TRIPLE('<http://nature.example.com/Article1>', '<http://purl.org/dc/elem
ents/1.1/title>', '<All about XYZ>')

SELECT a.triple.GET_SUBJECT() AS subject
  FROM articles_rdf_data a WHERE a.id = 1;

SUBJECT
--------------------------------------------------------------------------------
<http://nature.example.com/Article1>

SELECT a.triple.GET_PROPERTY() AS property
  FROM articles_rdf_data a WHERE a.id = 1;

PROPERTY
--------------------------------------------------------------------------------
<http://purl.org/dc/elements/1.1/title>

SELECT a.triple.GET_OBJECT() AS object
  FROM articles_rdf_data a WHERE a.id = 1;

OBJECT
--------------------------------------------------------------------------------
<All about XYZ>
```

1.5.1 Constructors for Inserting Triples

The following constructor formats are available for inserting triples into a model table. The only difference is that in the second format the data type for the object is CLOB, to accommodate very long literals.

```
SDO_RDF_TRIPLE_S (
  model_name VARCHAR2, -- Model name
  subject    VARCHAR2, -- Subject
  property   VARCHAR2, -- Property
  object     VARCHAR2) -- Object
  RETURN     SELF;
```

```
SDO_RDF_TRIPLE_S (
  model_name VARCHAR2, -- Model name
  subject    VARCHAR2, -- Subject
  property   VARCHAR2, -- Property
  object     CLOB) -- Object
  RETURN SELF;

GET_OBJ_VALUE() RETURN VARCHAR2;
```

Example 1-8 (page 1-22) uses the first constructor format to insert several triples.

Example 1-8 SDO_RDF_TRIPLE_S Constructor to Insert Triples

```
INSERT INTO articles_rdf_data VALUES (2,
  SDO_RDF_TRIPLE_S ('articles','<http://nature.example.com/Article1>',
    '<http://purl.org/dc/elements/1.1/creator>',
    '"Jane Smith"'));

INSERT INTO articles_rdf_data VALUES (98,
  SDO_RDF_TRIPLE_S ('articles:<http://examples.com/ns#Graph1>',
    '<http://nature.example.com/Article102>',
    '<http://purl.org/dc/elements/1.1/creator>',
    '_:b1'));

INSERT INTO articles_rdf_data VALUES (97,
  SDO_RDF_TRIPLE_S ('articles:<http://examples.com/ns#Graph1>',
    '_:b2',
    '<http://purl.org/dc/elements/1.1/creator>',
    '_:b1'));
```

1.6 Using the SEM_MATCH Table Function to Query Semantic Data

To query semantic data, use the SEM_MATCH table function. This function has the following attributes:

```
SEM_MATCH(
  query        VARCHAR2,
  models       SEM_MODELS,
  rulebases    SEM_RULEBASES,
  aliases      SEM_ALIASES,
  filter       VARCHAR2,
  index_status VARCHAR2,
  options      VARCHAR2,
  graphs       SEM_GRAPHS,
  named_graphs SEM_GRAPHS
 ) RETURN ANYDATASET;
```

The `query` attribute is required. The other attributes are optional (that is, each can be a null value).

The `query` attribute is a string literal (or concatenation of string literals) with one or more triple patterns, usually containing variables. (The `query` attribute cannot be a bind variable or an expression involving a bind variable.) A triple pattern is a triple of atoms followed by a period. Each atom can be a variable (for example, `?x`), a qualified name (for example, `rdf:type`) that is expanded based on the default namespaces and the value of the aliases attribute, or a full URI (for example, `<http://www.example.org/family/Male>`). In addition, the third atom can be a numeric literal (for example, `3.14`), a plain literal (for example, `"Herman"`), a language-tagged plain literal (for example, `"Herman"@en`), or a typed literal (for example, `"123"^^xsd:int`).

For example, the following `query` attribute specifies three triple patterns to find grandfathers (that is, grandparents who are also male) and the height of each of their grandchildren:

```
'{ ?x :grandParentOf ?y . ?x rdf:type :Male . ?y :height ?h }'
```

The `models` attribute identifies the model or models to use. Its data type is SEM_MODELS, which has the following definition: `TABLE OF VARCHAR2(25)`. If you are querying a virtual model, specify only the name of the virtual model and no other models. (Virtual models are explained in Virtual Models (page 1-13).)

The `rulebases` attribute identifies one or more rulebases whose rules are to be applied to the query. Its data type is SDO_RDF_RULEBASES, which has the following definition: `TABLE OF VARCHAR2(25)`. If you are querying a virtual model, this attribute must be null.

The `aliases` attribute identifies one or more namespaces, in addition to the default namespaces, to be used for expansion of qualified names in the query pattern. Its data type is SEM_ALIASES, which has the following definition: `TABLE OF SEM_ALIAS`, where each SEM_ALIAS element identifies a namespace ID and namespace value. The SEM_ALIAS data type has the following definition: `(namespace_id VARCHAR2(30), namespace_val VARCHAR2(4000))`

The following default namespaces (`namespace_id` and `namespace_val` attributes) are used by the SEM_MATCH table function and the SEM_CONTAINS and SEM_RELATED operators:

```
('ogc', 'http://www.opengis.net/ont/geosparql#')
('ogcf', 'http://www.opengis.net/def/function/geosparql/')
('ogcgml', 'http://www.opengis.net/ont/gml#')
('ogcsf', 'http://www.opengis.net/ont/sf#')
('orardf', 'http://xmlns.oracle.com/rdf/')
('orageo', 'http://xmlns.oracle.com/rdf/geo/')
('owl',    'http://www.w3.org/2002/07/owl#')
('rdf',    'http://www.w3.org/1999/02/22-rdf-syntax-ns#')
('rdfs',   'http://www.w3.org/2000/01/rdf-schema#')
('xsd',    'http://www.w3.org/2001/XMLSchema#')
```

You can override any of these defaults by specifying the `namespace_id` value and a different `namespace_val` value in the `aliases` attribute.

The `filter` attribute identifies any additional selection criteria. If this attribute is not null, it should be a string in the form of a `WHERE` clause without the `WHERE` keyword. For example: `'(h >= ''6'')'` to limit the result to cases where the height of the grandfather's grandchild is 6 or greater (using the example of triple patterns earlier in this section).

Note:

Instead of using the `filter` attribute, you are encouraged to use the FILTER keyword inside your query pattern whenever possible (as explained in Graph Patterns: Support for Curly Brace Syntax_ and OPTIONAL_ FILTER_ UNION_ and GRAPH Keywords (page 1-30)). Using the FILTER keyword is likely to give better performance because of internal optimizations. The `filter` argument, however, can be useful if you require SQL constructs that cannot be expressed with the FILTER keyword.

The `index_status` attribute lets you query semantic data even when the relevant entailment does not have a valid status. (If you are querying a virtual model, this attribute refers to the entailment associated with the virtual model.) If this attribute is null, the query returns an error if the entailment does not have a valid status. If this attribute is not null, it must be the string `INCOMPLETE` or `INVALID`. For an explanation of query behavior with different `index_status` values, see Performing Queries with Incomplete or Invalid Entailments (page 1-29).

The `options` attribute identifies options that can affect the results of queries. Options are expressed as keyword-value pairs. The following options are supported:

- `ALL_BGP_HASH` and `ALL_BGP_NL` are global query optimizer hints that specify that all inter-BGP joins (for example. the join between the root BGP and an OPTIONAL BGP) should use the specified join type. (BGP stands for *basic graph pattern*. From the W3C SPARQL Query Language for RDF Recommendation: "SPARQL graph pattern matching is defined in terms of combining the results from matching basic graph patterns. A sequence of triple patterns interrupted by a filter comprises a single basic graph pattern. Any graph pattern terminates a basic graph pattern."

  The `BGP_JOIN(USE_NL)` and `BGP_JOIN(USE_HASH)` HINT0 query optimizer hints can be used to control the join type with finer granularity.

  Example 1-14 (page 1-35) shows the ALL_BGP_HASH option used in a SEM_MATCH query.

- `ALL_LINK_HASH` and `ALL_LINK_NL` are global query optimizer hints that specify the join type for all RDF_LINK$ joins (that is, all joins between triple patterns within a BGP). `ALL_LINK_HASH` and `ALL_LINK_NL` can also be used within a HINT0 query optimizer hint for finer granularity.

- `ALL_MAX_PP_DEPTH(n)` is a global query optimizer hint that sets the maximum depth to use when evaluating * and + property path operators. The default value is 10. The `MAP_PP_DEPTH(n)` HINT0 hint can be used to specify maximum depth with finer granularity.

- `ALL_ORDERED` is a global query optimizer hint that specifies that the triple patterns in each BGP in the query should be evaluated in order.

  Example 1-14 (page 1-35) shows the ALL_ORDERED option used in a SEM_MATCH query.

- `ALL_USE_PP_HASH` and `ALL_USE_PP_NL` are global query optimizer hints that specify the join type to use when evaluating property path expressions. The `USE_PP_HASH` and `USE_PP_NL` HINT0 hints can be used for specifying join type with finer granularity.

- `ALLOW_DUP=T` generates an underlying SQL statement that performs a "union all" instead of a union of the semantic models and inferred data (if applicable). This option may introduce more rows (duplicate triples) in the result set, and you may need to adjust the application logic accordingly. If you do not specify this option, duplicate triples are automatically removed across all the models and inferred data to maintain the set semantics of merged RDF graphs; however, removing duplicate triples increases query processing time. In general, specifying `'ALLOW_DUP=T'` improves performance significantly when multiple semantic models are involved in a SEM_MATCH query.

If you are querying a virtual model, specifying `ALLOW_DUP=T` causes the SEMV_*vm_name* view to be queried; otherwise, the SEMU_*vm_name* view is queried.

- `ALLOW_PP_DUP=T` allows duplicate results for + and * property path queries. Allowing duplicate results may return the first result rows faster.
- `CLOB_AGG_SUPPORT=T` enables support for CLOB values for the following aggregates: MIN, MAX, GROUP_CONCAT, SAMPLE. Note that enabling CLOB support incurs a significant performance penalty.
- `CLOB_EXP_SUPPORT=T` enables support for CLOB values for some built-in SPARQL functions. Note that enabling CLOB support incurs a significant performance penalty.
- `CONSTRUCT_STRICT=T` eliminates invalid RDF triples from the result of SPARQL CONSTRUCT or SPARQL DESCRIBE syntax queries. RDF triples with literals in the subject position or literals or blank nodes in the predicate position are considered invalid.
- `CONSTRUCT_UNIQUE=T` eliminates duplicate RDF triples from the result of SPARQL CONSTRUCT or SPARQL DESCRIBE syntax queries.
- `DO_UNESCAPE=T` causes characters in the following return columns to be unescaped according to the W3C N-Triples specification (`http://www.w3.org/TR/rdf-testcases/#ntriples`): var, var$_PREFIX, var $_SUFFIX, var$RDFCLOB, var$RDFLTYP, var$RDFLANG, and var$RDFTERM.

  See also the reference information for SEM_APIS.ESCAPE_CLOB_TERM (page 11-42), SEM_APIS.ESCAPE_CLOB_VALUE (page 11-43), SEM_APIS.ESCAPE_RDF_TERM (page 11-44), SEM_APIS.ESCAPE_RDF_VALUE (page 11-45), SEM_APIS.UNESCAPE_CLOB_TERM (page 11-70), SEM_APIS.UNESCAPE_CLOB_VALUE (page 11-71), SEM_APIS.UNESCAPE_RDF_TERM (page 11-72), and SEM_APIS.UNESCAPE_RDF_VALUE (page 11-72).
- `FINAL_VALUE_HASH` and `FINAL_VALUE_NL` are global query optimizer hints that specify the join method that should be used to obtain the lexical values for any query variables that are not used in a FILTER clause.
- `GRAPH_MATCH_UNNAMED=T` allows unnamed triples (null `G_ID`) to be matched inside GRAPH clauses. That is, two triples will satisfy the graph join condition if their graphs are equal or if one or both of the graphs are null. This option may be useful when your dataset includes unnamed TBOX triples or unnamed entailed triples.
- `HINT0={<hint-string>}` (pronounced and written "hint" and the number zero) specifies one or more keywords with hints to influence the execution plan and results of queries. Conceptually, a graph pattern with *n* triple patterns and referring to *m* distinct variables results in an (*n*+*m*)-way join: *n*-way self-join of the target RDF model or models and optionally the corresponding entailment, and then *m* joins with RDF_VALUE$ for looking up the values for the *m* variables. A hint specification affects the join order and join type used for the query execution.

  The hint specification, <*hint-string*>, uses keywords, some of which have parameters consisting of a sequence or set of aliases, or references, for individual triple patterns and variables used in the query. Aliases for triple patterns are of the form t*i* where *i* refers to the 0-based ordinal numbers of triple patterns in the

query. For example, the alias for the first triple pattern in a query is `t0`, the alias for the second one is `t1`, and so on. Aliases for the variables used in a query are simply the names of those variables. Thus, `?x` will be used in the hint specification as the alias for a variable `?x` used in the graph pattern.

Hints used for influencing query execution plans include LEADING(<*sequence of aliases*>), USE_NL(<*set of aliases*>), USE_HASH(<*set of aliases*>), and INDEX(<*alias*> <*index_name*>). These hints have the same format and basic meaning as hints in SQL statements, which are explained in *Oracle Database SQL Language Reference*.

Example 1-10 (page 1-28) shows the HINT0 option used in a SEM_MATCH query.

- `HTTP_METHOD=POST_PAR` indicates that the HTTP POST method with URL-encoded parameters pass should be used for the SERVICE request. The default option for requests is the HTTP GET method. For more information about SPARQL protocol, see `http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/#protocol`.

- `INF_ONLY=T` queries only the entailed graph for the specified models and rulebases.

- `PLUS_RDFT=T` can be used with SPARQL SELECT syntax (see Expressions in the SELECT Clause (page 1-50)) to additionally return a *var*$RDFTERM CLOB column for each projected query variable. The value for this column is equivalent to the result of SEM_APIS.COMPOSE_RDF_TERM (page 11-13)(*var*, *var* $RDFVTYP, *var*$RDFLTYP, *var*$RDFLANG, *var*$RDFCLOB). When using this option, the return columns for each variable var will be *var*, *var*$RDFVID, *var* $_PREFIX, *var*$_SUFFIX, *var*$RDFVTYP, *var*$RDFCLOB, *var*$RDFLTYP, *var* $RDFLANG, and *var*$RDFTERM.

- `PLUS_RDFT=VC` can be used with SPARQL SELECT syntax (see Expressions in the SELECT Clause (page 1-50)) to additionally return a var$RDFTERM VARCHAR2(4000) column for each projected query variable. The value for this column is equivalent to the result of SEM_APIS.COMPOSE_RDF_TERM (page 11-13)(var, var$RDFVTYP, var$RDFLTYP, var$RDFLANG). When using this option, the return columns for each variable var will be var, var$RDFVID, var $_PREFIX, var$_SUFFIX, var$RDFVTYP, var$RDFCLOB, var$RDFLTYP, var $RDFLANG, and var$RDFTERM.

- `PROJ_EXACT_VALUES=T` disables canonicalization of values returned from functions and of constant values used in value assignment statements. Such values are canonicalized by default.

- `SERVICE_CLOB=F` sets the column values of var$RDFCLOB to null instead of saving values when calling the service. If CLOB data is not needed in your application, performance can be improved by using this option to skip CLOB processing.

- `SERVICE_ESCAPE=F` disables character escaping for RDF literal values returned by SPARQL SERVICE calls. RDF literal values are escaped by default. If character escaping is not relevant for your application, performance can be improved by disabling character escaping.

- `SERVICE_JPDWN=T` is a query optimizer hint for using nested loop join in SPARQL SERVICE. Example 1-70 (page 1-63) shows the `SERVICE_JPDWN=T` option used in a SEM_MATCH query.

- `SERVICE_PROXY=<proxy-string>` sets a proxy address to be used when performing http connections. The given proxy-string will be used in SERVICE queries. Example 1-73 (page 1-64) shows a SEM_MATCH query including a proxy address.
- `STRICT_AGG_CARD=T` uses SPARQL semantics (one null row) instead of SQL semantics (zero rows) for aggregate queries with graph patterns that fail to match. This option incurs a slight performance penalty.
- `STRICT_DEFAULT=T` restricts the default graph to unnamed triples when no dataset information is specified.

The `graphs` attribute specifies the set of named graphs from which to construct the default graph for a SEM_MACH query. Its data type is SEM_GRAPHS, which has the following definition: `TABLE OF VARCHAR2(4000)`. The default value for this attribute is `NULL`. When `graphs` is `NULL`, the "union all" of all graphs in the set of query models is used as the default graph.

The `named_graphs` attribute specifies the set of named graphs that can be matched by a GRAPH clause. Its data type is SEM_GRAPHS, which has the following definition: `TABLE OF VARCHAR2(4000)`. The default value for this attribute is `NULL`. When named_graphs is `NULL`, all named graphs in the set of query models can be matched by a GRAPH clause.

The SEM_MATCH table function returns an object of type ANYDATASET, with elements that depend on the input variables. In the following explanations, *var* represents the name of a variable used in the query. For each variable *var*, the result elements have the following attributes: *var*, *var*$RDFVID, *var*$_PREFIX, *var*$_SUFFIX, *var*$RDFVTYP, *var*$RDFCLOB, *var*$RDFLTYP, and *var*$RDFLANG.

In such cases, *var* has the lexical value bound to the variable, *var*$RDFVID has the VALUE_ID of the value bound to the variable, *var*$_PREFIX and *var*$_SUFFIX are the *prefix* and *suffix* of the value bound to the variable, *var*$RDFVTYP indicates the type of value bound to the variable (`URI`, `LIT` [literal], or `BLN` [blank node]), *var*$RDFCLOB has the lexical value bound to the variable if the value is a long literal, *var*$RDFLTYP indicates the type of literal bound if a literal is bound, and *var*$RDFLANG has the language tag of the bound literal if a literal with language tag is bound. *var*$RDFCLOB is of type CLOB, while all other attributes are of type VARCHAR2.

For a literal value or a blank node, its prefix is the value itself and its suffix is null. For a URI value, its prefix is the left portion of the value up to and including the rightmost occurrence of any of the three characters / (slash), # (pound), or : (colon), and its suffix is the remaining portion of the value to the right. For example, the prefix and suffix for the URI value `http://www.example.org/family/grandParentOf` are `http://www.example.org/family/` and `grandParentOf`, respectively.

Along with columns for variable values, a SEM_MATCH query that uses SPARQL SELECT syntax returns one additional NUMBER column, SEM$ROWNUM, which can be used to ensure the correct result ordering for queries that involve a SPARQL ORDER BY clause.

A SEM_MATCH query that uses SPARQL ASK syntax returns the columns ASK, ASK $RDFVID, ASK$_PREFIX, ASK$_SUFFIX, ASK$RDFVTYP, ASK$RDFCLOB, ASK $RDFLTYP, ASK$RDFLANG, and SEM$ROWNUM. This is equivalent to a SPARQL SELECT syntax query that projects a single `?ask` variable.

A SEM_MATCH query that uses SPARQL CONSTRUCT or SPARQL DESCRIBE syntax returns columns that contain RDF triple data rather than query result bindings. Such queries return values for subject, predicate and object components. See Graph Patterns: Support for SPARQL CONSTRUCT Syntax (page 1-39)for details.

To use the SEM_RELATED operator to query an OWL ontology, see Using Semantic Operators to Query Relational Data (page 2-19).

When you are querying multiple models or querying one or more models and the corresponding entailment, consider using virtual models (explained in Virtual Models (page 1-13)) because of the potential performance benefits.

This section also contains the following topics:

- Performing Queries with Incomplete or Invalid Entailments (page 1-29)
- Graph Patterns: Support for Curly Brace Syntax_ and OPTIONAL_ FILTER_ UNION_ and GRAPH Keywords (page 1-30)
- Graph Patterns: Support for SPARQL SELECT Syntax (page 1-46)
- Graph Patterns: Support for SPARQL 1.1 Constructs (page 1-50)
- Graph Patterns: Support for SPARQL 1.1 Federated Query (page 1-61)
- Inline Query Optimizer Hints (page 1-65)
- Full-Text Search (page 1-65)
- Spatial Support (page 1-66)
- Best Practices for Query Performance (page 1-69)
- Special Considerations When Using SEM_MATCH (page 1-72)

Example 1-9 SEM_MATCH Table Function

Example 1-9 (page 1-28) selects all grandfathers (grandparents who are male) and their grandchildren from the `family` model, using inferencing from both the `RDFS` and `family_rb` rulebases. (This example is an excerpt from Example 1-90 (page 1-96) in Example: Family Information (page 1-95).)

```
SELECT x, y
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male}',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

Example 1-10 HINT0 Option with SEM_MATCH Table Function

Example 1-10 (page 1-28) is functionally the same as Example 1-9 (page 1-28), but it adds the `HINT0` option.

```
SELECT x, y
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male}',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_Aliases(SEM_ALIAS('','http://www.example.org/family/')),
    null,
    null,
    'HINT0={LEADING(t0 t1) USE_NL(?x ?y) GET_CANON_VALUE(?x ?y)}'));
```

Example 1-11 SEM_MATCH Table Function

Example 1-11 (page 1-29) uses the Pathway/Genome `BioPax` ontology to get all chemical compound types that belong to both `Proteins` and `Complexes`:

```
SELECT t.r
  FROM TABLE (SEM_MATCH (
      '{?r rdfs:subClassOf :Proteins .
        ?r rdfs:subClassOf :Complexes}',
      SEM_Models ('BioPax'),
      SEM_Rulebases ('rdfs'),
      SEM_Aliases (SEM_ALIAS('', 'http://www.biopax.org/release1/biopax-
release1.owl')),
      NULL)) t;
```

As shown in Example 1-11 (page 1-29), the search pattern for the SEM_MATCH table function is specified using SPARQL-like syntax where the variable starts with the question-mark character (?). In this example, the variable `?r` must match to the same term, and thus it must be a subclass of both `Proteins` and `Complexes`.

1.6.1 Performing Queries with Incomplete or Invalid Entailments

You can query semantic data even when the relevant entailment does not have a valid status if you specify the string value `INCOMPLETE` or `INVALID` for the `index_status` attribute of the SEM_MATCH table function. (The entailment status is stored in the STATUS column of the MDSYS.SEM_RULES_INDEX_INFO view, which is described in Entailments (Rules Indexes) (page 1-11). The SEM_MATCH table function is described in Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).)

The index_status attribute value affects the query behavior as follows:

- If the entailment has a valid status, the query behavior is not affected by the value of the `index_status` attribute.
- If you provide no value or specify a null value for `index_status`, the query returns an error if the entailment does not have a valid status.
- If you specify the string `INCOMPLETE` for the `index_status` attribute, the query is performed if the status of the entailment is incomplete or valid.
- If you specify the string `INVALID` for the `index_status` attribute, the query is performed regardless of the actual status of the entailment (invalid, incomplete, or valid).

However, the following considerations apply if the status of the entailment is incomplete or invalid:

- If the status is incomplete, the content of an entailment may be approximate, because some triples that are inferable (due to the recent insertions into the underlying models) may not actually be present in the entailment, and therefore results returned by the query may be inaccurate.
- If the status is invalid, the content of the entailment may be approximate, because some triples that are no longer inferable (due to recent modifications to the underlying models or rulebases, or both) may still be present in the entailment, and this may affect the accuracy of the result returned by the query. In addition to possible presence of triples that are no longer inferable, some inferable rows may not actually be present in the entailment.

1.6.2 Graph Patterns: Support for Curly Brace Syntax, and OPTIONAL, FILTER, UNION, and GRAPH Keywords

The SEM_MATCH table function accepts the syntax for the graph pattern in which a sequence of triple patterns is enclosed within curly braces. The period is usually required as a separator unless followed by the OPTIONAL, FILTER, UNION, or GRAPH keyword. With this syntax, you can do any combination of the following:

- Use the OPTIONAL construct to retrieve results even in the case of a partial match
- Use the FILTER construct to specify a filter expression in the graph pattern to restrict the solutions to a query
- Use the UNION construct to match one of multiple alternative graph patterns
- Use the GRAPH construct (explained in GRAPH Keyword Support (page 1-37)) to scope graph pattern matching to a set of named graphs

In addition to arithmetic operators (+, -, *, /), Boolean operators and logical connectives (||, &&, !), and comparison operators (<, >, <=, >=, =, !=), several built-in functions are available for use in FILTER clauses. Table 1-12 (page 1-30) lists built-in functions that you can use in the FILTER clause. In the Description column of Table 1-12 (page 1-30), *x*, *y*, and *z* are arguments of the appropriate types.

Table 1-12 Built-in Functions Available for FILTER Clause

Function	Description
ABS(RDF term)	Returns the absolute value of `term`. If `term` is a non-numerical value, returns null.
BNODE(literal) or BNODE()	Constructs a blank node that is distinct from all blank nodes in the dataset of the query, and those created by this function in other queries. The form with no arguments results in a distinct blank node in every call. The form with a simple literal results in distinct blank nodes for different simple literals, and the same blank node for calls with the same simple literal.
BOUND(variable)	BOUND(x) returns `true` if `x` is bound (that is, non-null) in the result, `false` otherwise.
CEIL(RDF term)	Returns the closest number with no fractional part which is not less than term. If term is a non-numerical value, returns null.
COALESCE(term list)	Returns the first element on the argument list that is evaluated without raising an error. Unbound variables raise an error if evaluated. Returns null if there are no valid elements in the term list.
CONCAT(term list)	Returns an `xsd:String` value resulting of the concatenation of the string values in the term list.
CONTAINS(literal, match)	Returns `true` if the string `match` is found anywhere in `literal`. It returns `false` otherwise.
DATATYPE(literal)	DATATYPE(x) returns a URI representing the datatype of `x`.

Table 1-12 (Cont.) Built-in Functions Available for FILTER Clause

Function	Description
DAY(argument)	Returns an integer corresponding to the day part of argument. If the argument is not a `dateTime` or `date` data type, it returns a null value.
ENCODE_FOR_URI(literal)	Returns a string where the reserved characters in `literal` are escaped and converted to its percent-encode form.
EXISTS(pattern)	Returns `true` if the pattern matches the query data set, using the current bindings in the containing group graph pattern and the current active graph. If there are no matches, it returns `false`.
FLOOR(RDF term)	Returns the closest number with no fractional part which is less than `term`. If `term` is a non-numerical value, returns null.
HOURS(argument)	Returns an integer corresponding to the hours part of `argument`. If the argument is not a `dateTime` or `date` data type, it returns a null value.
IF(condition , expression1, expression2)	Evaluates the condition and obtains the effective Boolean value. If true, the first expression is evaluated and its value returned. If false, the second expression is used. If the condition raises an error, the error is passed as the result of the IF statement.
IRI(RDF term)	Returns an IRI resolving the string representation of argument `term`. If there is a base IRI defined in the query, the IR is resolve against it, and the result must result in an absolute IRI.
isBLANK(RDF term)	isBLANK(x) returns `true` if `x` is a blank node, `false` otherwise.
isIRI(RDF term)	isIRI(x) returns `true` if `x` is an IRI, `false` otherwise.
isLITERAL(RDF term)	isLiteral(x) returns `true` if `x` is a literal, `false` otherwise.
IsNUMERIC(RDF term)	Returns `true` if `term` is a numeric value, `false` otherwise.
isURI(RDF term)	isURI(x) returns `true` if `x` is a URI, `false` otherwise.
LANG(literal)	LANG(x) returns a plain literal serializing the language tag of `x`.
LANGMATCHES(literal, literal)	LANGMATCHES(x, y) returns `true` if language tag `x` matches language range `y`, `false` otherwise.
LCASE(literal)	Returns a string where each character in literal is converted to its lowercase correspondent.
MD5(literal)	Returns the checksum for `literal`, corresponding to the MD5 hash function.
MINUTES(argument)	Returns an integer corresponding to the minutes part of `argument`. If the argument is not a `dateTime` or `date` data type, it returns a null value.

Table 1-12 (Cont.) Built-in Functions Available for FILTER Clause

Function	Description
MONTH(argument)	Returns an integer corresponding to the month part of `argument`. If the argument is not a `dateTime` or `date` data type, it returns a null value.
NOT_EXISTS(pattern)	Returns `true` if the pattern does not match the query data set, using the current bindings in the containing group graph pattern and the current active graph. It returns `false` otherwise.
NOW()	Returns an `xsd:dateTime` value corresponding to the current time at the moment of the query execution.
RAND()	Generates a numeric value in the range of [0,1).
REGEX(string, pattern)	REGEX(x,y) returns `true` if `x` matches the regular expression `y`, `false` otherwise. For more information about the regular expressions supported, see the Oracle Regular Expression Support appendix in *Oracle Database SQL Language Reference*.
REGEX(string, pattern, flags)	REGEX(x,y,z) returns `true` if `x` matches the regular expression `y` using the options given in `z`, `false` otherwise. Available options: `'s'` – dot all mode (`'.'` matches any character including the newline character); `'m'` – multiline mode (`'^'` matches the beginning of any line and `'$'` matches the end of any line); `'i'` – case insensitive mode; `'x'` – remove whitespace characters from the regular expression before matching.
REPLACE(string, pattern, replacement)	Returns a string where each match of the regular expression `pattern` in `string` is replaced by `replacement`. For more information about the regular expressions supported, see the Oracle Regular Expression Support appendix in *Oracle Database SQL Language Reference*.
REPLACE(string, pattern, replacement, flags)	Returns a string where each match of the regular expression `pattern` in string is replaced by `replacement`. Available options: `'s'` – dot all mode (`'.'` matches any character including the newline character); `'m'` – multiline mode (`'^'` matches the beginning of any line and `'$'` matches the end of any line); `'i'` – case insensitive mode; `'x'` – remove whitespace characters from the regular expression before matching. For more information about the regular expressions supported, see the Oracle Regular Expression Support appendix in *Oracle Database SQL Language Reference*.
ROUND(RDF term)	Returns the closest number with no fractional part to `term`. If two values exist, the value closer to positive infinite is returned. If term is a non-numerical value, returns null.
sameTerm(RDF term, RDF term)	sameTerm(x, y) returns `true` if `x` and `y` are the same RDF term, `false` otherwise.
SECONDS(argument)	Returns an integer corresponding to the seconds part of `argument`. If the argument is not a `dateTime` or `date` data type, it returns a null value.

Table 1-12 (Cont.) Built-in Functions Available for FILTER Clause

Function	Description
SHA1(literal)	Returns the checksum for `literal`, corresponding to the SHA1 hash function.
SHA256(literal)	Returns the checksum for `literal`, corresponding to the SHA256 hash function.
SHA384(literal)	Returns the checksum for `literal`, corresponding to the SHA384 hash function.
SHA512(literal)	Returns the checksum for `literal`, corresponding to the SHA512 hash function.
STR(RDF term)	STR(x) returns a plain literal of the string representation of `x` (that is, what would be stored in the VALUE_NAME column of MDSYS.RDF_VALUE$ enclosed within double quotes).
STRAFTER(literal, literal)	StrAfter (x,y) returns the portion of the string corresponding to substring that precedes in `x` the first match of `y`, and the end of `x`. If `y` cannot be matched inside `x`, the empty string is returned.
STRBEFORE(literal, literal)	StrBefore (x,y) returns the portion of the string corresponding to the start of `x` and the first match of `y`. If `y` cannot be matched inside `x`, the empty string is returned.
STRDT(string, datatype)	Construct a literal term composed by the `string` lexical form and the `datatype` passed as arguments. `datatype` must be a URI; otherwise, the function returns a null value.
STRENDS(literal, match)	Returns `true` if the string `literal` ends with the string `match`. It returns `false` otherwise.
STRLANG (string, languageTag)	Constructs a string composed by the `string` lexical form and language tag passed as arguments.
STRLEN(literal)	Returns the length of the lexical form of the `literal`.
STRSTARTS(literal, match)	Returns `true` if the string `literal` starts with the string `match`. It returns `false` otherwise.
STRUUID()	Returns a string containing the scheme section of a new UUID.
SUBSTR(term, startPos)	Returns the string corresponding to the portion of `term` that starts at `startPos` and continues until term ends. The index of the first character is 1.
SUBSTR(term, startPos, length)	Returns the string corresponding to the portion of term that starts at `startPos` and continues for `length` characters. The index of the first character is 1.
term IN (term list)	The expression x IN(term list) returns `true` if `x` can be found in any of the values in `termlist`. Returns `false` if not found. Zero-length lists are legal. An error is raised if any of the values in `termlist` raises an error and `x` is not found.

Table 1-12 (Cont.) Built-in Functions Available for FILTER Clause

Function	Description
term NOT IN (term list)	The expression x NOT IN(term list) returns `false` if `x` can be found in any of the values in term list. Returns `true` if not found. Zero-length lists are legal. An error is raised if any of the values in term list raises an error and `x` is not found.
TIMEZONE(argument)	Returns the time zones section of `argument` as an `xsd:dayTimeDuration` value. If the argument is not a `dateTime` or `date` data type, it returns a null value.
TZ(argument)	Returns an integer corresponding to the time zone part of `argument`. If the argument is not a `dateTime` or `date` data type, it returns a null value.
UCASE(literal)	Returns a string where each character in `literal` is converted to its uppercase correspondent.
URI(RDF term)	(Synonym for IRI(RDF term)
UUID()	Returns a URI with a new Universal Unique Identifier. The value and the version correspond to the PL/SQL function `sys_guid ()`.
YEAR(argument)	Returns an integer corresponding to the year part of `argument`.

See also the descriptions of the built-in functions defined in the SPARQL query language specification (http://www.w3.org/TR/sparql11-query/), to better understand the built-in functions available in SEM_MATCH.

The following XML Schema casting functions are available for use in FILTER clauses. These functions take an RDF term as input and return a new RDF term of the desired type or raise an error if the term cannot be cast to the desired type. Details of type casting can be found in Section 17.1 of the XPath query specification: http://www.w3.org/TR/xpath-functions/#casting-from-primitive-to-primitive. These functions use the XML Namespace `xsd : http://www.w3.org/2001/XMLSchema#`.

- xsd:string (RDF term)
- xsd:dateTime (RDF term)
- xsd:boolean (RDF term)
- xsd:integer (RDF term)
- xsd:float (RDF term)
- xsd:double (RDF term)
- xsd:decimal (RDF term)

If you use the syntax with curly braces to express a graph pattern:

- The query always returns canonical lexical forms for the matching values for the variables.

- Any hints specified in the `options` argument using HINT0={*<hint-string>*} (explained in Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22)), should be constructed only on the basis of the portion of the graph pattern inside the root BGP. For example, the only valid aliases for use in a hint specification for the query in Example 1-13 (page 1-35) are `t0`, `t1`, `?x`, and `?y`. Inline query optimizer hints can be used to influence other parts of the graph pattern (see Inline Query Optimizer Hints (page 1-65)).
- The FILTER construct is not supported for variables bound to long literals.

Example 1-12 Curly Brace Syntax

Example 1-12 (page 1-35) uses the syntax with curly braces and a period to express a graph pattern in the SEM_MATCH table function.

```
SELECT x, y
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male}',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

Example 1-13 Curly Brace Syntax and OPTIONAL Construct

Example 1-13 (page 1-35) uses the OPTIONAL construct to modify Example 1-12 (page 1-35), so that it also returns, for each grandfather, the names of the games that he plays or null if he does not play any games.

```
SELECT x, y, game
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male .
      OPTIONAL{?x :plays ?game}
     }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null,
    null,
    'HINT0={LEADING(t0 t1) USE_NL(?x ?y)}'));
```

Example 1-14 Curly Brace Syntax and Multi-Pattern OPTIONAL Construct

When multiple triple patterns are present in an OPTIONAL graph pattern, values for optional variables are returned only if a match is found for each triple pattern in the OPTIONAL graph pattern. Example 1-14 (page 1-35) modifies Example 1-13 (page 1-35) so that it returns, for each grandfather, the names of the games both he and his grandchildren play, or null if he and his grandchildren have no such games in common. It also uses global query optimizer hints to specify that triple patterns should be evaluated in order within each BGP and that a hash join should be used to join the root BGP with the OPTIONAL BGP.

```
SELECT x, y, game
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male .
      OPTIONAL{?x :plays ?game . ?y :plays ?game}
     }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
```

```
    null,
    'ALL_ORDERED ALL_BGP_HASH'));
```

Example 1-15 Curly Brace Syntax and Nested OPTIONAL Construct

A single query can contain multiple OPTIONAL graph patterns, which can be nested or parallel. Example 1-15 (page 1-36) modifies Example 1-14 (page 1-35) with a nested OPTIONAL graph pattern. This example returns, for each grandfather, (1) the games he plays or null if he plays no games and (2) if he plays games, the ages of his grandchildren that play the same game, or null if he has no games in common with his grandchildren. Note that in Example 1-15 (page 1-36) a value is returned for `?game` even if the nested OPTIONAL graph pattern `?y :plays ?game . ?y :age ?age` is not matched.

```
SELECT x, y, game, age
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male .
      OPTIONAL{?x :plays ?game
                          OPTIONAL {?y :plays ?game . ?y :age ?age} }
     }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

Example 1-16 Curly Brace Syntax and Parallel OPTIONAL Construct

Example 1-16 (page 1-36) modifies Example 1-14 (page 1-35) with a parallel OPTIONAL graph pattern. This example returns, for each grandfather, (1) the games he plays or null if he plays no games and (2) his email address or null if he has no email address. Note that, unlike nested OPTIONAL graph patterns, parallel OPTIONAL graph patterns are treated independently. That is, if an email address is found, it will be returned regardless of whether or not a game was found; and if a game was found, it will be returned regardless of whether an email address was found.

```
SELECT x, y, game, email
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male .
      OPTIONAL{?x :plays ?game}
      OPTIONAL{?x :email ?email}
     }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

Example 1-17 Curly Brace Syntax and FILTER Construct

Example 1-17 (page 1-36) uses the FILTER construct to modify Example 1-12 (page 1-35), so that it returns grandchildren information for only those grandfathers who are residents of either NY or CA.

```
SELECT x, y
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male . ?x :residentOf ?z
       FILTER (?z = "NY"  || ?z = "CA")}',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

Example 1-18 Curly Brace Syntax and FILTER with REGEX and STR Built-In Constructs

Example 1-18 (page 1-37) uses the REGEX built-in function to select all grandfathers who have an Oracle email address. Note that backslash (\) characters in the regular expression pattern must be escaped in the query string; for example, `\\.` produces the following pattern: `\.`

```
SELECT x, y, z
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male . ?x :email ?z
       FILTER (REGEX(STR(?z), "@oracle\\.com$"))}',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

Example 1-19 Curly Brace Syntax and UNION and FILTER Constructs

Example 1-19 (page 1-37) uses the UNION construct to modify Example 1-17 (page 1-36), so that grandfathers are returned only if they are residents of NY or CA or own property in NY or CA, or if both conditions are true (they reside in and own property in NY or CA).

```
SELECT x, y
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male
       {{?x :residentOf ?z} UNION {?x :ownsPropertyIn ?z}}
       FILTER (?z = "NY"  || ?z = "CA")}',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

1.6.2.1 GRAPH Keyword Support

A SEM_MATCH query is executed against an RDF Dataset. An RDF Dataset is a collection of graphs that includes one unnamed graph, known as the default graph, and one or more named graphs, which are identified by a URI. Graph patterns that appear inside a GRAPH clause are matched against the set of named graphs, and graph patterns that do not appear inside a graph clause are matched against the default graph. The `graphs` and `named_graphs` SEM_MATCH parameters are used to construct the default graph and set of named graphs for a given SEM_MATCH query. A summary of possible dataset configurations is shown in Table 1-13 (page 1-37).

Table 1-13 SEM_MATCH graphs and named_graphs Values, and Resulting Dataset Configurations

Parameter Values	Default Graph	Set of Named Graphs
`graphs`: NULL `named_graphs`: NULL	Union All of all unnamed triples and all named graph triples. (But if the `options` parameter contains `STRICT_DEFAULT=T`, only unnamed triples are included in the default graph.)	All named graphs
`graphs`: NULL `named_graphs`: {g1,…, gn}	Empty set	{g1,…, gn}
`graphs`: {g1,…, gm} `named_graphs`: NULL	Union All of {g1,…, gm}	Empty set

Table 1-13 (Cont.) SEM_MATCH graphs and named_graphs Values, and Resulting Dataset Configurations

Parameter Values	Default Graph	Set of Named Graphs
graphs: {g1,…, gm} named_graphs: {gn,…, gz}	Union All of {g1,…, gm}	{gn,…, gz}

See also the W3C SPARQL specification for more information on RDF data sets and the GRAPH construct, specifically: http://www.w3.org/TR/rdf-sparql-query/#rdfDataset

Example 1-20 Named Graph Construct

Example 1-20 (page 1-38) uses the GRAPH construct to scope graph pattern matching to a specific named graph. This example finds the names and email addresses of all people in the <http://www.example.org/family/Smith> named graph.

```
SELECT name, email
  FROM TABLE(SEM_MATCH(
    '{GRAPH :Smith {
       ?x :name ?name . ?x :email ?email } }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

Example 1-21 Using the named_graphs Parameter

In addition to URIs, variables can appear after the GRAPH keyword. Example 1-21 (page 1-38) uses a variable, ?g, with the GRAPH keyword, and uses the named_graphs parameter to restrict the possible values of ?g to the <http://www.example.org/family/Smith> and <http://www.example.org/family/Jones> named graphs. Aliases specified in SEM_ALIASES argument can be used in the graphs and named_graphs parameters.

```
SELECT name, email
  FROM TABLE(SEM_MATCH(
    '{GRAPH ?g {
       ?x :name ?name . ?x :email ?email } }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null,null,null,null,
    SEM_GRAPHS('<http://www.example.org/family/Smith>',
               ':Jones')));
```

Example 1-22 Using the graphs Parameter

Example 1-22 (page 1-38) uses the default graph to query the union of the <http://www.example.org/family/Smith> and <http://www.example.org/family/Jones> named graphs.

```
FROM TABLE(SEM_MATCH(
    '{?x :name ?name . ?x :email ?email }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null,null,null,
    SEM_GRAPHS('<http://www.example.org/family/Smith>',
```

```
                ':Jones'),
    null));
```

1.6.3 Graph Patterns: Support for SPARQL ASK Syntax

SEM_MATCH allows fully-specified SPARQL ASK queries in the query parameter.

ASK queries are used to test whether or not a solution exists for a given query pattern. In contrast to other forms of SPARQL queries, ASK queries do not return any information about solutions to the query pattern. Instead, such queries return `"true"^^xsd:boolean` if a solution exists and `"false"^^xsd:boolean` if no solution exists.

All SPARQL ASK queries return the same columns: ASK, ASK$RDFVID, ASK $_PREFIX, ASK$_SUFFIX, ASK$RDFVTYP, ASK$RDFCLOB, ASK$RDFLTYP, ASK $RDFLANG, SEM$ROWNUM. Note that these columns are the same as a SPARQL SELECT syntax query that projects a single `?ask` variable.

SPARQL ASK queries will generally give better performance than an equivalent SPARQL SELECT syntax query because the ASK query does not have to retrieve lexical values for query variables, and query execution can stop after a single result has been found.

SPARQL ASK queries use the same syntax as SPARQL SELECT queries, but the topmost SELECT clause must be replaced with the keyword ASK.

Example 1-23 SPARQL ASK

Example 1-23 (page 1-39) shows a SPARQL ASK query that determines whether or not any cameras are for sale with more than 10 megapixels that cost less than 50 dollars.

```
SELECT ask
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     ASK
     WHERE
      {?x :price ?p .
       ?x :megapixels ?m .
       FILTER (?p < 50 && ?m > 10)
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

See also the W3C SPARQL specification for more information on SPARQL ASK queries, specifically: `http://www.w3.org/TR/sparql11-query/#ask`

1.6.4 Graph Patterns: Support for SPARQL CONSTRUCT Syntax

SEM_MATCH allows fully-specified SPARQL CONSTRUCT queries in the query parameter.

CONSTRUCT queries are used to build RDF graphs from stored RDF data. In contrast to SPARQL SELECT queries, CONSTRUCT queries return a set of RDF triples rather than a set of query solutions (variable bindings).

All SPARQL CONSTRUCT queries return the same columns from SEM_MATCH. These columns correspond to the subject, predicate and object of an RDF triple, and there are 10 columns for each triple component. In addition, a SEM$ROWNUM column is returned. More specifically, the following columns are returned:

```
SUBJ
SUBJ$RDFVID
SUBJ$_PREFIX
SUBJ$_SUFFIX
SUBJ$RDFVTYP
SUBJ$RDFCLOB
SUBJ$RDFLTYP
SUBJ$RDFLANG
SUBJ$RDFTERM
SUBJ$RDFCLBT
PRED
PRED$RDFVID
PRED$_PREFIX
PRED$_SUFFIX
PRED$RDFVTYP
PRED$RDFCLOB
PRED$RDFLTYP
PRED$RDFLANG
PRED$RDFTERM
PRED$RDFCLBT
OBJ
OBJ$RDFVID
OBJ$_PREFIX
OBJ$_SUFFIX
OBJ$RDFVTYP
OBJ$RDFCLOB
OBJ$RDFLTYP
OBJ$RDFLANG
OBJ$RDFTERM
OBJ$RDFCLBT
SEM$ROWNUM
```

For each component, COMP, COMP$RDFVID, COMP$_PREFIX, COMP$_SUFFIX, COMP$RDFVTYP, COMP$RDFCLOB, COMP$RDFLTYP, and COMP$RDFLANG correspond to the same values as those from SPARQL SELECT queries. COMP $RDFTERM holds a VARCHAR2(4000) RDF term in N-Triple syntax, and COMP $RDFCLBT holds a CLOB RDF term in N-Triple syntax.

SPARQL CONSTRUCT queries use the same syntax as SPARQL SELECT queries, except the topmost SELECT clause is replaced with a CONSTRUCT template. The CONSTRUCT template determines how to construct the result RDF graph using the results of the query pattern defined in the WHERE clause. A CONSTRUCT template consists of the keyword CONSTRUCT followed by sequence of SPARQL triple patterns that are enclosed within curly braces. The keywords OPTIONAL, UNION, FILTER, MINUS, BIND, VALUES, and GRAPH are not allowed within CONSTRUCT templates, and property path expressions are not allowed within CONSTRUCT templates. These keywords, however, are allowed within the query pattern inside the WHERE clause.

SPARQL CONSTRUCT queries build result RDF graphs in the following manner. For each result row returned by the WHERE clause, variable values are substituted into the CONSTRUCT template to create one or more RDF triples. Suppose the graph pattern in the WHERE clause of Example 1-24 (page 1-41) returns the following result rows.

E$RDFTERM	FNAME$RDFTERM	LNAME$RDFTERM
ent:employee1	"Fred"	"Smith"

E$RDFTERM	FNAME$RDFTERM	LNAME$RDFTERM
ent:employee2	"Jane"	"Brown"
ent:employee3	"Bill"	"Jones"

The overall SEM_MATCH CONSTRUCT query in Example 1-24 (page 1-41) would then return the following rows, which correspond to six RDF triples (two for each result row of the query pattern).

SUBJ$RDFTERM	PRED$RDFTERM	OBJ$RDFTERM
ent:employee1	foaf:givenName	"Fred"
ent:employee1	foaf:familyName	"Smith"
ent:employee2	foaf:givenName	"Jane"
ent:employee2	foaf:familyName	"Brown"
ent:employee3	foaf:givenName	"Bill"
ent:employee3	foaf:familyName	"Jones"

There are two SEM_MATCH query options that influence the behavior of SPARQL CONSTRUCT: `CONSTRUCT_UNIQUE=T` and `CONSTRUCT_STRICT=T`. Using the `CONSTRUCT_UNIQUE=T` query option ensures that only unique RDF triples are returned from the CONSTRUCT query. Using the `CONSTRUCT_STRICT=T` query option ensures that only valid RDF triples are returned from the CONSTRUCT query. Valid RDF triples are those that have (1) a URI or blank node in the subject position, (2) a URI in the predicate position, and (3) a URI, blank node or RDF literal in the object position. Both of these query options are turned off by default for improved query performance.

Example 1-24 SPARQL CONSTRUCT

Example 1-24 (page 1-41) shows a SPARQL CONSTRUCT query that builds an RDF graph of employee names using the foaf vocabulary.

```
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
  FROM TABLE(SEM_MATCH(
    'PREFIX  ent: <http://www.example.org/enterprise/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     CONSTRUCT
      {?e foaf:givenName  ?fname .
       ?e foaf:familyName ?lname
      }
     WHERE
      {?e ent:fname ?fname .
       ?e ent:lname ?lname
      }',
    SEM_Models('enterprise'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-25 CONSTRUCT with Solution Modifiers

SPARQL SOLUTION modifiers can be used with CONSTRUCT queries. Example 1-25 (page 1-42) shows the use of ORDER BY and LIMIT to build a graph about the top two highest-paid employees. Note that the LIMIT 2 clause applies to the query pattern not to the overall CONSTRUCT query. That is, the query pattern will return two result rows, but the overall CONSTRUCT query will return 6 RDF triples (three for each of the two employees bound to `?e`).

```
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
  FROM TABLE(SEM_MATCH(
    'PREFIX  ent: <http://www.example.org/enterprise/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     CONSTRUCT
      { ?e ent:fname       ?fname .
        ?e ent:lname       ?lname .
        ?e ent:dateOfBirth ?dob }
     WHERE
      { ?e ent:fname  ?fname .
        ?e ent:lname  ?lname .
        ?e ent:salary ?sal
      }
     ORDER BY DESC(?sal)
     LIMIT 2',
    SEM_Models('enterprise'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-26 SPARQL 1.1 Features with CONSTRUCT

SPARQL 1.1 features are supported within CONSTRUCT query patterns. Example 1-26 (page 1-42) shows the use of subqueries and SELECT expressions within a CONSTRUCT query.

```
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
  FROM TABLE(SEM_MATCH(
    'PREFIX  ent: <http://www.example.org/enterprise/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     CONSTRUCT
      { ?e foaf:name  ?name }
     WHERE
      { SELECT ?e (CONCAT(?fname," ",?lname) AS ?name)
        WHERE { ?e ent:fname ?fname .
                ?e ent:lname ?lname }
      }',
    SEM_Models('enterprise'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-27 SPARQL CONSTRUCT with Named Graphs

Named graph data cannot be returned from SPARQL CONSTRUCT queries because, in accordance with the W3C SPARQL specification, only RDF triples are returned, not RDF quads. The FROM, FROM NAMED and GRAPH keywords, however, can be used when matching the query pattern defined in the WHERE clause.

Example 1-27 (page 1-42) constructs an RDF graph with `ent:name` triples from the UNION of named graphs `ent:g1` and `ent:g2`, `ent:dateOfBirth` triples from named graph `ent:g3`, and `ent:ssn` triples from named `graph ent:g4`.

```
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
  FROM TABLE(SEM_MATCH(
```

```
    'PREFIX  ent: <http://www.example.org/enterprise/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     CONSTRUCT
      { ?e ent:name ?name .
        ?e ent:dateOfBirth ?dob .
        ?e ent:ssn ?ssn
      }
     FROM ent:g1
     FROM ent:g2
     FROM NAMED ent:g3
     FROM NAMED ent:g4
     WHERE
      { ?e foaf:name ?name .
        GRAPH ent:g3 { ?e ent:dateOfBirth ?dob }
        GRAPH ent:g4 { ?e ent:ssn ?ssn }
      }',
    SEM_Models('enterprise'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-28 SPARQL CONSTRUCT Normal Form

```
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
  FROM TABLE(SEM_MATCH(
    'PREFIX  ent: <http://www.example.org/enterprise/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     CONSTRUCT
      {?e foaf:givenName  ?fname .
       ?e foaf:familyName ?lname
      }
     WHERE
      {?e ent:fname ?fname .
       ?e ent:lname ?lname
      }',
    SEM_Models('enterprise'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-29 SPARQL CONSTRUCT Short Form

A short form of CONSTRUCT is supported when the CONSTRUCT template is exactly the same as the WHERE clause. In this case, only the keyword CONSTRUCT is needed, and the graph pattern in the WHERE clause will also be used as a CONSTRUCT template. Example 1-29 (page 1-43) shows the short form of Example 1-28 (page 1-43).

```
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
  FROM TABLE(SEM_MATCH(
    'PREFIX  ent: <http://www.example.org/enterprise/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     CONSTRUCT
     WHERE
      {?e ent:fname ?fname .
       ?e ent:lname ?lname
      }',
    SEM_Models('enterprise'),
    SEM_Rulebases('RDFS'),
    null, null));
```

1.6.4.1 Typical SPARQL CONSTRUCT Workflow

A typical workflow for SPARQL CONSTRUCT would be to execute a CONSTRUCT query to extract and/or transform RDF triple data from an existing semantic model and then load this data into an existing or new semantic model. The data loading can be accomplished through simple INSERT statements or executing the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE (page 11-11) procedure.

Example 1-30 SPARQL CONSTRUCT Workflow

Example 1-30 (page 1-44) constructs `foaf:name` triples from existing `ent:fname` and `ent:lname` triples and then bulk loads these new triples back into the original model. Afterward, you can query the original model for `foaf:name` values.

```
-- use create table as select to build a staging table
CREATE TABLE STAB(RDF$STC_sub, RDF$STC_pred, RDF$STC_obj) AS
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
FROM TABLE(SEM_MATCH(
 'PREFIX  ent: <http://www.example.org/enterprise/>
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  CONSTRUCT
   { ?e foaf:name  ?name }
  WHERE
   { SELECT ?e (CONCAT(?fname," ",?lname) AS ?name)
     WHERE { ?e ent:fname ?fname .
             ?e ent:lname ?lname }
   }',
 SEM_Models('enterprise'),
 null, null, null));

-- grant privileges on STAB
GRANT SELECT ON STAB TO MDSYS;

-- bulk load data back into the enterprise model
BEGIN
  SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE(
    model_name=>'enterprise',
    table_owner=>'rdfuser',
    table_name=>'stab',
    flags=>' parallel_create_index parallel=4 ');
END;
/

-- query for foaf:name data
SELECT e$rdfterm, name$rdfterm
FROM TABLE(SEM_MATCH(
 'PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  SELECT ?e ?name
  WHERE { ?e foaf:name ?name }',
 SEM_Models('enterprise'),
null, null, null));
```

See also the W3C SPARQL specification for more information on SPARQL CONSTRUCT queries, specifically: `http://www.w3.org/TR/sparql11-query/#construct`

1.6.5 Graph Patterns: Support for SPARQL DESCRIBE Syntax

SEM_MATCH allows fully-specified SPARQL DESCRIBE queries in the query parameter.

SPARQL DESCRIBE queries are useful for exploring RDF data sets. You can easily find information about a given resource or set of resources without knowing information about the exact RDF properties used in the data set. A DESCRIBE query returns a "description" of a resource `r`, where a "description" is the set of RDF triples in the query data set that contain `r` in either the subject or object position.

Like CONSTRUCT queries, DESCRIBE queries return an RDF graph instead of result bindings. Each DESCRIBE query, therefore, returns the same columns as a CONSTRUCT query (see Graph Patterns: Support for SPARQL CONSTRUCT Syntax (page 1-39) for a listing of return columns).

SPARQL DESCRIBE queries use the same syntax as SPARQL SELECT queries, except the topmost SELECT clause is replaced with a DESCRIBE clause. A DESCRIBE clause consists of the DESCRIBE keyword followed by a sequence of URIs and/or variables separated by whitespace or the DESCRIBE keyword followed by a single * (asterisk).

Two SEM_MATCH query options affect SPARQL DESCRIBE queries: `CONSTRUCT_UNIQUE=T` and `CONSTRUCT_STRICT=T`. `CONSTRUCT_UNIQUE=T` ensures that duplicate triples are eliminated from the result, and `CONSTRUCT_STRICT=T` ensures that invalid triples are eliminated from the result. Both of these options are turned off by default. These options are described in more detail in Graph Patterns: Support for SPARQL CONSTRUCT Syntax (page 1-39).

See also the W3C SPARQL specification for more information on SPARQL DESCRIBE queries, specifically: `http://www.w3.org/TR/sparql11-query/#describe`

Example 1-31 SPARQL DESCRIBE Short Form

A short form of SPARQL DESCRIBE is provided to describe a single constant URI. In the short form, only a DESCRIBE clause is needed. Example 1-31 (page 1-45) shows a short form SPARQL DESCRIBE query.

```
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
FROM TABLE(SEM_MATCH(
 'DESCRIBE <http://www.example.org/enterprise/emp_1>',
 SEM_Models('enterprise'),
null, null, null));
```

Example 1-32 SPARQL DESCRIBE Normal Form

The normal form of SPARQL DESCRIBE specifies a DESCRIBE clause and a SPARQL query pattern, possibly including solution modifiers. Example 1-32 (page 1-45) shows a SPARQL DESCRIBE query that describes all employees whose departments are located in New Hampshire.

```
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
FROM TABLE(SEM_MATCH(
 'PREFIX  ent: <http://www.example.org/enterprise/>
  DESCRIBE ?e
  WHERE
   { ?e ent:department ?dept .
     ?dept ent:locatedIn "New Hampshire" }',
 SEM_Models('enterprise'),
null, null, null));
```

Example 1-33 DESCRIBE *

With the normal form of DESCRIBE, as shown in Example 1-32 (page 1-45), all resources bound to variables listed in the DESCRIBE clause are described. In Example 1-32 (page 1-45), all employees returned from the query pattern and bound

to ?e will be described. When DESCRIBE * is used, all visible variables in the query are described.

Example 1-33 (page 1-45) shows a modified version of Example 1-32 (page 1-45) that describes both employees (bound to ?e) and departments (bound to ?dept).

```
SELECT subj$rdfterm, pred$rdfterm, obj$rdfterm
FROM TABLE(SEM_MATCH(
 'PREFIX  ent: <http://www.example.org/enterprise/>
  DESCRIBE *
  WHERE
   { ?e ent:department ?dept .
     ?dept ent:locatedIn "New Hampshire" }',
 SEM_Models('enterprise'),
null, null, null));
```

1.6.6 Graph Patterns: Support for SPARQL SELECT Syntax

In addition to curly-brace graph patterns, SEM_MATCH allows fully-specified SPARQL SELECT queries in the query parameter. When using the SPARQL SELECT syntax option, SEM_MATCH supports the following query constructs: BASE, PREFIX, SELECT, SELECT DISTINCT, FROM, FROM NAMED, WHERE, ORDER BY, LIMIT, and OFFSET. Each SPARQL SELECT syntax query must include a SELECT clause and a graph pattern.

A key difference between curly-brace and SPARQL SELECT syntax when using SEM_MATCH is that only variables appearing in the SPARQL SELECT clause are returned from SEM_MATCH when using SPARQL SELECT syntax.

One additional column, SEM$ROWNUM, is returned from SEM_MATCH when using SPARQL SELECT syntax. This NUMBER column can be used to order the results of a SEM_MATCH query so that the result order matches the ordering specified by a SPARQL ORDER BY clause.

The SPARQL ORDER BY clause can be used to order the results of SEM_MATCH queries. This clause specifies a sequence of comparators used to order the results of a given query. A comparator consists of an expression composed of variables, RDF terms, arithmetic operators (+, -, *, /), Boolean operators and logical connectives (||, &&, !), comparison operators (<, >, <=, >=, =, !=), and any functions available for use in FILTER expressions.

The following order of operations is used when evaluating SPARQL SELECT queries:

1. Graph pattern matching
2. Grouping (see Grouping and Aggregation (page 1-52).)
3. Aggregates (see Grouping and Aggregation (page 1-52))
4. Having (see Grouping and Aggregation (page 1-52))
5. Values (see Value Assignment (page 1-56))
6. Select expressions
7. Order by
8. Projection
9. Distinct

10. Offset

11. Limit

See also the W3C SPARQL specification for more information on SPARQL BASE, PREFIX, SELECT, SELECT DISTINCT, FROM, FROM NAMED, WHERE, ORDER BY, LIMIT, and OFFSET constructs, specifically: `http://www.w3.org/TR/sparql11-query/`

Example 1-34 SPARQL PREFIX, SELECT, and WHERE Clauses

Example 1-34 (page 1-47) uses the following SPARQL constructs:

- SPARQL PREFIX clause to specify an abbreviation for the `http://www.example.org/family/` and `http://xmlns.com/foaf/0.1/` namespaces
- SPARQL SELECT clause to specify the set of variables to project out of the query
- SPARQL WHERE clause to specify the query graph pattern

```
SELECT y, name
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/family/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     SELECT ?y ?name
     WHERE
     {?x :grandParentOf ?y .
      ?x foaf:name ?name }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    null, null));
```

Example 1-34 (page 1-47) returns the following columns: y, y$RDFVID, y$_PREFIX, y $_SUFFIX, y$RDFVTYP, y$RDFCLOB, y$RDFLTYP, y$RDFLANG, name, name $RDFVID, name$_PREFIX, name$_SUFFIX, name$RDFVTYP, name$RDFCLOB, name $RDFLTYP, name$RDFLANG, and SEM$ROWNUM.

Example 1-35 SPARQL SELECT * (All Variables in Triple Pattern)

The SPARQL SELECT clause specifies either (A) a sequence of variables and/or expressions (see Expressions in the SELECT Clause (page 1-50)), or (B) * (asterisk), which projects all variables that appear in a specified triple pattern. Example 1-35 (page 1-47) uses the SPARQL SELECT clause to select all variables that appear in a specified triple pattern.

```
SELECT x, y, name
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/family/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     SELECT *
     WHERE
     {?x :grandParentOf ?y .
      ?x foaf:name ?name }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    null, null));
```

Example 1-36 SPARQL SELECT DISTINCT

The DISTINCT keyword can be used after SELECT to remove duplicate result rows. Example 1-36 (page 1-47) uses SELECT DISTINCT to select only the distinct names.

```
SELECT name
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/family/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     SELECT DISTINCT ?name
     WHERE
     {?x :grandParentOf ?y .
      ?x foaf:name ?name }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    null, null));
```

Example 1-37 RDF Dataset Specification Using FROM and FROM NAMED

SPARQL FROM and FROM NAMED are used to specify the RDF dataset for a query. FROM clauses are used to specify the set of graphs that make up the default graph, and FROM NAMED clauses are used to specify the set of graphs that make up the set of named graphs. Example 1-37 (page 1-48) uses FROM and FROM NAMED to select email addresses and friend of relationships from the union of the `<http://www.friends.com/friends>` and `<http://www.contacts.com/contacts>` graphs and grandparent information from the `<http://www.example.org/family/Smith>` and `<http://www.example.org/family/Jones>` graphs.

```
SELECT x, y, z, email
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/family/>
     PREFIX foaf: <http://xmlns.com/foaf/0.1/>
     PREFIX friends: <http://www.friends.com/>
     PREFIX contacts: <http://www.contacts.com/>
     SELECT *
     FROM friends:friends
     FROM contacts:contacts
     FROM NAMED :Smith
     FROM NAMED :Jones
     WHERE
     {?x foaf:frendOf ?y .
      ?x :email ?email .
      GRAPH ?g {
        ?x :grandParentOf ?z }
     }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    null, null));
```

Example 1-38 SPARQL ORDER BY

In a SPARQL ORDER BY clause:

- Single variable ordering conditions do not require enclosing parenthesis, but parentheses are required for more complex ordering conditions.
- An optional ASC() or DESC() order modifier can be used to indicate the desired order (ascending or descending, respectively). Ascending is the default order.
- When using SPARQL ORDER BY in SEM_MATCH, the containing SQL query should be ordered by SEM$ROWNUM to ensure that the desired ordering is maintained through any enclosing SQL blocks.

Example 1-38 (page 1-48) uses a SPARQL ORDER BY clause to select all cameras, and it specifies ordering by descending type and ascending total price (`price * (1 - discount) * (1 + tax)`).

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT *
     WHERE
      {?x :price ?p .
       ?x :discount ?d .
       ?x :tax ?t .
       ?x :cameraType ?cType .
      }
     ORDER BY DESC(?cType) ASC(?p * (1-?d) * (1+?t))',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null))
ORDER BY SEM$ROWNUM;
```

Example 1-39 SPARQL LIMIT

SPARQL LIMIT and SPARQL OFFSET can be used to select different subsets of the query solutions. Example 1-39 (page 1-49) uses SPARQL LIMIT to select the five cheapest cameras, and Example 1-40 (page 1-49) uses SPARQL LIMIT and OFFSET to select the fifth through tenth cheapest cameras.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?p
     WHERE
      {?x :price ?p .
       ?x :cameraType ?cType .
      }
     ORDER BY ASC(?p)
     LIMIT 5',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null))
ORDER BY SEM$ROWNUM;
```

Example 1-40 SPARQL OFFSET

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?p
     WHERE
      {?x :price ?p .
       ?x :cameraType ?cType .
      }
     ORDER BY ASC(?p)
     LIMIT 5
     OFFSET 5',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null))
ORDER BY SEM$ROWNUM;
```

Example 1-41 Query Using Full URIs

The SPARQL BASE keyword is used to set a global prefix. All relative IRIs will be resolved with the BASE IRI using the basic algorithm described in Section 5.2 of the *Uniform Resource Identifier (URI): Generic Syntax (RFC3986)* (`http://`

www.ietf.org/rfc/rfc3986.txt). Example 1-41 (page 1-49) is a simple query using full URIs, and Example 1-42 (page 1-50) is an equivalent query using a base IRI.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'SELECT ?employee ?position
     WHERE
      {?x <http://www.example.org/employee> ?p .
       ?p <http://www.example.org/employee/name> ?employee .
       ?p <http://www.example.org/employee/position> ?pos .
       ?pos <http://www.example.org/positions/name> ?position
      }',
    SEM_Models('enterprise'),
    null,
    null, null))
ORDER BY 1,2;
```

Example 1-42 Query Using a Base IRI

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'BASE <http://www.example.org/>
     SELECT ?employee ?position
     WHERE
      {?x <employee> ?p .
       ?p <employee/name> ?employee .
       ?p <employee/position> ?pos .
       ?pos <positions/name> ?position
      }',
    SEM_Models('enterprise'),
    null,
    null, null))
ORDER BY 1,2;
```

1.6.7 Graph Patterns: Support for SPARQL 1.1 Constructs

SEM_MATCH supports the following SPARQL 1.1 constructs:

- An expanded set of functions (all items in Table 1-12 (page 1-30) in Graph Patterns: Support for Curly Brace Syntax_ and OPTIONAL_ FILTER_ UNION_ and GRAPH Keywords (page 1-30))
- Expressions in the SELECT Clause (page 1-50)
- Subqueries (page 1-51)
- Grouping and Aggregation (page 1-52)
- Negation (page 1-55)
- Value Assignment (page 1-56)
- Property Paths (page 1-59)

1.6.7.1 Expressions in the SELECT Clause

Expressions can be used in the SELECT clause to project the value of an expression from a query. A SELECT expression is composed of variables, RDF terms, arithmetic operators (+, -, *, /), Boolean operators and logical connectives (||, &&, !), comparison operators (<, >, <=, >=, =, !=), and any functions available for use in FILTER expressions. The expression must be aliased to a single variable using the AS keyword,

and the overall *<expression> AS <alias>* fragment must be enclosed in parentheses. The alias variable cannot already be defined in the query. A SELECT expression may reference the result of a previous SELECT expression (that is, an expression that appears earlier in the SELECT clause).

Example 1-43 SPARQL SELECT Expression

Example 1-43 (page 1-51) uses a SELECT expression to project the total price for each camera.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ((?p * (1-?d) * (1+?t)) AS ?totalPrice)
     WHERE
      {?x :price ?p .
       ?x :discount ?d .
       ?x :tax ?t .
       ?x :cameraType ?cType .
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-44 SPARQL SELECT Expressions (2)

Example 1-44 (page 1-51) uses two SELECT expressions to project the discount price with and without sales tax.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ((?p * (1-?d)) AS ?preTaxPrice) ((?preTaxPrice * (1+?t)) AS ?
finalPrice)
     WHERE
      {?x :price ?p .
       ?x :discount ?d .
       ?x :tax ?t .
       ?x :cameraType ?cType .
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

1.6.7.2 Subqueries

Subqueries are allowed with SPARQL SELECT syntax. That is, fully-specified SPARQL SELECT queries may be embedded within other SPARQL SELECT queries. Subqueries have many uses, for example, limiting the number of results from a subcomponent of a query.

Example 1-45 SPARQL SELECT Subquery

Example 1-45 (page 1-51) uses a subquery to find the manufacturer that makes the cheapest camera and then finds all other cameras made by this manufacturer.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?c1
     WHERE {?c1 rdf:type :Camera .
            ?c1 :manufacturer ?m .
            {
```

```
          SELECT ?m
          WHERE {?c2 rdf:Type :Camera .
                 ?c2 :price ?p .
                 ?c2 :manufacturer ?m .
          }
          ORDER BY ASC(?p)
          LIMIT 1
         }
 }',
SEM_Models('electronics'),
SEM_Rulebases('RDFS'),
null, null));
```

Subqueries are logically evaluated first, and the results are projected up to the outer query. Note that only variables projected in the subquery's SELECT clause are visible to the outer query.

1.6.7.3 Grouping and Aggregation

The GROUP BY keyword used to perform grouping. Syntactically, the GROUP BY keyword must appear after the WHERE clause and before any solution modifiers such as ORDER BY or LIMIT.

Aggregates are used to compute values across results within a group. An aggregate operates over a collection of values and produces a single value as a result. SEM_MATCH supports the following built-in Aggregates: COUNT, SUM, MIN, MAX, AVG, GROUP_CONCAT and SAMPLE. These aggregates are described in Table 1-14 (page 1-52).

Table 1-14 Built-in Aggregates

Aggregate	Description
AVG(expression)	Returns the numeric average of *expression* over the values within a group.
COUNT(* | expression)	Counts the number of times *expression* has a bound, non-error value within a group; asterisk (*) counts the number of results within a group.
GROUP_CONCAT(expression [; SEPARATOR = "STRING"])	Performs string concatenation of *expression* over the values within a group. If provided, an optional separator string will be placed between each value.
MAX(expression)	Returns the maximum value of *expression* within a group based on the ordering defined by SPARQL ORDER BY.
MIN(expression)	Returns the minimum value of *expression* within a group based on the ordering defined by SPARQL ORDER BY.
SAMPLE(expression)	Returns *expression* evaluated for a single arbitrary value from a group.
SUM(expression)	Calculates the numeric sum of *expression* over the values within a group.

Certain restrictions on variable references apply when using grouping and aggregation. Only group-by variables (single variables in the GROUP BY clause) and alias variables from GROUP BY value assignments can be used in non-aggregate expressions in the SELECT or HAVING clauses.

Example 1-46 Simple Grouping Query

Example 1-46 (page 1-53) shows a query that uses the GROUP BY keyword to find all the different types of cameras.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?cType
     WHERE
      {?x rdf:type :Camera .
       ?x :cameraType ?cType .
      }
     GROUP BY ?cType',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

A grouping query partitions the query results into a collection of groups based on a grouping expression (`?cType` in Example 1-46 (page 1-53)) such that each result within a group has the same values for the grouping expression. The final result of the grouping operation will include one row for each group.

Example 1-47 Complex Grouping Expression

A grouping expression consists of a sequence of one or more of the following: a variable, an expression, or a value assignment of the form (`<`*`expression`*`> as <`*`alias`*`>`). Example 1-47 (page 1-53) shows a grouping query that uses one of each type of component in the grouping expression.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?cType ?totalPrice
     WHERE
      {?x rdf:type :Camera .
       ?x :cameraType ?cType .
       ?x :manufacturer ?m .
       ?x :price ?p .
       ?x :tax ?t .
      }
     GROUP BY ?cType (STR(?m)) ((?p*(1+?t)) AS ?totalPrice)',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-48 Aggregation

Example 1-48 (page 1-53) uses aggregates to select the maximum, minimum, and average price for each type of camera.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?cType
            (MAX(?p) AS ?maxPrice)
            (MIN(?p) AS ?minPrice)
            (AVG(?p) AS ?avgPrice)
     WHERE
      {?x rdf:type :Camera .
       ?x :cameraType ?cType .
       ?x :manufacturer ?m .
```

```
      ?x :price ?p .
     }
    GROUP BY ?cType',
   SEM_Models('electronics'),
   SEM_Rulebases('RDFS'),
   null, null));
```

Example 1-49 Aggregation Without Grouping

If an aggregate is used without a grouping expression, then the entire result set is treated as a single group. Example 1-49 (page 1-54) computes the total number of cameras for the whole data set.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT (COUNT(?x) as ?cameraCnt)
     WHERE
      { ?x rdf:type :Camera
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-50 Aggregation with DISTINCT

The DISTINCT keyword can optionally be used as a modifier for each aggregate. When DISTINCT is used, duplicate values are removed from each group before computing the aggregate. Syntactically, DISTINCT must appear as the first argument to the aggregate. Example 1-50 (page 1-54) uses DISTINCT to find the number of distinct camera manufacturers. In this case, duplicate values of `STR(?m)` are removed before counting.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT (COUNT(DISTINCT STR(?m)) as ?mCnt)
     WHERE
      { ?x rdf:type :Camera .
        ?x :manufacturer ?m
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-51 HAVING Clause

The HAVING keyword can be used to filter groups based on constraints. HAVING expressions can be composed of variables, RDF terms, arithmetic operators (+, -, *, /), Boolean operators and logical connectives (||, &&, !), comparison operators (<, >, <=, >=, =, !=), aggregates, and any functions available for use in FILTER expressions. Syntactically, the HAVING keyword appears after the GROUP BY clause and before any other solution modifiers such as ORDER BY or LIMIT.

Example 1-51 (page 1-54) uses a HAVING expression to find all manufacturers that sell cameras for less than $200.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?m
     WHERE
```

```
     { ?x rdf:type :Camera .
       ?x :manufacturer ?m .
       ?x :price ?p
     }
     GROUP BY ?m
     HAVING (MIN(?p) < 200)
     ORDER BY ASC(?m)',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

1.6.7.4 Negation

SEM_MATCH supports two forms of negation in SPARQL query patterns: NOT EXISTS and MINUS. NOT EXISTS can be used to filter results based on whether or not a graph pattern matches, and MINUS can be used to remove solutions based on their relation to another graph pattern.

Example 1-52 Negation with NOT EXISTS

Example 1-52 (page 1-55) uses a NOT EXISTS FILTER to select those cameras that do not have any user reviews.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?p
     WHERE
      {?x :price ?p .
       ?x :cameraType ?cType .
       FILTER( NOT EXISTS({?x :userReview ?r}) )
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-53 EXISTS

Conversely, the EXISTS operator can be used to ensure that a pattern matches. Example 1-53 (page 1-55) uses an EXISTS FILTER to select only those cameras that have a user review.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?p
     WHERE
      {?x :price ?p .
       ?x :cameraType ?cType .
       FILTER( EXISTS({?x :userReview ?r}) )
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-54 Negation with MINUS

Example 1-54 (page 1-55) uses MINUS to arrive at the same result as Example 1-52 (page 1-55). Only those solutions that are not compatible with solutions from the MINUS pattern are included in the result. That is, if a solution has the same values for all shared variables as a solution from the MINUS pattern, it is removed from the result.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?p
     WHERE
      {?x :price ?p .
       ?x :cameraType ?cType .
       MINUS {?x :userReview ?r}
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-55 Negation with NOT EXISTS (2)

NOT EXISTS and MINUS represent two different styles of negation and have different results in certain cases. One such case occurs when no variables are shared between the negation pattern and the rest of the query. For example, the NOT EXISTS query in Example 1-55 (page 1-56) removes all solutions because `{?subj ?prop ?obj}` matches any triple, but the MINUS query in Example 1-56 (page 1-56) removes no solutions because there are no shared variables.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?p
     WHERE
      {?x :price ?p .
       ?x :cameraType ?cType .
       FILTER( NOT EXISTS({?subj ?prop ?obj}) )
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-56 Negation with MINUS (2)

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?p
     WHERE
      {?x :price ?p .
       ?x :cameraType ?cType .
       MINUS {?subj ?prop ?obj}
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

1.6.7.5 Value Assignment

SEM_MATCH provides a variety of ways to assign values to variables in a SPARQL query.

The value of an expression can be assigned to a new variable in three ways: (1) expressions in the SELECT clause, (2) expressions in the GROUP BY clause, and (3) the BIND keyword. In each case, the new variable must not already be defined in the query. After assignment, the new variable can be used in the query and returned in results. As discussed in Expressions in the SELECT Clause (page 1-50), the syntax for

value assignment is (*<expression>* AS *<alias>)* where *alias* is the new variable, for example, `((?price * (1+?tax)) AS ?totalPrice)`.

Example 1-57 Nested SELECT Expression

Example 1-57 (page 1-57) uses a nested SELECT expression to compute the total price of a camera and assign the value to a variable (`?totalPrice`). This variable is then used in a FILTER in the outer query to find cameras costing less than $200.

```
SELECT *
 FROM TABLE(SEM_MATCH(
   'PREFIX : <http://www.example.org/electronics/>
    SELECT ?x ?cType ?totalPrice
    WHERE
     {?x :cameraType ?cType .
       { SELECT ?x ( ((?price*(1+?tax)) AS ?totalPrice )
         WHERE { ?x :price ?price .
                 ?x :tax ?tax }
       }
      FILTER (?totalPrice < 200)
     }',
   SEM_Models('electronics'),
   SEM_Rulebases('RDFS'),
   null, null));
```

Example 1-58 BIND

The BIND keyword can be used inside a basic graph pattern to assign a value and is syntactically more compact than an equivalent nested SELECT expression. Example 1-58 (page 1-57) uses the BIND keyword to expresses a query that is logically equivalent to Example 1-57 (page 1-57).

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?totalPrice
     WHERE
      {?x :cameraType ?cType .
       ?x :price ?price .
       ?x :tax ?tax .
       BIND ( ((?price*(1+?tax)) AS ?totalPrice )
       FILTER (?totalPrice < 200)
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-59 GROUP BY Expression

Value assignments in the GROUP BY clause can subsequently be used in the SELECT clause, the HAVING clause, and the outer query (in the case of a nested grouping query). Example 1-59 (page 1-57) uses a GROUP BY expression to find the maximum number of megapixels for cameras at each price point less than $1000.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?totalPrice (MAX(?mp) as ?maxMP)
     WHERE
      {?x rdf:type :Camera .
       ?x :price ?price .
       ?x :tax ?tax .
```

```
      GROUP BY ( ((?price*(1+?tax)) AS ?totalPrice )
      HAVING (?totalPrice < 1000)
     }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-60 VALUES

In addition to the preceding three ways to assign the value of an expression to a new variable, the VALUES keyword can be used to introduce an unordered solution sequence that is combined with the query results through a join operation. A VALUES block can appear inside a query pattern or at the end of a SPARQL SELECT query block after any solution modifiers. The VALUES construct can be used in subqueries.

Example 1-60 (page 1-58) uses the VALUES keyword to constrain the query results to DSLR cameras made by `:Company1` or any type of camera made by `:Company2`. The keyword UNDEF is used to represent an unbound variable in the solution sequence.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?m
     WHERE
      { ?x rdf:type :Camera .
        ?x :cameraType ?cType .
        ?x :manufacturer ?m
      }
     VALUES (?cType ?m)
     { ("DSLR" :Company1)
       (UNDEF  :Company2)
     }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-61 Simplified VALUES Syntax

A simplified syntax can be used for the common case of a single variable. Specifically, the parentheses around the variable and each solution can be omitted. Example 1-61 (page 1-58) uses the simplified syntax to constrain the query results to cameras made by `:Company1` or `:Company2`.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?m
     WHERE
      { ?x rdf:type :Camera .
        ?x :cameraType ?cType .
        ?x :manufacturer ?m
      }
     VALUES ?m
     { :Company1
       :Company2
     }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

Example 1-62 Inline VALUES Block

Example 1-62 (page 1-59) also constrains the query results to any camera made by `:Company1` or `:Company2`, but specifies the VALUES block inside the query pattern.

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?x ?cType ?m
     WHERE
      { VALUES ?m { :Company1 :Company2 }
        ?x rdf:type :Camera .
        ?x :cameraType ?cType .
        ?x :manufacturer ?m
      }',
    SEM_Models('electronics'),
    SEM_Rulebases('RDFS'),
    null, null));
```

1.6.7.6 Property Paths

A SPARQL Property Path describes a possible path between two RDF resources (nodes) in an RDF graph. A property path appears in the predicate position of a triple pattern and uses a regular expression-like syntax to place constraints on the properties (edges) making up a path from the subject of the triple pattern to the object of a triple pattern. Property paths allow SPARQL queries to match arbitrary length paths in the RDF graph and also provide a more concise way to express other graph patterns.

Table 1-15 (page 1-59) describes the syntax constructs available for constructing SPARQL Property Paths. Note that *iri* is either an IRI or a prefixed name, and *elt* is a property path element, which may itself be composed of other property path elements.

Table 1-15 Property Path Syntax Constructs

Syntax Construct	Matches
iri	An IRI or a prefixed name. A path of length 1 (one).
^*elt*	Inverse path (object to subject).
!iri or *!(iri1 | … | irin)*	Negated property set. An IRI that is not one of *irii*.
!^iri or *!(iri1 | … | irij | ^irij+1 | … | ^irin)*	Negated property set with some inverse properties. An IRI that is not one of *irii*, nor one of *irij+1...irin* as reverse paths. *!^iri* is short for *!(^iri)*. The order of properties and inverse properties is not important. They can occur in mixed order.
(elt)	A group path *elt*; brackets control precedence.
elt_1 / elt_2	A sequence path of elt_1, followed by elt_2.
elt_1 | elt_2	An alternative path of elt_1, or elt_2 (all possibilities are tried).
*elt**	A path of zero or more occurrences of *elt*.
elt+	A path of one or more occurrences of *elt*.
elt?	A path of zero or one occurrence of *elt*.

The precedence of the syntax constructs is as follows (from highest to lowest):

- IRI, prefixed names
- Negated property sets
- Groups
- Unary operators *, ?, +
- Unary ^ inverse links
- Binary operator /
- Binary operator |

Precedence is left-to-right within groups.

Special Considerations for Property Path Operators + and *

In general, truly unbounded graph traversals using the + (plus sign) and * (asterisk) operator can be very expensive. For this reason, a depth-limited version of the + and * operator is used by default, and the default depth limit is 10. In addition, the depth-limited implementation can be run in parallel. The `ALL_MAX_PP_DEPTH(n)` SEM_MATCH query option or the `PP_MAX_DEPTH(n)` inline HINT0 query optimizer hint can be used to change the depth-limit setting. To achieve a truly unbounded traversal, you can set a depth limit of less than 1 to fall back to a CONNECT BY-based implementation.

Query Hints for Property Paths

Other query hints are available to influence the performance of property path queries. The `ALLOW_PP_DUP=T` query option can be used with * and + queries to allow duplicate results. Allowing duplicate results may return the first rows from a query faster. In addition, `ALL_USE_PP_HASH` and `ALL_USE_PP_NL` query options are available to influence the join types used when evaluating property path expressions. Analogous `USE_PP_HASH` and `USE_PP_NL` inline HINT0 query optimizer hints can also be used.

Example 1-63 SPARQL Property Path (Using rdfs:subClassOf Relations)

Example 1-63 (page 1-60) uses a property path to find all Males based on transitivity of the `rdfs:subClassOf relationship`. A property path allows matching an arbitrary number of consecutive `rdfs:subClassOf` relations.

```
SELECT x, name
  FROM TABLE(SEM_MATCH(
    '{ ?x foaf:name ?name .
       ?x rdf:type ?t .
       ?t rdfs:subClassOf* :Male }',
    SEM_Models('family'),
    null,
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')
                SEM_ALIAS('foaf',' http://xmlns.com/foaf/0.1/')),
    null));
```

Example 1-64 SPARQL Property Path (Using foaf:friendOf or foaf:knows Relationships)

Example 1-64 (page 1-60) uses a property path to find all of Scott's close friends (those people reachable within two hops using `foaf:friendOf` or `foaf:knows` relationships).

```
SELECT name
  FROM TABLE(SEM_MATCH(
```

```
    '{ { :Scott (foaf:friendOf | foaf:knows) ?f }
       UNION
       { :Scott (foaf:friendOf | foaf:knows)/(foaf:friendOf | foaf:knows) ?f }
       ?f foaf:name ?name .
       FILTER (!sameTerm(?f, :Scott)) }',
    SEM_Models('family'),
    null,
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/'),
                SEM_ALIAS('foaf',' http://xmlns.com/foaf/0.1/')),
    null));
```

Example 1-65 Specifying Property Path Maximum Depth Value

Example 1-65 (page 1-61) specifies a maximum depth of 12 for all property path expressions with the `ALL_MAX_PP_DEPTH(n)` query option value.

```
SELECT x, name
  FROM TABLE(SEM_MATCH(
    '{ ?x foaf:name ?name .
       ?x rdf:type ?t .
       ?t rdfs:subClassOf* :Male }',
    SEM_Models('family'),
    null,
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')
                SEM_ALIAS('foaf',' http://xmlns.com/foaf/0.1/')),
    null,
    null,
    ' ALL_MAX_PP_DEPTH(12) '));
```

Example 1-66 Specifying Property Path Join Hint

Example 1-66 (page 1-61) shows an inline HINT0 query optimizer hint that requests a nested loop join for evaluating the property path expression.

```
SELECT x, name
  FROM TABLE(SEM_MATCH(
    '{ # HINT0={ USE_PP_NL }
       ?x foaf:name ?name .
       ?x rdf:type ?t .
       ?t rdfs:subClassOf* :Male }',
    SEM_Models('family'),
    null,
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')
                SEM_ALIAS('foaf',' http://xmlns.com/foaf/0.1/')),
    null));
```

1.6.8 Graph Patterns: Support for SPARQL 1.1 Federated Query

SEM_MATCH supports SPARQL 1.1 Federated Query (see `http://www.w3.org/TR/sparql11-federated-query/#SPROT`). The SERVICE construct can be used to retrieve results from a specified SPARQL endpoint URL. With this capability, you can combine local RDF data (native RDF data or RDF views of relational data) with other, possibly remote, RDF data served by a W3C standards-compliant SPARQL endpoint.

This section also contains the following topics:

- Privileges Required to Execute Federated SPARQL Queries (page 1-62)
- SPARQL SERVICE Join Push Down (page 1-63)
- SPARQL SERVICE SILENT (page 1-63)

- Using a Proxy Server with SPARQL SERVICE (page 1-64)
- Accessing SPARQL Endpoints with HTTP Basic Authentication (page 1-64)

Example 1-67 SPARQL SERVICE Clause to Retrieve All Triples

Example 1-67 (page 1-62) shows a query that uses a SERVICE clause to retrieve all triples from the SPARQL endpoint available at `http://www.example1.org/sparql`.

```
SELECT s, p, o
  FROM TABLE(SEM_MATCH(
    'SELECT ?s ?p ?o
     WHERE {
       SERVICE <http://www.example1.org/sparql>{ ?s ?p ?o }
     }',
    SEM_Models('electronics'),
    null, null, null, null, ' '));
```

Example 1-68 SPARQL SERVICE Clause to Join Remote and Local RDF Data

Example 1-68 (page 1-62) joins remote RDF data with local RDF data. This example joins camera types `?cType` from local model `electronics` with the camera names `?name` from the SPARQL endpoint at `http://www.example1.org/sparql`.

```
SELECT cType, name
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?cType ?name
     WHERE {
       ?s :cameraType ?cType
       SERVICE <http://www.example1.org/sparql>{ ?s :name ?name }
      }',
    SEM_Models('electronics'),
    null, null, null, null, ' '));
```

1.6.8.1 Privileges Required to Execute Federated SPARQL Queries

You need certain database privileges to use the SERVICE construct within SEM_MATCH queries. You should be granted EXECUTE privilege on the SPARQL_SERVICE MDSYS function by a user with DBA privileges: The following example grants this access to a user named RDFUSER:

```
grant execute on mdsys.sparql_service to rdfuser;
```

Also, an Access Control List (ACL) should be used to grant the CONNECT privilege to the user attempting a federated query. Example 1-69 (page 1-62) creates a new ACL to grant the user RDFUSER the CONNECT privilege and assigns the domain * to the ACL. For more information about ACLs, see *Oracle Database PL/SQL Packages and Types Reference*.

Example 1-69 Access Control List and Host Assignment

```
dbms_network_acl_admin.create_acl (
  acl       => 'rdfuser.xml',
  description => 'Allow rdfuser to query SPARQL endpoints',
  principal => 'RDFUSER',
  is_grant  => true,
  privilege => 'connect'
);

dbms_network_acl_admin.assign_acl (
```

```
  acl  => 'rdfuser.xml',
  host => '*'
);
```

After the necessary privileges are granted, you are ready to execute federated queries from SEM_MATCH

1.6.8.2 SPARQL SERVICE Join Push Down

The SPARQL SERVICE Join Push Down (`SERVICE_JPDWN=T`) feature can be used to improve the performance of certain SPARQL SERVICE queries. By default, the query pattern within the SERVICE clause is executed first on the remote SPARQL endpoint. The full result of this remote execution is then joined with the local portion of the query. This strategy can result in poor performance if the local portion of the query is very selective and the remote portion of the query is very unselective.

The SPARQL SERVICE Join Push Down feature cannot be used in a query that contains more than one SERVICE clause.

Example 1-70 SPARQL SERVICE Join Push Down

Example 1-70 (page 1-63) shows the SPARQL SERVICE Join Push Down feature.

```
SELECT s, prop, obj
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?s ?prop ?obj
     WHERE {
       ?s rdf:type   :Camera .
       ?s :modelName "Camera 12345"
       SERVICE <http://www.example1.org/sparql> { ?s ?prop ?obj }
      }',
    SEM_Models('electronics'),
    null, null, null, null, ' SERVICE_JPDWN=T '));
```

In Example 1-70 (page 1-63), the local portion of the query will return a very small number of rows, but the remote portion of the query is completely unbound and will return the entire remote dataset. When the `SERVICE_JPDWN=T` option is specified, SEM_MATCH performs a nested-loop style evaluation by first executing the local portion of the query and then executing a modified version of the remote query once for each row returned by the local portion. The remote query is modified with a FILTER clause that effectively performs a substitution for the join variable `?s`. For example, if `<urn:camera1>` and `<urn:camera2>` are returned from the local portion of Example 1-70 (page 1-63) as bindings for `?s`, then the following two queries are sent to the remote endpoint: `{ ?s ?prop ?obj FILTER (?s = <urn:camera1>) }` and `{ s ?prop ?obj FILTER (?s = <urn:camera2>) }`.

1.6.8.3 SPARQL SERVICE SILENT

When the SILENT keyword is used in federated queries, errors while accessing the specified remote SPARQL endpoint will be ignored. If the SERVICE SILENT request fails, a single solution with no bindings will be returned.

Example 1-71 (page 1-64) uses SERVICE with the SILENT keyword inside an OPTIONAL clause, so that, when connection errors accessing `http://www.example1.org/sparql` appear, such errors will be ignored and all the rows retrieved from triple `?s :cameratype ?k` will be combined with a null value for `?n`.

Example 1-71 SPARQL SERVICE with SILENT Keyword

```
SELECT s, n
  FROM TABLE(SEM_MATCH(
    'PREFIX : <http://www.example.org/electronics/>
     SELECT ?s ?n
     WHERE {
       ?s :cameraType ?k
       OPTIONAL { SERVICE SILENT <http://www.example1.org/sparql>{ ?k :name ?n } }
      }',
    SEM_Models('electronics'),
    null, null, null, null));
```

1.6.8.4 Using a Proxy Server with SPARQL SERVICE

The following methods are available for sending SPARQL SERVICE requests through an HTTP proxy:

- Specifying the HTTP proxy that should be used for requests in the current session. This can be done through the SET_PROXY function of UTL_HTTP package. Example 1-72 (page 1-64) sets the proxy `proxy.example.com` to be used for HTTP requests, excluding those to hosts in the domain `example2.com`. (For more information about the SET_PROXY procedure, see *Oracle Database PL/SQL Packages and Types Reference*.)
- Using the SERVICE_PROXY SEM_MATCH option, which allows setting the proxy address for SPARQL SERVICE request. However, in this case no exceptions can be specified, and all requests are sent to the given proxy server. Example 1-73 (page 1-64) shows a SEM_MATCH query where the proxy address `proxy.example.com` at port 80 is specified.

Example 1-72 Setting Proxy Server with UTL_HTTP.SET_PROXY

```
BEGIN
  UTL_HTTP.SET_PROXY('proxy.example.com:80', 'example2.com');
END;
/
```

Example 1-73 Setting Proxy Server in SPARQL SERVICE

```
SELECT *
  FROM TABLE(SEM_MATCH(
    'SELECT *
     WHERE {
       SERVICE <http://www.example1.org/sparql>{ ?s ?p ?o }
      }',
    SEM_Models('electronics'),
    null, null, null, null, ' SERVICE_PROXY=proxy.example.com:80 '));
```

1.6.8.5 Accessing SPARQL Endpoints with HTTP Basic Authentication

To allow accessing of SPARQL endpoints with HTTP Basic Authentication, user credentials should be saved in Session Context SDO_SEM_HTTP_CTX. A user with DBA privileges must grant EXECUTE on this context to the user that wishes to use basic authentication. The following example grants this access to a user named RDFUSER:

```
grant execute on mdsys.sdo_sem_http_ctx to rdfuser;
```

After the privilege is granted, the user should save the user name and password for each SPARQL Endpoint with HTTP Authentication through functions

`mdsys.sdo_sem_http_ctx.set_usr` and `mdsys.sdo_sem_http_ctx.set_pwd`. The following example sets a user name and password for the SPARQL endpoint at `http://www.example1.org/sparql`:

```
BEGIN
  mdsys.sdo_sem_http_ctx.set_usr('http://www.example1.org/sparql','user');
  mdsys.sdo_sem_http_ctx.set_pwd('http://www.example1.org/sparql','pwrd');
END;
/
```

1.6.9 Inline Query Optimizer Hints

In SEM_MATCH, the SPARQL comment construct has been overloaded to allow inline HINT0 query optimizer hints. In SPARQL, the hash (#) character indicates that the remainder of the line is a comment. To associate an inline hint with a particular BGP, place a HINT0 hint string inside a SPARQL comment and insert the comment between the opening curly bracket ({) and the first triple pattern in the BGP. Inline hints enable you to influence the execution plan for each BGP in a query. Example 1-74 (page 1-65) shows a query with inline query optimizer hints.

Example 1-74 Inline Query Optimizer Hints

```
SELECT x, y, hp, cp
  FROM TABLE(SEM_MATCH(
    '{ # HINT0={ LEADING(t0) USE_NL(?x ?y ?bd) }
      ?x :grandParentOf ?y . ?x rdf:type :Male . ?x :birthDate ?bd
      OPTIONAL { # HINT0={ LEADING(t0 t1) BGP_JOIN(USE_HASH) }
                 ?x :homepage ?hp . ?x :cellPhoneNum ?cp }
     }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

The BGP_JOIN hint influences inter-BGP joins and has the following syntax: `BGP_JOIN(<join_type>)`, where *<join_type>* is USE_HASH or USE_NL. Example 1-74 (page 1-65) uses the BGP_JOIN(USE_HASH) hint to specify that a hash join should be used when joining the OPTIONAL BGP with its parent BGP.

Inline optimizer hints override any hints passed to SEM_MATCH through the `options` argument. For example, a global ALL_ORDERED hint applies to each BGP that does not specify an inline optimizer hint, but those BGPs with an inline hint use the inline hint instead of the ALL_ORDERED hint.

1.6.10 Full-Text Search

The Oracle-specific `orardf:textContains` SPARQL FILTER function uses full-text indexes on the MDSYS.RDF_VALUE$ table. This function has the following syntax (where `orardf` is a built-in prefix that expands to `<http://xmlns.oracle.com/rdf/>`):

```
orardf:textContains(variable, pattern)
```

The first argument to `orardf:textContains` must be a local variable (that is, a variable present in the BGP that contains the `orardf:textContains` filter), and the second argument must be a constant plain literal.

For example, `orardf:textContains(x, y)` returns true if `x` matches the expression `y`, where `y` is a valid expression for the Oracle Text SQL operator CONTAINS. For more information about such expressions, see *Oracle Text Reference*.

Before using `orardf:textContains`, you must create an Oracle Text index for the RDF network. To create such an index, invoke the SEM_APIS.ADD_DATATYPE_INDEX (page 11-1) procedure as follows:

```
EXECUTE SEM_APIS.ADD_DATATYPE_INDEX('http://xmlns.oracle.com/rdf/text');
```

Performance for wildcard searches like `orardf:textContains(?x, "%abc%")` can be improved by using prefix and substring indexes. You can include any of the following options to the SEM_APIS.ADD_DATATYPE_INDEX (page 11-1) procedure:

- `prefix_index=true` – for adding prefix index
- `prefix_min_length=<number>` – minimum length for prefix index tokens
- `prefix_max_length=<number>` – maximum length for prefix index tokens
- `substring_index=true` – for adding substring index

For more information about Oracle Text indexing elements, see *Oracle Text Reference*.

When performing large bulk loads into a semantic network with a text index, the overall load time may be faster if you drop the text index, perform the bulk load, and then re-create the text index. See Using Data Type Indexes (page 1-88) for more information about data type indexing.

After creating a text index, you can use the `orardf:textContains` FILTER function in SEM_MATCH queries. Example 1-75 (page 1-66) uses `orardf:textContains` to find all grandfathers whose names start with the letter *A* or *B*.

Example 1-75 Full-Text Search

```
SELECT x, y, hp, cp
  FROM TABLE(SEM_MATCH(
    '{ ?x :grandParentOf ?y . ?x rdf:type :Male . ?x :name ?n
       FILTER (orardf:textContains(?n, " A% | B% ")) }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
          null));
```

1.6.11 Spatial Support

RDF Semantic Graph supports storage and querying of spatial geometry data through the OGC GeoSPARQL standard and through Oracle-specific SPARQL extensions. Geometry data can be stored as `orageo:WKTLiteral` or `ogc:wktLiteral` typed literals, and geometry data can be queried using several query functions for spatial operations. Spatial indexing for increased performance is also supported.

`orageo` is a built-in prefix that expands to `<http://xmlns.oracle.com/rdf/geo/>`, `ogc` is a built-in prefix that expands to `<http://www.opengis.net/ont/geosparql#>`, and `ogcf` is a built-in prefix that expands to `<http://www.opengis.net/def/function/geosparql>`.

This section covers the following topics:

- OGC GeoSPARQL Support (page 1-67)
- Representing Spatial Data in RDF (page 1-67)
- Indexing Spatial Data (page 1-68)

- Querying Spatial Data (page 1-69)

1.6.11.1 OGC GeoSPARQL Support

RDF Semantic Graph supports the following conformance classes for the OGC GeoSPARQL standard (`http://www.opengeospatial.org/standards/geosparql`) using well-known text (WKT) serialization and the Simple Features relation family.

- Core
- Topology Vocabulary Extension (*Simple Features*)
- Geometry Extension (*WKT, 1.2.0*)
- Geometry Topology Extension (*Simple Features, WKT, 1.2.0*)
- RDFS Entailment Extension (*Simple Features, WKT, 1.2.0*)

Specifics for representing and querying spatial data using GeoSPARQL are covered in sections that follow this one.

1.6.11.2 Representing Spatial Data in RDF

Spatial geometries can be represented in RDF as `orageo:WKTLiteral` or `ogc:wktLiteral` typed literals.

Example 1-76 Spatial Point Geometry Represented as orageo:WKTLiteral

Example 1-76 (page 1-67) shows the `orageo:WKTLiteral` encoding for a simple point geometry; Example 1-77 (page 1-67) shows the GeoSPARQL encoding for the same point.

```
"Point(-83.4 34.3)"^^<http://xmlns.oracle.com/rdf/geo/WKTLiteral>
```

Example 1-77 Spatial Point Geometry Represented as ogc:wktLiteral

```
"Point(-83.4 34.3)"^^<http://www.opengis.net/ont/geosparql#wktLiteral>
```

Example 1-78 Spatial Data Encoded Using orageo:WKTLiteral Values

Both `orageo:WKTLiteral` and `ogc:wktLiteral` encodings consist of an optional spatial reference system URI, followed by a Well-Known Text (WKT) string that encodes a geometry value. The spatial reference system URI and the WKT string should be separated by a whitespace character. (In this document the term ***geometry literal*** is used to refer to both `orageo:WKTLiteral` and `ogc:wktLiteral` typed literals.)

Supported spatial reference system URIs have the following form `<http://xmlns.oracle.com/rdf/geo/srid/{srid}>`, where {srid} is a valid spatial reference system ID from Oracle Spatial and Graph. If a geometry literal value does not include a spatial reference system URI, then the default spatial reference system, WGS84 Longitude-Latitude (URI `<http://xmlns.oracle.com/rdf/geo/srid/8307>`), is used. The same default spatial reference system is used when geometry literal values are encountered in a query string.

Several geometry types can be represented as geometry literal values, including point, linestring, polygon, polyhedral surface, triangle, TIN, multipoint, multi-linestring, multipolygon, and geometry collection. Up to 500,000 vertices per geometry are supported for two-dimensional geometries.

Example 1-78 (page 1-67) shows some RDF spatial data (in N-triple format) encoded using `orageo:WKTLiteral` values. In this example, the first two geometries (in lot1) use the default coordinate system (SRID 8307), but the other two geometries (in lot2) specify SRID 8265.

```
# spatial data for lot1 using the default WGS84 Longitude-Latitude spatial reference
system
<urn:lot1> <urn:hasExactGeometry> "Polygon((-83.6 34.1, -83.6 34.5, -83.2 34.5,
-83.2 34.1, -83.6 34.1))"^^<http://xmlns.oracle.com/rdf/geo/WKTLiteral> .
<urn:lot1> <urn:hasPointGeometry> "Point(-83.4 34.3)"^^<http://
xmlns.oracle.com/rdf/geo/WKTLiteral> .
# spatial data for lot2 using the NAD83 Longitude-Latitude spatial reference system
<urn:lot2> <urn:hasExactGeometry> "<http://xmlns.oracle.com/rdf/geo/srid/8265>
Polygon((-83.6  34.1, -83.6 34.3, -83.4 34.3, -83.4 34.1, -83.6 34.1))"^^<http://
xmlns.oracle.com/rdf/geo/WKTLiteral> .
<urn:lot2> <urn:hasPointGeometry> "<http://xmlns.oracle.com/rdf/geo/srid/8265>
Point(-83.5 34.2)"^^<http://xmlns.oracle.com/rdf/geo/WKTLiteral> .
```

For more information, see the chapter about coordinate systems (spatial reference systems) in *Oracle Spatial and Graph Developer's Guide*. See also the material about the WKT geometry representation in the Open Geospatial Consortium (OGC) Simple Features document, available at: `http://www.opengeospatial.org/standards/sfa`

1.6.11.3 Indexing Spatial Data

Before you can use any of the SPARQL extension functions (introduced in Querying Spatial Data (page 1-69)) to query spatial data, you must create a spatial index on the RDF network by calling the SEM_APIS.ADD_DATATYPE_INDEX (page 11-1) procedure.

When you create the spatial index, you must specify the following information:

- SRID - The ID for the spatial reference system in which to create the spatial index. Any valid spatial reference system ID from Oracle Spatial and Graph can be used as an SRID value.
- TOLERANCE – The tolerance value for the spatial index. Tolerance is a positive number indicating how close together two points must be to be considered the same point. The units for this value are determined by the default units for the SRID used (for example, meters for WGS84 Long-Lat). Tolerance is explained in detail in *Oracle Spatial and Graph Developer's Guide*.
- DIMENSIONS - A text string encoding dimension information for the spatial index. Each dimension is represented by a sequence of three comma-separated values: name, minimum value, and maximum value. Each dimension is enclosed in parentheses, and the set of dimensions is enclosed by an outer parenthesis.

Example 1-79 Adding a Spatial Data Type Index on RDF Data

Example 1-79 (page 1-68) adds a spatial data type index on the RDF network, specifying the WGS84 Longitude-Latitude spatial reference system, a tolerance value of 10 meters, and the recommended dimensions for the indexing of spatial data that uses this coordinate system. The TOLERANCE, SRID, and DIMENSIONS keywords are case sensitive, and creating a data type index for `<http://xmlns.oracle.com/rdf/geo/WKTLiteral>` will also index `<http://www.opengis.net/ont/geosparql#wktLiteral>` geometry literals, and vice versa (that is, creating a data type index for `<http://www.opengis.net/ont/`

geosparql#wktLiteral> will also index <http://xmlns.oracle.com/rdf/geo/WKTLiteral> geometry literals).

```
EXECUTE sem_apis.add_datatype_index('http://xmlns.oracle.com/rdf/geo/WKTLiteral',
options=>'TOLERANCE=10 SRID=8307 DIMENSIONS=((LONGITUDE,-180,180)
(LATITUDE,-90,90))');
```

No more than one spatial data type index is supported for an RDF network. Geometry literal values stored in the RDF network are automatically normalized to the spatial reference system used for the index, so a single spatial index can simultaneously support geometry literal values from different spatial reference systems. This coordinate transformation is done transparently for indexing and spatial computations. When geometry literal values are returned from a SEM_MATCH query, the original, untransformed geometry is returned.

For more information about spatial indexing, see the chapter about indexing and querying spatial data in *Oracle Spatial and Graph Developer's Guide*.

1.6.11.4 Querying Spatial Data

Several SPARQL extension functions are available for performing spatial queries in SEM_MATCH. For example, for spatial RDF data, you can find the area and perimeter (length) of a geometry, the distance between two geometries, and the centroid and the minimum bounding rectangle (MBR) of a geometry, and you can check various topological relationships between geometries.

SEM_MATCH Support for Spatial Queries (page B-1) contains reference and usage information about the available functions, grouped into two categories:

- GeoSPARQL functions
- Oracle-specific functions

1.6.12 Best Practices for Query Performance

This section describes some recommended practices for using the SEM_MATCH table function to query semantic data. It includes the following subsections:

- FILTER Constructs Involving xsd:dateTime_ xsd:date_ and xsd:time (page 1-69)
- Function-Based Indexes for FILTER Constructs Involving Typed Literals (page 1-70)
- FILTER Constructs Involving Relational Expressions (page 1-70)
- Optimizer Statistics and Dynamic Sampling (page 1-70)
- Multi-Partition Queries (page 1-71)
- Compression on Systems with OLTP Index Compression (page 1-71)
- Unbounded Property Path Expressions (page 1-71)
- Grouping and Aggregation (page 1-72)

1.6.12.1 FILTER Constructs Involving xsd:dateTime, xsd:date, and xsd:time

By default, SEM_MATCH complies with the XML Schema standard for comparison of xsd:date, xsd:time, and xsd:dateTime values. According to this standard, when comparing two calendar values c1 and c2 where c1 has an explicitly specified time

zone and c2 does not have a specified time zone, c2 is converted into the interval [c2-14:00, c2+14:00]. If c2-14:00 <= c1 <= c2+14:00, then the comparison is undefined and will always evaluate to false. If c1 is outside this interval, then the comparison is defined.

However, the extra logic required to evaluate such comparisons (value with a time zone and value without a time zone) can significantly slow down queries with FILTER constructs that involve calendar values. For improved query performance, you can disable this extra logic by specifying `FAST_DATE_FILTER=T` in the `options` parameter of the SEM_MATCH table function. When `FAST_DATE_FILTER=T` is specified, all calendar values without time zones are assumed to be in Greenwich Mean Time (GMT).

Note that using `FAST_DATE_FILTER=T` does *not* affect query *correctness* when either (1) all calendar values in the data set have a time zone or (2) all calendar values in the data set do not have a time zone.

1.6.12.2 Function-Based Indexes for FILTER Constructs Involving Typed Literals

The evaluation of SEM_MATCH queries involving the FILTER construct often requires executing one or more SQL functions against the RDF_VALUE$ table. For example, the filter `(?x < "1929-11-16Z"^^xsd:date)` invokes the SEM_APIS.GETV$DATETZVAL (page 11-53) function.

Function-based indexes can be used to improve the performance of queries that contain a filter condition involving a typed literal. For example, an `xsd:date` function-based index may speed up evaluation of the filter `(?x < "1929-11-16Z"^^xsd:date)`.

Convenient interfaces are provided for creating, altering, and dropping these function-based indexes. For more information, see Using Data Type Indexes (page 1-88).

Note, however, that the existence of these function-based indexes on the MDSYS.RDF_VALUE$ table can significantly slow down bulk load operations. In many cases it may be faster to drop the indexes, perform the bulk load, and then re-create the indexes, as opposed to doing the bulk load with the indexes in place.

1.6.12.3 FILTER Constructs Involving Relational Expressions

The following recommendations apply to FILTER constructs involving relational expressions:

- The `sameTerm` built-in function is more efficient than using `=` or `!=` when comparing two variables in a FILTER clause, so (for example) use `sameTerm(?a, ?b)` instead of `(?a = ?b)` and use `(!sameTerm(?a, ?b))` instead of `(?a != ?b)` whenever possible.
- When comparing values in FILTER expressions, you may get better performance by reducing the use of negation. For example, it is more efficient to evaluate `(?x <= "10"^^xsd:int)` than it is to evaluate the expression `(!(?x > "10"^^xsd:int))`.

1.6.12.4 Optimizer Statistics and Dynamic Sampling

Having sufficient statistics for the query optimizer is critical for good query performance. In general, you should ensure that you have gathered basic statistics for the semantic network using the SEM_PERF.GATHER_STATS (page 13-3) procedure (described in SEM_PERF Package Subprograms (page 13-1)).

Due to the inherent flexibility of the RDF data model, static information may not produce optimal execution plans for SEM_MATCH queries. Dynamic sampling can often produce much better query execution plans. Dynamic sampling levels can be set at the session or system level using the `optimizer_dynamic_sampling` parameter, and at the individual query level using the `dynamic_sampling(level)` SQL query hint. In general, it is good to experiment with dynamic sampling levels between 3 and 6. For information about estimating statistics with dynamic sampling, see *Oracle Database SQL Tuning Guide*.

Example 1-80 (page 1-71) uses a SQL hint for a dynamic sampling level of 6.

Example 1-80 SQL Hint for Dynamic Sampling

```
SELECT /*+ DYNAMIC_SAMPLING(6) */ x, y
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y .
      ?x rdf:type :Male .
      ?x :birthDate ?bd }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null, null, '' ));
```

1.6.12.5 Multi-Partition Queries

The following recommendations apply to the use of multiple semantic models, semantic models plus entailments, and virtual models:

- If you execute SEM_MATCH queries against multiple semantic models or against semantic models plus entailments, you can probably improve query performance if you create a virtual model (see Virtual Models (page 1-13)) that contains all the models and entailments you are querying and then query this single virtual model.
- Use the `ALLOW_DUP=T` query option. If you do not use this option, then an expensive (in terms of processing) duplicate-elimination step is required during query processing, in order to maintain set semantics for RDF data. However, if you use this option, the duplicate-elimination step is not performed, and this results in significant performance gains.

1.6.12.6 Compression on Systems with OLTP Index Compression

On systems where OLTP index compression is supported (such as Exadata). you can take advantage of the feature to improve the compression ratio for some of the B-tree indexes used by the semantic network.

For example, a DBA can use the following command to change the compression scheme on the MDSYS.RDF_VAL_NAMETYLITLNG_IDX index from prefix compression to OLTP index compression:

```
SQL> alter index mdsys.RDF_VAL_NAMETYLITLNG_IDX rebuild compress for oltp high;
```

1.6.12.7 Unbounded Property Path Expressions

A depth-limited search should be used for + and * property path operators whenever possible. The depth-limited implementation for * and + is likely to significantly outperform the CONNECT BY-based implementation in large and/or highly connected graphs. A depth limit of 10 is used by default. For a given graph, depth limits larger than the graph's diameter are not useful. See Property Paths (page 1-59) for more information on setting depth limits.

A backward chaining style inference using `rdfs:subClassOf+` for ontologies with very deep class hierarchies may be an exception to this rule. In such cases, unbounded CONNECT BY-based evaluations may perform better than depth-limited evaluations with very high depth limits (for example, 50).

1.6.12.8 Grouping and Aggregation

`MIN`, `MAX` and `GROUP_CONCAT` aggregates require special logic to fully capture SPARQL semantics for input of non-uniform type (for example, `MAX(?x)`). For certain cases where a uniform input type can be determined at compile time (for example, `MAX(STR(?x))` – plain literal input), optimizations for built-in SQL aggregates can be used. Such optimizations generally give an order of magnitude increase in performance. The following cases are optimized:

- MIN/MAX(<plain literal>)
- MIN/MAX(<numeric>)
- MIN/MAX(<dateTime>)
- GROUP_CONCAT(<plain literal>)

Example 1-81 (page 1-72) uses MIN/MAX(<numeric>) optimizations.

Example 1-81 Aggregate Optimizations

```
SELECT dept, minSal, maxSal
  FROM TABLE(SEM_MATCH(
    'SELECT ?dept (MIN(xsd:decimal(?sal)) AS ?minSal) (MAX(xsd:decimal(?sal)) AS ?
maxSal)
     WHERE
       {?x :salary ?y .
        ?x :department ?dept }
     GROUP BY ?dept',
    SEM_Models('hr_data'),
    null, null, null, null, '' ));
```

1.6.13 Special Considerations When Using SEM_MATCH

The following considerations apply to SPARQL queries executed by RDF Semantic Graph using SEM_MATCH:

- Value assignment
  - A compile-time error is raised when undefined variables are referenced in the source of a value assignment.
- Grouping and aggregation
  - Non-grouping variables (query variables not used for grouping and therefore not valid for projection) cannot be reused as a target for value assignment.
  - Non-numeric values are ignored by the AVG and SUM aggregates.
  - By default, SEM_MATCH returns no rows for an aggregate query with a graph pattern that fails to match. The W3C specification requires a single, null row for this case. W3C-compliant behavior can be obtained with the `STRICT_AGG_CARD=T` query option for a small performance penalty.
- ORDER BY

  - When using SPARQL ORDER BY in SEM_MATCH, the containing SQL query should be ordered by SEM$ROWNUM to ensure that the desired ordering is maintained through any enclosing SQL blocks.

- Numeric computations

  - The native Oracle NUMBER type is used internally for all arithmetic operations, and the results of all arithmetic operations are serialized as `xsd:decimal`. Note that the native Oracle NUMBER type is more precise than both BINARY_FLOAT and BINARY_DOUBLE. See *Oracle Database SQL Language Reference* for more information on the NUMBER built-in data type.

  - Division by zero causes a runtime error instead of producing an unbound value.

- Negation

  - EXISTS and NOT EXISTS filters that reference *potentially unbound variables* are not supported in the following contexts:

    * Non-aliased expressions in GROUP BY

    * Input to aggregates

    * Expressions in ORDER BY

    * FILTER expressions within OPTIONAL graph patterns that also reference variables that do not appear inside of the OPTIONAL graph pattern

    The first three cases can be realized by first assigning the result of the EXISTS or NOT EXISTS filter to a variable using a BIND clause or SELECT expression.

    These restrictions do *not* apply to EXISTS and NOT EXISTS filters that only reference definitely bound variables.

- Blank nodes

  - Blank nodes are not supported within graph patterns.

  - The `BNODE(literal)` function returns the same blank node value every time it is called with the same literal argument.

- Property paths

  - Unbounded operators + and * use a 10-hop depth limit by default for performance reasons. This behavior can be changed to a truly unbounded search by setting a depth limit of 0. See Property Paths (page 1-59) for details.

- Long literals (CLOBs)

  - SPARQL functions and aggregates do not support long literals by default.

  - Specifying the `CLOB_EXP_SUPPORT=T` query option enables long literal support for the following SPARQL functions: IF, COALESCE, STRLANG, STRDT, SUBSTR, STRBEFORE, STRAFTER, CONTAINS, STRLEN, STRSTARTS, STRENDS.

  - Specifying the `CLOB_AGG_SUPPORT=T` query option enables long literal support for the following aggregates: MIN, MAX, SAMPLE, GROUP_CONCAT.

- Canonicalization of RDF literals
  - By default, RDF literals returned from SPARQL functions and constant RDF literals used in value assignment statements (BIND, SELECT expressions, GROUP BY expressions) are canonicalized. This behavior is consistent with the SPARQL 1.1 D-Entailment Regime.
  - Canonicalization can be disabled with the `PROJ_EXACT_VALUES=T` query option.

1.7 Loading and Exporting Semantic Data

To load semantic data into a model, use one or more of the following options:

- Bulk load or append data into the semantic data store from a staging table, with each row containing the three components -- subject, predicate, and object -- of an RDF triple and optionally a named graph. This is explained in Bulk Loading Semantic Data Using a Staging Table (page 1-74).

  This is the fastest option for loading large amounts of data; however, it cannot handle triples containing object values with more than 4000 bytes.

- Batch load using a Java client interface to load or append data from an N-Triple format file into the semantic data store (see Batch Loading N-Triple Format Semantic Data Using the Java API (page 1-77)).

  This option is slower than bulk loading, but it handles triples containing object values with more than 4000 bytes. However, this option does not handle named graphs.

- Load into the application table using SQL INSERT statements that call the SDO_RDF_TRIPLE_S constructor, which results in the corresponding RDF triple, possibly including a graph name, to be inserted into the semantic data store, as explained in Loading Semantic Data Using INSERT Statements (page 1-79).

  This option is convenient for loading small amounts of data.

To export semantic data, that is, to retrieve semantic data from Oracle Database where the results are in N-Triple or N-Quad format that can be stored in a staging table, use the SQL queries described in Exporting Semantic Data (page 1-79).

Note:

Effective with Oracle Database Release 12.1, you can export and import a semantic network using the full database export and import features of the Oracle Data Pump utility, as explained in Exporting or Importing a Semantic Network Using Oracle Data Pump (page 1-82).

1.7.1 Bulk Loading Semantic Data Using a Staging Table

You can load semantic data (and optionally associated non-semantic data) in bulk using a staging table. Call the SEM_APIS.LOAD_INTO_STAGING_TABLE (page 11-60) procedure (described in SEM_APIS Package Subprograms (page 11-1))

to load the data, and you can have during the load operation to check for syntax correctness. Then, you can call the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE (page 11-11) procedure to load the data into the semantic store from the staging table. (If the data was not parsed during the load operation into the staging table, you must specify the PARSE keyword in the `flags` parameter when you call the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE (page 11-11) procedure.)

The following example shows the format for the staging table, including all required columns and the required names for these columns, plus the optional RDF$STC_graph column which must be included if one or more of the RDF triples to be loaded include a graph name:

```
CREATE TABLE stage_table (
                    RDF$STC_sub varchar2(4000) not null,
                    RDF$STC_pred varchar2(4000) not null,
                    RDF$STC_obj varchar2(4000) not null,
                    RDF$STC_graph varchar2(4000)
);
```

If you also want to load non-semantic data, specify additional columns for the non-semantic data in the CREATE TABLE statement. The non-semantic column names must be different from the names of the required columns. The following example creates the staging table with two additional columns (SOURCE and ID) for non-semantic attributes.

```
CREATE TABLE stage_table_with_extra_cols (
                    source VARCHAR2(4000),
                    id NUMBER,
                    RDF$STC_sub varchar2(4000) not null,
                    RDF$STC_pred varchar2(4000) not null,
                    RDF$STC_obj varchar2(4000) not null,
                    RDF$STC_graph varchar2(4000)
);
```

Note:

For either form of the CREATE TABLE statement, you may want to add the COMPRESS clause to use table compression, which will reduce the disk space requirements and may improve bulk-load performance.

Both the invoker and the MDSYS user must have the following privileges: SELECT privilege on the staging table, and INSERT privilege on the application table.

See also the following:

- Loading the Staging Table (page 1-75)
- Recording Event Traces During Bulk Loading (page 1-77)

1.7.1.1 Loading the Staging Table

You can load semantic data into the staging table, as a preparation for loading it into the semantic store, in several ways. Some of the common ways are the following:

- Using Oracle SQL*Loader to load the staging table, as described in Loading N-Triple Format Data into a Staging Table Using SQL*Loader (page 1-76)

- Using an external table to load the staging table, as described in Loading N-Quad Format Data into a Staging Table Using an External Table (page 1-76)

1.7.1.1.1 Loading N-Triple Format Data into a Staging Table Using SQL*Loader

You can use the SQL*Loader utility to parse and load semantic data into a staging table. If you installed the demo files from the Oracle Database Examples media (see *Oracle Database Examples Installation Guide*), a sample control file is available at `$ORACLE_HOME/md/demo/network/rdf_demos/bulkload.ctl`. You can modify and use this file if the input data is in N-Triple format.

Objects longer than 4000 bytes cannot be loaded. If you use the sample SQL*Loader control file, triples (rows) containing such long values will be automatically rejected and stored in a SQL*Loader "bad" file. However, you can load these rejected rows by inserting them into the application table using SQL INSERT statements (see Loading Semantic Data Using INSERT Statements (page 1-79)).

1.7.1.1.2 Loading N-Quad Format Data into a Staging Table Using an External Table

You can use an Oracle external table to load N-Quad format data (extended triple having four components) into a staging table, as follows:

1. Call the SEM_APIS.CREATE_SOURCE_EXTERNAL_TABLE (page 11-29) procedure to create an external table, and then use the SQL STATEMENT ALTER TABLE to alter the external table to include the relevant input file name or names. You must have READ and WRITE privileges for the directory object associated with folder containing the input file or files.
2. After you create the external table, grant the MDSYS user SELECT and INSERT privileges on the table.
3. Call the SEM_APIS.LOAD_INTO_STAGING_TABLE (page 11-60) procedure to populate the staging table.
4. After the loading is finished, issue a COMMIT statement to complete the transaction.

Example 1-82 Using an External Table to Load a Staging Table

```
-- Create a source external table (note: table names are case sensitive)
BEGIN
  sem_apis.create_source_external_table(
    source_table    => 'stage_table_source'
   ,def_directory   => 'DATA_DIR'
   ,bad_file        => 'CLOBrows.bad'
   );
END;
/
grant SELECT on "stage_table_source" to MDSYS;

-- Use ALTER TABLE to target the appropriate file(s)
alter table "stage_table_source" location ('demo_datafile.nt');

-- Load the staging table (note: table names are case sensitive)
BEGIN
  sem_apis.load_into_staging_table(
    staging_table => 'STAGE_TABLE'
   ,source_table  => 'stage_table_source'
   ,input_format  => 'N-QUAD');
END;
/
```

Rows where the objects and graph URIs (combined) are longer than 4000 bytes will be rejected and stored in a "bad" file. However, you can load these rejected rows by inserting them into the application table using SQL INSERT statements (see Loading Semantic Data Using INSERT Statements (page 1-79)).

Example 1-82 (page 1-76) shows the use of an external table to load a staging table.

1.7.1.2 Recording Event Traces During Bulk Loading

If a table named RDF$ET_TAB exists in the invoker's schema and if the MDSYS user has been granted the INSERT and UPDATE privileges on this table, event traces for some of the tasks performed during executions of the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE (page 11-11) procedure will be added to the table. You may find the content of this table useful if you ever need to report any problems in bulk load. The RDF$ET_TAB table must be created as follows:

```
CREATE TABLE RDF$ET_TAB (
  proc_sid VARCHAR2(30),
  proc_sig VARCHAR2(200),
  event_name varchar2(200),
  start_time timestamp,
  end_time timestamp,
  start_comment varchar2(1000) DEFAULT NULL,
  end_comment varchar2(1000) DEFAULT NULL
);
GRANT INSERT, UPDATE on RDF$ET_TAB to MDSYS;
```

1.7.2 Batch Loading N-Triple Format Semantic Data Using the Java API

Note:

The Java class `oracle.spatial.rdf.client.BatchLoader` described in this section has been deprecated, and it does not support loading of N-Quad data.

You are instead encouraged to use the bulk loading capabilities of the RDF Semantic Graph support for Apache Jena, as described in Bulk Loading Using RDF Semantic Graph Support for Apache Jena (page 7-44).

You can perform a batch (bulk) load operation for N-Triple format semantic data using the Java class `oracle.spatial.rdf.client.BatchLoader`, which is packaged in `<ORACLE_HOME>/md/jlib/sdordf.jar`. Before performing a batch load operation, ensure that the following are true:

- The semantic data is in N-Triple format. (Several tools are available for converting RDF/XML to N-Triple format; see the Oracle Technology Network or perform a Web search for information about RDF/XML to N-Triple conversion.)
- Oracle Database Release 11 or later, with Oracle Spatial and Graph, is installed, and partitioning is enabled.
- A semantic technologies network, an application table, and its corresponding semantic model have been created in the database.
- The CLASSPATH definition includes `ojdbc6.jar`.

- You are using JDK version 1.5 or later. (You can use the Java version packaged under `<ORACLE_HOME>/jdk/bin`.)

To run the `oracle.spatial.rdf.client.BatchLoader` class, use a command (on a single command line) in the following general form (replacing the sample example database connection information with your own connection information).

- Linux systems:

```
java -Ddb.user=scott -Ddb.password=password -Ddb.host=127.0.0.1 -Ddb.port=1522 -
Ddb.sid=orcl -classpath ${ORACLE_HOME}/md/jlib/sdordf.jar:${ORACLE_HOME}/
jdbc/lib/ojdbc6.jar oracle.spatial.rdf.client.BatchLoader <N-TripleFile>
<tablename> <tablespaceName> <modelName>
```

- Windows systems:

```
java -Ddb.user=scott -Ddb.password=password -Ddb.host=127.0.0.1 -Ddb.port=1522 -
Ddb.sid=orcl -classpath %ORACLE_HOME%\md\jlib\sdordf.jar;%ORACLE_HOME%\jdbc\lib
\ojdbc6.jar oracle.spatial.rdf.client.BatchLoader <N-TripleFile> <tablename>
<tablespaceName> <modelName>
```

The values for `-Ddb.user` and `-Ddb.password` must correspond either to the owner of the model `<modelName>` or to a DBA user.

By default, `BatchLoader` assumes there are at least two columns, a column named ID of type NUMBER and a column named TRIPLE of type SDO_RDF_TRIPLE_S, in the user's application table. However, you can override the default names by using the JVM properties `-DidColumn=<idColumnName>` and `-DtripleColumn=<tripleColumnName>`. The ID column is not required; and to prevent `BatchLoader` from generating a sequence-like identifier in the ID column for each triple inserted, specify the JVM property `-DjustTriple=true`.

If the application table is not empty and if you want the batch loading to be done in append mode, specify an additional JVM property: `-Dappend=true`. Moreover, in append mode you might want to choose a different starting value for ID column in user's application table, and to accomplish this you can add the JVM property `-DstartID=<startingIntegerValue>` to the command line. By default, the ID column starts at 1 and is increased sequentially as new triples are inserted into the application table.

To skip the first *n* triples in `<N-TripleFile>`, add the JVM property `-Dskip=<numberOfTriplesSkipped>` to the command line.

To load an N-Triple file with a character set different from the default, specify the JVM property `-Dcharset=<charsetName>`. For example, `-Dcharset="UTF-8"` will recognize UTF-8 encoding. However, for UTF-8 characters to be stored properly in the N-Triple file, the Oracle database must be configured to use a corresponding universal character set, such as AL32UTF8.

The `BatchLoader` class supports loading an N-Triple file in compressed format. If the `<N-TripleFile>` has a file extension of .zip or .jar, the file will be uncompressed and loaded at the same time.

See Also:

Bulk Loading Using RDF Semantic Graph Support for Apache Jena (page 7-44)

1.7.3 Loading Semantic Data Using INSERT Statements

To load semantic data using INSERT statements, the data should be encoded using < > (angle brackets) for URIs, _: (underscore colon) for blank nodes, and " " (quotation marks) for literals. Spaces are not allowed in URIs or blank nodes. Use the SDO_RDF_TRIPLE_S constructor to insert the data, as described in Constructors for Inserting Triples (page 1-21). You must have INSERT privilege on the application table.

Note:

If URIs are not encoded with < > and literals with " ", statements will still be processed. However, the statements will take longer to load, since they will have to be further processed to determine their VALUE_TYPE values.

The following example includes statements with URIs, a blank node, a literal, a literal with a language tag, and a typed literal:

```
INSERT INTO nsu_data VALUES (SDO_RDF_TRIPLE_S('nsu', '<http://nature.example.com/nsu/rss.rdf>',
  '<http://purl.org/rss/1.0/title>', '"Nature''s Science Update"'));
INSERT INTO nsu_data VALUES (SDO_RDF_TRIPLE_S('nsu', '_:BNSEQN1001A',
  '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
  '<http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq>'));
INSERT INTO nsu_data VALUES (SDO_RDF_TRIPLE_S('nsu',
  '<http://nature.example.com/cgi-taf/dynapage.taf?file=/nature/journal/v428/n6978/index.html>',
  '<http://purl.org/dc/elements/1.1/language>', '"English"@en-GB'));
INSERT INTO nature VALUES (SDO_RDF_TRIPLE_S('nsu', '<http://dx.doi.org/10.1038/428004b>',
  '<http://purl.org/dc/elements/1.1/date>', '"2004-03-04"^^xsd:date'));
```

To convert semantic XML data to INSERT statements, you can edit the sample `rss2insert.xsl` XSLT file to convert all the features in the semantic data XML file. The blank node constructor is used to insert statements with blank nodes. After editing the XSLT, download the Xalan XSLT processor (`http://xml.apache.org/xalan-j/`) and follow the installation instructions. To convert a semantic data XML file to INSERT statements using your edited version of the `rss2insert.xsl` file, use a command in the following format:

```
java org.apache.xalan.xslt.Process -in input.rdf -xsl rss2insert.xsl -out output.nt
```

1.7.3.1 Loading Data into Named Graphs Using INSERT Statements

To load an RDF triple with a non-null graph name using an INSERT statement, you must append the graph name, enclosed within angle brackets (< >), after the model name and colon (:) separator character, as shown in the following example:

```
INSERT INTO articles_rdf_data VALUES (99,
  SDO_RDF_TRIPLE_S ('articles:<http://examples.com/ns#Graph1>',
    '<http://nature.example.com/Article101>',
    '<http://purl.org/dc/elements/1.1/creator>',
    '"John Smith"'));
```

1.7.4 Exporting Semantic Data

This section contains the following topics related to exporting semantic data, that is, retrieving semantic data from Oracle Database where the results are in N-Triple or N-Quad format that can be stored in a staging table.

- Retrieving Semantic Data from an Application Table (page 1-80)
- Retrieving Semantic Data from an RDF Model (page 1-81)
- Removing Model and Graph Information from Retrieved Blank Node Identifiers (page 1-81)

1.7.4.1 Retrieving Semantic Data from an Application Table

Semantic data can be retrieved from an application table using the member functions of SDO_RDF_TRIPLE_S, as shown in Example 1-83 (page 1-80).

Example 1-83 Retrieving Semantic Data from an Application Table

```
--
-- Retrieves model-graph, subject, predicate, and object
--
SQL> SELECT a.triple.GET_MODEL() AS model_graph, a.triple.GET_SUBJECT() AS sub,
a.triple.GET_PROPERTY() pred, a.triple.GET_OBJECT() obj FROM articles_rdf_data a
where id in (2,99);

MODEL_GRAPH
------------------------------------------------------------
SUB
------------------------------------------------------------
PRED
------------------------------------------------------------
OBJ
------------------------------------------------------------
ARTICLES
<http://nature.example.com/Article1>
<http://purl.org/dc/elements/1.1/creator>
"Jane Smith"

ARTICLES:<http://examples.com/ns#Graph1>
<http://nature.example.com/Article101>
<http://purl.org/dc/elements/1.1/creator>
"John Smith"

--
-- Retrieves graph, subject, predicate, and object
--
SQL> select (case sep_pos when 0 then NULL else substr(model_graph,sep_pos+1) end)
graph, sub, pred, obj from (SELECT instr(a.triple.GET_MODEL(),':') AS sep_pos,
a.triple.GET_MODEL() AS model_graph, a.triple.GET_SUBJECT() AS sub,
a.triple.GET_PROPERTY() pred, a.triple.GET_OBJECT() obj FROM articles_rdf_data a
where id in (2,99));

GRAPH
--------------------------------------------------------------------------------
SUB
------------------------------------------------------------
PRED
------------------------------------------------------------
OBJ
------------------------------------------------------------

<http://nature.example.com/Article1>
<http://purl.org/dc/elements/1.1/creator>
"Jane Smith"

<http://examples.com/ns#Graph1>
```

```
<http://nature.example.com/Article101>
<http://purl.org/dc/elements/1.1/creator>
"John Smith"
```

1.7.4.2 Retrieving Semantic Data from an RDF Model

Semantic data can be retrieved from an RDF model using the SEM_MATCH table function (described in Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22)), as shown in Example 1-84 (page 1-81).

Example 1-84 Retrieving Semantic Data from an RDF Model

```
--
-- Retrieves graph, subject, predicate, and object
--
SQL> select to_char(g$rdfterm) graph, to_char(x$rdfterm) sub, to_char(p$rdfterm)
pred, y$rdfterm obj from table(sem_match('Select ?g ?x ?p ?y FROM NAMED <http://
examples.com/ns#Graph1> {GRAPH ?g {?x ?p ?
y}}',sem_models('articles'),null,null,null,null,' GRAPH_MATCH_UNNAMED=T PLUS_RDFT=T
'));

GRAPH
------------------------------------------------------------
SUB
------------------------------------------------------------
PRED
------------------------------------------------------------
OBJ
--------------------------------------------------------------------------------
<http://examples.com/ns#Graph1>
_:m99g3C687474703A2F2F6578616D706C65732E636F6D2F6E73234772617068313Egmb2
<http://purl.org/dc/elements/1.1/creator>
_:m99g3C687474703A2F2F6578616D706C65732E636F6D2F6E73234772617068313Egmb1

<http://examples.com/ns#Graph1>
<http://nature.example.com/Article102>
<http://purl.org/dc/elements/1.1/creator>
_:m99g3C687474703A2F2F6578616D706C65732E636F6D2F6E73234772617068313Egmb1

<http://examples.com/ns#Graph1>
<http://nature.example.com/Article101>
<http://purl.org/dc/elements/1.1/creator>
"John Smith"

<http://nature.example.com/Article1>
<http://purl.org/dc/elements/1.1/creator>
"Jane Smith"
```

1.7.4.3 Removing Model and Graph Information from Retrieved Blank Node Identifiers

Blank node identifiers retrieved during the retrieval of semantic data can be trimmed to remove the occurrence of model and graph information using the transformations shown in the code excerpt in Example 1-85 (page 1-82), which are applicable to VARCHAR2 (for example, subject component) and CLOB (for example, object component) data, respectively.

Example 1-86 (page 1-82) shows the results obtained after using these two transformations in Example 1-85 (page 1-82) on the `sub` and `obj` columns, respectively, using the semantic data retrieval query described in Retrieving Semantic Data from an RDF Model (page 1-81).

Example 1-85 Retrieving Semantic Data from an Application Table

```
--
-- Transformation on column "sub VARCHAR2"
-- holding blank node identifier values
--
Select (case substr(sub,1,2) when '_:' then '_:' || substr(sub,instr(sub,'m',1,2)+1)
else sub end) from …
--
-- Transformation on column "obj CLOB"
-- holding blank node identifier values
--
Select (case dbms_lob.substr(obj,2,1) when '_:' then to_clob('_:' ||
substr(obj,instr(obj,'m',1,2)+1)) else obj end) from …
```

Example 1-86 Results from Applying Transformations from* Example 1-85 *(page 1-82)

```
--
-- Results obtained by applying transformations on the sub and pred cols
--
SQL> select (case substr(sub,1,2) when '_:' then '_:' || substr(sub,instr(sub,'m',
1,2)+1) else sub end) sub, pred, (case dbms_lob.substr(obj,2,1) when '_:' then
to_clob('_:' || substr(obj,instr(obj,'m',1,2)+1)) else obj end) obj from (select
to_char(g$rdfterm) graph, to_char(x$rdfterm) sub, to_char(p$rdfterm) pred, y$rdfterm
obj from table(sem_match('Select ?g ?x ?p ?y FROM NAMED <http://examples.com/
ns#Graph1> {GRAPH ?g {?x ?p ?y}}',sem_models('articles'),null,null,null,null,'
GRAPH_MATCH_UNNAMED=T PLUS_RDFT=T ')));

SUB
------------------------------------------------------------
PRED
------------------------------------------------------------
OBJ
--------------------------------------------------------------------------------
_:b2
<http://purl.org/dc/elements/1.1/creator>
_:b1

<http://nature.example.com/Article102>
<http://purl.org/dc/elements/1.1/creator>
_:b1
```

1.7.5 Exporting or Importing a Semantic Network Using Oracle Data Pump

Effective with Oracle Database Release 12.1, you can export and import a semantic network using the full database export and import features of the Oracle Data Pump utility. The network is moved as part of the full database export or import, where the whole database is represented in an Oracle dump (`.dmp`) file.

The following usage notes apply to using Data Pump to export or import a semantic network:

- The target database for an import must have the RDF Semantic Graph software installed, and there cannot be a pre-existing semantic network.
- Semantic networks using fine-grained access control (triple-level or resource-level OLS or VPD) cannot be exported or imported.
- Version-enabled semantic networks using Workspace Manager cannot be exported or imported.

- Semantic document indexes for SEM_CONTAINS (MDSYS.SEMCONTEXT index type) and semantic indexes for SEM_RELATED (MDSYS.SEM_INDEXTYPE index type) must be dropped before an export and re-created after an import.
- Only default privileges for semantic network objects (those that exist just after object creation) are preserved during export and import. For example, if user A creates semantic model `M` and grants SELECT on MDSYS.RDFM_M to user B, only user A's SELECT privilege on MDSYS.RDFM_M will be present after the import. User B will not have SELECT privilege on MDSYS.RDFM_M after the import. Instead, user B's SELECT privilege will have to be granted again.
- The Data Pump command line option `transform=oid:n` must be used when exporting or importing semantic network data. For example, use a command in the following format:

  ```
  impdp system/<password-for-system> directory=dpump_dir dumpfile=rdf.dmp full=YES
  version=12 transform=oid:n
  ```

For Data Pump usage information and examples, see the relevant chapters in Part I of *Oracle Database Utilities*.

1.7.6 Purging Unused Values

Deletion of triples over time may lead to a subset of the values in the RDF_VALUE$ table becoming unused in any of the RDF triples or rules currently in the semantic network. If the count of such unused values becomes large and a significant portion of the RDF_VALUE$ table, you may want to purge the unused values using the SEM_APIS.PURGE_UNUSED_VALUES (page 11-64) subprogram.

Before the purging, MDSYS must be granted SELECT privilege on application tables for all the RDF models. This can be done directly using the GRANT command or by using the SEM_APIS.PRIVILEGE_ON_APP_TABLES (page 11-63) subprogram.

Event traces for tasks performed during the purge operation may be recorded into the RDF$ET_TAB table, if present in the invoker's schema, as described in Recording Event Traces During Bulk Loading (page 1-77).

The following example purges unused values from the RDF_VALUE$ table. The example does not consider named graphs or CLOBs. It also assumes that the data from the example in Example: Journal Article Information (page 1-94) has been loaded.

Example 1-87 Purging Unused Values

```
-- Purging unused values
set numwidth 20

-- Create view to show the values actually used in the RDF model
CREATE VIEW values_used_in_model (value_id) as
  SELECT a.triple.rdf_s_id FROM articles_rdf_data a UNION
  SELECT a.triple.rdf_p_id FROM articles_rdf_data a UNION
  SELECT a.triple.rdf_c_id FROM articles_rdf_data a UNION
  SELECT a.triple.rdf_o_id FROM articles_rdf_data a;

View created.

-- Create views to show triples in the model
CREATE VIEW triples_in_app_table as
  SELECT a.triple.GET_SUBJECT() AS s, a.triple.GET_PROPERTY() AS p,
a.triple.GET_OBJ_VALUE() AS o
    FROM articles_rdf_data a;
```

```
View created.

CREATE VIEW triples_in_rdf_model as
  SELECT s, p, o FROM TABLE ( SEM_MATCH('{?s ?p ?o}', SEM_MODELS('articles'), null,
null, null ));

View created.

--
-- Content before deletion
--

-- Values in mdsys.RDF_VALUE$
CREATE TABLE values_before_deletion as select value_id from mdsys.rdf_value$;

Table created.

-- Values used in the RDF model
CREATE TABLE used_values_before_deletion as
  SELECT * FROM values_used_in_model;

Table created.

-- Content of RDF model
CREATE TABLE atab_triples_before_deletion
  as select * from triples_in_app_table;

Table created.

CREATE TABLE model_triples_before_deletion
  as select * from triples_in_rdf_model;

Table created.

-- Delete some triples so that some of the values become unused
DELETE FROM articles_rdf_data a
   WHERE a.triple.GET_PROPERTY() = '<http://purl.org/dc/elements/1.1/title>'
     OR a.triple.GET_SUBJECT() = '<http://nature.example.com/Article1>';

5 rows deleted.

-- Content of RDF model after deletion
CREATE TABLE atab_triples_after_deletion
  as select * from triples_in_app_table;

Table created.

CREATE TABLE model_triples_after_deletion
  as select * from triples_in_rdf_model;

Table created.

-- Values that became unused in the RDF model
SELECT * from used_values_before_deletion
  MINUS
  SELECT * FROM values_used_in_model;

VALUE_ID
--------------------
 1399113999628774496
 4597469165946334122
```

```
 6345024408674005890
 7299961478807817799
 7995347759607176041

-- RDF_VALUE$ content, however, is unchanged
SELECT value_id from values_before_deletion
  MINUS
  select value_id from mdsys.rdf_value$;

no rows selected

-- Now purge the values from RDF_VALUE$ (requires that MDSYS has
-- SELECT privilege on *all* the app tables in the semantic network)
EXECUTE sem_apis.privilege_on_app_tables;

PL/SQL procedure successfully completed.

EXECUTE sem_apis.purge_unused_values;

PL/SQL procedure successfully completed.

-- RDF_VALUE$ content is NOW changed due to the purge of unused values
SELECT value_id from values_before_deletion
  MINUS
  select value_id from mdsys.rdf_value$;

VALUE_ID
--------------------
 1399113999628774496
 4597469165946334122
 6345024408674005890
 7299961478807817799
 7995347759607176041

-- Content of RDF model after purge
CREATE TABLE atab_triples_after_purge
  as select * from triples_in_app_table;

Table created.

CREATE TABLE model_triples_after_purge
  as select * from triples_in_rdf_model;

Table created.

-- Compare triples present before purging of values and after purging
SELECT * from atab_triples_after_deletion
  MINUS
  SELECT * FROM atab_triples_after_purge;

no rows selected

SELECT * from model_triples_after_deletion
  MINUS
  SELECT * FROM model_triples_after_purge;

no rows selected
```

1.8 Using Semantic Network Indexes

Semantic network indexes are nonunique B-tree indexes that you can add, alter, and drop for use with models and entailments in a semantic network. You can use such indexes to tune the performance of SEM_MATCH queries on the models and entailments in the network. As with any indexes, semantic network indexes enable index-based access that suits your query workload. This can lead to substantial performance benefits, such as in the following example scenarios:

- If your graph pattern is `'{<John> ?p <Mary>}'`, you may want to have a usable `'CSP'` or `'SCP'` index for the target model or models and on the corresponding entailment, if used in the query.
- If your graph pattern is `'{?x <talksTo> ?y . ?z ?p ?y}'`, you may want to have a usable semantic network index on the relevant model or models and entailment, with `C` as the leading key (for example, `'C'` or `'CPS'`).

However, using semantic network indexes can affect overall performance by increasing the time required for DML, load, and inference operations.

You can create and manage semantic network indexes using the following subprograms:

- SEM_APIS.ADD_SEM_INDEX (page 11-2)
- SEM_APIS.ALTER_SEM_INDEX_ON_MODEL (page 11-6)
- SEM_APIS.ALTER_SEM_INDEX_ON_ENTAILMENT (page 11-5)
- SEM_APIS.DROP_SEM_INDEX (page 11-38)

All of these subprograms have an `index_code` parameter, which can contain any sequence of the following letters (without repetition): `P`, `C`, `S`, `G`, `M`. These letters used in the index_code correspond to the following columns in the SEMM_* and SEMI_* views: P_VALUE_ID, CANON_END_NODE_ID, START_NODE_ID, G_ID, and MODEL_ID.

The SEM_APIS.ADD_SEM_INDEX (page 11-2) procedure creates a semantic network index that results in creation of a nonunique B-tree index in UNUSABLE status for each of the existing models and entailments. The name of the index is RDF_LNK_<*index_code*>_IDX and the index is owned by MDSYS. This operation is allowed only if the invoker has DBA role. The following example shows creation of the `PSCGM` index with the following key: <P_VALUE_ID, START_NODE_ID, CANON_END_NODE_ID, G_ID, MODEL_ID>.

```
EXECUTE SEM_APIS.ADD_SEM_INDEX('PSCGM');
```

After you create a semantic network index, each of the corresponding nonunique B-tree indexes is in the UNUSABLE status, because making it usable can cause significant time and resources to be used, and because subsequent index maintenance operations might involve performance costs that you do not want to incur. You can make a semantic network index usable or unusable for specific models or entailments that you own by calling the SEM_APIS.ALTER_SEM_INDEX_ON_MODEL (page 11-6) and SEM_APIS.ALTER_SEM_INDEX_ON_ENTAILMENT (page 11-5) procedures and specifying `'REBUILD'` or `'UNUSABLE'` as the `command` parameter. Thus, you can experiment by making different semantic network indexes usable and unusable, and checking for any differences in performance. For example, the following statement makes the `PSCGM` index usable for the `FAMILY` model:

```
EXECUTE SEM_APIS.ALTER_SEM_INDEX_ON_MODEL('FAMILY','PSCGM','REBUILD');
```

Also note the following:

- Independent of any semantic network indexes that you create, when a semantic network is created, one of the indexes that is automatically created is an index that you can manage by referring to the `index_code` as `'PSCGM'` when you call the subprograms mentioned in this section.
- When you create a new model or a new entailment, a new nonunique B-tree index is created for each of the semantic network indexes, and each such B-tree index is in the USABLE status.
- Including the MODEL_ID column in a semantic network index key (by including 'M' in the `index_code` value) may improve query performance. This is particularly relevant when virtual models are used.

1.8.1 MDSYS.SEM_NETWORK_INDEX_INFO View

Information about all network indexes on models and entailments is maintained in the MDSYS.SEM_NETWORK_INDEX_INFO view, which includes (a partial list) the columns shown in Table 1-16 (page 1-87) and one row for each network index.

Table 1-16 MDSYS.SEM_NETWORK_INDEX_INFO View Columns (Partial List)

Column Name	Data Type	Description
NAME	VARCHAR2(30)	Name of the RDF model or entailment
TYPE	VARCHAR2(10)	Type of object on which the index is built: `MODEL`, `ENTAILMENT`, or `NETWORK`
ID	NUMBER	ID number for the model or entailment, or zero (0) for an index on the network
INDEX_CODE	VARCHAR2(25)	Code for the index (for example, `PSCGM`).
INDEX_NAME	VARCHAR2(30)	Name of the index (for example, `RDF_LNK_PSCGM_IDX`)
LAST_REFRESH	TIMESTAMP(6) WITH TIME ZONE	Timestamp for the last time this content was refreshed

In addition to the columns listed in Table 1-16 (page 1-87), the MDSYS.SEM_NETWORK_INDEX_INFO view contains columns from the ALL_INDEXES and ALL_IND_PARTITIONS views (both described in *Oracle Database Reference*), including:

- From the ALL_INDEXES view: UNIQUENESS, COMPRESSION, PREFIX_LENGTH
- From the ALL_IND_PARTITIONS view: STATUS, TABLESPACE_NAME, BLEVEL, LEAF_BLOCKS, NUM_ROWS, DISTINCT_KEYS, AVG_LEAF_BLOCKS_PER_KEY, AVG_DATA_BLOCKS_PER_KEY, CLUSTERING_FACTOR, SAMPLE_SIZE, LAST_ANALYZED

Note that the information in the MDSYS.SEM_NETWORK_INDEX_INFO view may sometimes be stale. You can refresh this information by using the SEM_APIS.REFRESH_SEM_NETWORK_INDEX_INFO (page 11-65) procedure.

1.9 Using Data Type Indexes

Data type indexes are indexes on the values of typed literals stored in a semantic network. These indexes may significantly improve the performance of SEM_MATCH queries involving certain types of FILTER expressions. For example, a data type index on `xsd:dateTime` literals may speed up evaluation of the filter `(?x < "1929-11-16T13:45:00Z"^^xsd:dateTime)`. Indexes can be created for several data types, which are listed in Table 1-17 (page 1-88).

Table 1-17 Data Types for Data Type Indexing

Data Type URI	Oracle Type	Index Type
http://www.w3.org/2001/XMLSchema#decimal	NUMBER	Non-unique B-tree (creates a single index for all xsd numeric types, including `xsd:float`, `xsd:double`, and `xsd:decimal` and all of its subtypes)
http://www.w3.org/2001/XMLSchema#string	VARCHAR2	Non-unique B-tree (creates a single index for `xsd:string` typed literals and plain literals)
http://www.w3.org/2001/XMLSchema#time	TIMESTAMP WITH TIMEZONE	Non-unique B-tree
http://www.w3.org/2001/XMLSchema#date	TIMESTAMP WITH TIMEZONE	Non-unique B-tree
http://www.w3.org/2001/XMLSchema#dateTime	TIMESTAMP WITH TIMEZONE	Non-unique B-tree
http://xmlns.oracle.com/rdf/text	(Not applicable)	CTXSYS.CONTEXT
http://xmlns.oracle.com/rdf/geo/WKTLiteral	SDO_GEOMETRY	MDSYS.SPATIAL_INDEX
http://www.opengis.net/geosparql#wktLiteral	SDO_GEOMETRY	MDSYS.SPATIAL_INDEX

The suitability of data type indexes depends on your query workload. Data type indexes on `xsd` data types can be used for filters that compare a variable with a constant value, and are particularly useful when queries have an unselective graph pattern with a very selective filter condition. Appropriate data type indexes are required for queries with spatial or text filters.

While data type indexes improve query performance, overhead from incremental index maintenance can degrade the performance of DML and bulk load operations on the semantic network. For bulk load operations, it may often be faster to drop data type indexes, perform the bulk load, and then re-create the data type indexes.

You can add, alter, and drop data type indexes using the following procedures, which are described in SEM_APIS Package Subprograms (page 11-1):

- SEM_APIS.ADD_DATATYPE_INDEX (page 11-1)
- SEM_APIS.ALTER_DATATYPE_INDEX (page 11-3)
- SEM_APIS.DROP_DATATYPE_INDEX (page 11-35)

Information about existing data type indexes is maintained in the MDSYS.SEM_DTYPE_INDEX_INFO view, which has the columns shown in Table 1-18 (page 1-89) and one row for each data type index.

Table 1-18 MDSYS.SEM_DTYPE_INDEX_INFO View Columns

Column Name	Data Type	Description
DATATYPE	VARCHAR2(51)	Data type URI
INDEX_NAME	VARCHAR2(30)	Name of the index
STATUS	VARCHAR2(8)	Status of the index: `USABLE` or `UNUSABLE`
TABLESPACE_NAME	VARCHAR2(30)	Tablespace for the index

You can use the `HINT0` hint to ensure that data type indexes are used during query evaluation, as shown in Example 1-88 (page 1-89), which finds all grandfathers who were born before November 16, 1929.

Example 1-88 Using HINT0 to Ensure Use of Data Type Index

```
SELECT x, y
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male . ?x :birthDate ?bd
       FILTER (?bd <= "1929-11-15T23:59:59Z"^^xsd:dateTime) }',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null, null,
    'HINT0={ LEADING(?bd) INDEX(?bd rdf_v$dateTime_idx) }
             FAST_DATE_FILTER=T' ));
```

1.10 Managing Statistics for Semantic Models and the Semantic Network

Statistics are critical to the performance of SPARQL queries and OWL inference against semantic data stored in an Oracle database. Oracle Database Release 11*g* introduced SEM_APIS.ANALYZE_MODEL (page 11-9), SEM_APIS.ANALYZE_ENTAILMENT (page 11-7), and SEM_PERF.GATHER_STATS (page 13-3) to analyze semantic data and keep statistics up to date. These APIs are straightforward to use and they are targeted at regular users who may not care about the internal details about table and partition statistics.

You can export, import, set, and delete model and entailment statistics, and can export, import, and delete network statistics, using the following subprograms:

- SEM_APIS.DELETE_ENTAILMENT_STATS (page 11-33)
- SEM_APIS.DELETE_MODEL_STATS (page 11-33)
- SEM_APIS.EXPORT_ENTAILMENT_STATS (page 11-46)
- SEM_APIS.EXPORT_MODEL_STATS (page 11-46)
- SEM_APIS.IMPORT_ENTAILMENT_STATS (page 11-57)
- SEM_APIS.IMPORT_MODEL_STATS (page 11-58)
- SEM_APIS.SET_ENTAILMENT_STATS (page 11-68)
- SEM_APIS.SET_MODEL_STATS (page 11-68)
- SEM_PERF.DELETE_NETWORK_STATS (page 13-1)
- SEM_PERF.EXPORT_NETWORK_STATS (page 13-2)
- SEM_PERF.IMPORT_NETWORK_STATS (page 13-4)

This section contains the following topics related to managing statistics for semantic models and the semantic network:

- Saving Statistics at a Model Level (page 1-90)
- Restoring Statistics at a Model Level (page 1-91)
- Saving Statistics at the Network Level (page 1-91)
- Restoring Statistics at the Network Level (page 1-92)
- Setting Statistics at a Model Level (page 1-92)
- Deleting Statistics at a Model Level (page 1-92)

1.10.1 Saving Statistics at a Model Level

If queries and inference against an existing model are executed efficiently, as the owner of the model, you can save the statistics of the existing model.

```
-- Login as the model owner (for example, SCOTT)
-- Create a stats table. This is required.
execute dbms_stats.create_stat_table('scott','rdf_stat_tab');

-- You must grant access to MDSYS
SQL> grant select, insert, delete, update on scott.rdf_stat_tab to MDSYS;

-- Now export the statistics of model TEST
execute sdo_rdf.export_model_stats('TEST','rdf_stat_tab',
'model_stat_saved_on_AUG_10', true, 'SCOTT', 'OBJECT_STATS');
```

You can also save the statistics of an entailment (entailed graph) by using SEM_APIS.EXPORT_ENTAILMENT_STATS (page 11-46).

```
execute
sem_apis.create_entailment('test_inf',sem_models('test'),sem_rulebases('owl2rl'),
0,null);
PL/SQL procedure successfully completed.
```

```
execute sem_apis.export_entailment_stats('TEST_INF','rdf_stat_tab',
'inf_stat_saved_on_AUG_10', true, 'SCOTT', 'OBJECT_STATS');
```

1.10.2 Restoring Statistics at a Model Level

As the owner of a model, can restore the statistics that were previously saved with SEM_APIS.EXPORT_MODEL_STATS (page 11-46). This may be necessary if updates have been applied to this model and statistics have been re-collected. A change in statistics might cause a plan change to existing SPARQL queries, and if such a plan change is undesirable, then an old set of statistics can be restored.

```
execute sem_apis.import_model_stats('TEST','rdf_stat_tab',
'model_stat_saved_on_AUG_10', true, 'SCOTT', false, true, 'OBJECT_STATS');
```

You can also restore the statistics of an entailment (entailed graph) by using SEM_APIS.IMPORT_ENTAILMENT_STATS (page 11-57).

```
execute sem_apis.import_entailment_stats('TEST','rdf_stat_tab',
'inf_stat_saved_on_AUG_10', true, 'SCOTT', false, true, 'OBJECT_STATS');
```

1.10.3 Saving Statistics at the Network Level

You can save statistics at the network level.

```
-- First, create a user RDF_ADMIN and assign access to package SEM_PERF to RDF_ADMIN
--
-- As SYS
--
create user RDF_ADMIN identified by RDF_ADMIN;

grant connect, resource, unlimited tablespace to RDF_ADMIN;

grant execute on sem_perf to RDF_ADMIN;

conn RDF_ADMIN/<password>

execute dbms_stats.create_stat_table('RDF_ADMIN','rdf_stat_tab');
grant select, insert, delete, update on RDF_ADMIN.rdf_stat_tab to MDSYS;

--
-- This API call will save the statistics of both MDSYS.RDF_VALUE$ table
-- and MDSYS.RDF_LINK$
--
execute sem_perf.export_network_stats('rdf_stat_tab', 'NETWORK_ALL_saved_on_Aug_10',
true, 'RDF_ADMIN', 'OBJECT_STATS');

--
-- Alternatively, you can save statistics of the MDSYS.RDF_VALUE$ table
--
execute sem_perf.export_network_stats('rdf_stat_tab',
'NETWORK_VALUE_TAB_saved_on_Aug_10', true, 'RDF_ADMIN', 'OBJECT_STATS', options=>
mdsys.sdo_rdf.VALUE_TAB_ONLY);

--
-- Or, you can save statistics of the MDSYS.RDF_LINK$ table
--
execute sem_perf.export_network_stats('rdf_stat_tab',
'NETWORK_LINK_TAB_saved_on_Aug_10', true, 'RDF_ADMIN', 'OBJECT_STATS', options=>
mdsys.sdo_rdf.LINK_TAB_ONLY);
```

1.10.4 Restoring Statistics at the Network Level

The privileged user from Saving Statistics at the Network Level (page 1-91) can restore the network level statistics using SEM_PERF.IMPORT_NETWORK_STATS (page 13-4).

```
conn RDF_ADMIN/<password>

execute sem_perf.import_network_stats('rdf_stat_tab', 'NETWORK_ALL_saved_on_Aug_10',
true, 'RDF_ADMIN', false, true, 'OBJECT_STATS');
```

1.10.5 Setting Statistics at a Model Level

As the owner of a model, you can manually adjust the statistics for this model. (However, before you adjust statistics, you should save the statistics first so that they can be restored if necessary.) The following example sets two metrics: number of rows and number of blocks for the model.

```
execute sem_apis.set_model_stats('TEST', numrows=>10,
numblks=>1,no_invalidate=>false);
```

You can also set the statistics for the entailment by using SEM_APIS.SET_ENTAILMENT_STATS (page 11-68).

```
execute sem_apis.set_entailment_stats('TEST_INF', numrows=>10,
numblks=>1,no_invalidate=>false);
```

1.10.6 Deleting Statistics at a Model Level

Removing statistics can also have an impact on execution plans. As owner of a model, you can remove the statistics for the model.

```
execute sem_apis.delete_model_stats('TEST', no_invalidate=> false);
```

You can also remove the statistics for the entailment by using SEM_APIS.DELETE_ENTAILMENT_STATS (page 11-33). (However, before you remove statistics of a model or an entailment, you should save the statistics first so that they can be restored if necessary.)

```
execute sem_apis.delete_entailment_stats('TEST_INF', no_invalidate=> false);
```

1.11 Quick Start for Using Semantic Data

To work with semantic data in an Oracle database, follow these general steps:

1. Create a tablespace for the system tables. You must be connected as a user with appropriate privileges to create the tablespace. The following example creates a tablespace named RDF_TBLSPACE:

   ```
   CREATE TABLESPACE rdf_tblspace
    DATAFILE '/oradata/orcl/rdf_tblspace.dat' SIZE 1024M REUSE
    AUTOEXTEND ON NEXT 256M MAXSIZE UNLIMITED
    SEGMENT SPACE MANAGEMENT AUTO;
   ```

2. Create a semantic data network.

   Creating a semantic data network adds semantic data support to an Oracle database. You must create a semantic data network as a user with DBA privileges,

specifying a valid tablespace with adequate space. Create the network only once for an Oracle database.

The following example creates a semantic data network using a tablespace named RDF_TBLSPACE (which must already exist):

```
EXECUTE SEM_APIS.CREATE_SEM_NETWORK('rdf_tblspace');
```

3. Connect as the database user under whose schema you will store your semantic data; do not perform the following steps while connected as SYS, SYSTEM, or MDSYS.

4. Create a table to store references to the semantic data. (You do not need to be connected as a user with DBA privileges for this step and the remaining steps.)

   This table must contain a column of type SDO_RDF_TRIPLE_S, which will contain references to all data associated with a single model.

   The following example creates a table named ARTICLES_RDF_DATA:

   ```
   CREATE TABLE articles_rdf_data (id NUMBER, triple SDO_RDF_TRIPLE_S);
   ```

5. Create a model.

   When you create a model, you specify the model name, the table to hold references to semantic data for the model, and the column of type SDO_RDF_TRIPLE_S in that table.

   The following command creates a model named ARTICLES, which will use the table created in the preceding step.

   ```
   EXECUTE SEM_APIS.CREATE_SEM_MODEL('articles', 'articles_rdf_data', 'triple');
   ```

6. Where possible, create Oracle database indexes on conditions that will be specified in the WHERE clause of SELECT statements, to provide better performance for direct queries against the application table's SDO_RDF_TRIPLE_S column. (These indexes are not relevant if the SEM_MATCH table function is being used.) The following example creates such indexes:

   ```
   -- Create indexes on the subjects, properties, and objects
   -- in the ARTICLES_RDF_DATA table.
   CREATE INDEX articles_sub_idx ON articles_rdf_data (triple.GET_SUBJECT());
   CREATE INDEX articles_prop_idx ON articles_rdf_data (triple.GET_PROPERTY());
   CREATE INDEX articles_obj_idx ON articles_rdf_data (TO_CHAR(triple.GET_OBJECT()));
   ```

After you create the model, you can insert triples into the table, as shown in the examples in Semantic Data Examples (PL/SQL and Java) (page 1-93).

1.12 Semantic Data Examples (PL/SQL and Java)

This section contains the following PL/SQL examples:

- Example: Journal Article Information (page 1-94)
- Example: Family Information (page 1-95)

In addition to the examples in this guide, see the sample code at http://www.oracle.com/technetwork/indexes/samplecode/semantic-sample-522114.html.

1.12.1 Example: Journal Article Information

This section presents a simplified PL/SQL example of model for statements about journal articles. Example 1-89 (page 1-94) contains descriptive comments, refers to concepts that are explained in this chapter, and uses functions and procedures documented in SEM_APIS Package Subprograms (page 11-1).

Example 1-89 Using a Model for Journal Article Information

```
-- Basic steps:
-- After you have connected as a privileged user and called
-- SEM_APIS.CREATE_SEM_NETWORK to add RDF support,
-- connect as a regular database user and do the following.
-- 1. For each desired model, create a table to hold its data.
-- 2. For each model, create a model (SEM_APIS.CREATE_SEM_MODEL).
-- 3. For each table to hold semantic data, insert data into the table.
-- 4. Use various subprograms and constructors.

-- Create the table to hold data for the model.
CREATE TABLE articles_rdf_data (id NUMBER, triple SDO_RDF_TRIPLE_S);

-- Create the model.
EXECUTE SEM_APIS.CREATE_SEM_MODEL('articles', 'articles_rdf_data', 'triple');

-- Information to be stored about some fictitious articles:
-- Article1, titled "All about XYZ" and written by Jane Smith, refers
--   to Article2 and Article3.
-- Article2, titled "A review of ABC" and written by Joe Bloggs,
--   refers to Article3.
-- Seven SQL statements to store the information. In each statement:
-- Each article is referred to by its complete URI The URIs in
--   this example are fictitious.
-- Each property is referred to by the URL for its definition, as
--   created by the Dublin Core Metadata Initiative.

-- Insert rows into the table.

-- Article1 has the title "All about XYZ".
INSERT INTO articles_rdf_data VALUES (1,
  SDO_RDF_TRIPLE_S ('articles','<http://nature.example.com/Article1>',
    '<http://purl.org/dc/elements/1.1/title>','"All about XYZ"'));

-- Article1 was created (written) by Jane Smith.
INSERT INTO articles_rdf_data VALUES (2,
  SDO_RDF_TRIPLE_S ('articles','<http://nature.example.com/Article1>',
    '<http://purl.org/dc/elements/1.1/creator>',
    '"Jane Smith"'));

-- Article1 references (refers to) Article2.
INSERT INTO articles_rdf_data VALUES (3,
  SDO_RDF_TRIPLE_S ('articles',
    '<http://nature.example.com/Article1>',
    '<http://purl.org/dc/terms/references>',
    '<http://nature.example.com/Article2>'));

-- Article1 references (refers to) Article3.
INSERT INTO articles_rdf_data VALUES (4,
  SDO_RDF_TRIPLE_S ('articles',
    '<http://nature.example.com/Article1>',
    '<http://purl.org/dc/terms/references>',
```

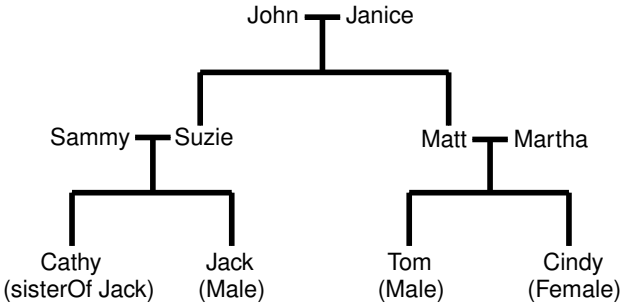
```
    '<http://nature.example.com/Article3>'));

-- Article2 has the title "A review of ABC".
INSERT INTO articles_rdf_data VALUES (5,
  SDO_RDF_TRIPLE_S ('articles',
    '<http://nature.example.com/Article2>',
    '<http://purl.org/dc/elements/1.1/title>',
    '"A review of ABC"'));

-- Article2 was created (written) by Joe Bloggs.
INSERT INTO articles_rdf_data VALUES (6,
  SDO_RDF_TRIPLE_S ('articles',
    '<http://nature.example.com/Article2>',
    '<http://purl.org/dc/elements/1.1/creator>',
    '"Joe Bloggs"'));

-- Article2 references (refers to) Article3.
INSERT INTO articles_rdf_data VALUES (7,
  SDO_RDF_TRIPLE_S ('articles',
    '<http://nature.example.com/Article2>',
    '<http://purl.org/dc/terms/references>',
    '<http://nature.example.com/Article3>'));

COMMIT;

-- Query semantic data.

SELECT SEM_APIS.GET_MODEL_ID('articles') AS model_id FROM DUAL;

SELECT SEM_APIS.GET_TRIPLE_ID(
  'articles',
  '<http://nature.example.com/Article2>',
  '<http://purl.org/dc/terms/references>',
  '<http://nature.example.com/Article3>') AS RDF_triple_id FROM DUAL;

SELECT SEM_APIS.IS_TRIPLE(
  'articles',
  '<http://nature.example.com/Article2>',
  '<http://purl.org/dc/terms/references>',
  '<http://nature.example.com/Article3>') AS is_triple FROM DUAL;

-- Use SDO_RDF_TRIPLE_S member functions in queries.

SELECT a.triple.GET_TRIPLE() AS triple
  FROM articles_rdf_data a WHERE a.id = 1;
SELECT a.triple.GET_SUBJECT() AS subject
  FROM articles_rdf_data a WHERE a.id = 1;
SELECT a.triple.GET_PROPERTY() AS property
  FROM articles_rdf_data a WHERE a.id = 1;
SELECT a.triple.GET_OBJECT() AS object
  FROM articles_rdf_data a WHERE a.id = 1;
```

1.12.2 Example: Family Information

This section presents a simplified PL/SQL example of a model for statements about family tree (genealogy) information. Example 1-89 (page 1-94) contains descriptive comments, refers to concepts that are explained in this chapter, and uses functions and procedures documented in SEM_APIS Package Subprograms (page 11-1).

The family relationships in this example reflect the family tree shown in Figure 1-3 (page 1-96). This figure also shows some of the information directly stated in the example: Cathy is the sister of Jack, Jack and Tom are male, and Cindy is female.

Figure 1-3 Family Tree for RDF Example



Example 1-90 Using a Model for Family Information

```
-- Basic steps:
-- After you have connected as a privileged user and called
-- SEM_APIS.CREATE_SEM_NETWORK to enable RDF support,
-- connect as a regular database user and do the following.
-- 1. For each desired model, create a table to hold its data.
-- 2. For each model, create a model (SEM_APIS.CREATE_SEM_MODEL).
-- 3. For each table to hold semantic data, insert data into the table.
-- 4. Use various subprograms and constructors.

-- Create the table to hold data for the model.
CREATE TABLE family_rdf_data (id NUMBER, triple SDO_RDF_TRIPLE_S);

-- Create the model.
execute SEM_APIS.create_sem_model('family', 'family_rdf_data', 'triple');

-- Insert rows into the table. These express the following information:
-----------------
-- John and Janice have two children, Suzie and Matt.
-- Matt married Martha, and they have two children:
--   Tom (male, height 5.75) and Cindy (female, height 06.00).
-- Suzie married Sammy, and they have two children:
--   Cathy (height 5.8) and Jack (male, height 6).

-- Person is a class that has two subslasses: Male and Female.
-- parentOf is a property that has two subproperties: fatherOf and motherOf.
-- siblingOf is a property that has two subproperties: brotherOf and sisterOf.
-- The domain of the fatherOf and brotherOf properties is Male.
-- The domain of the motherOf and sisterOf properties is Female.
------------------------

-- John is the father of Suzie.
INSERT INTO family_rdf_data VALUES (1,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/John>',
'<http://www.example.org/family/fatherOf>',
'<http://www.example.org/family/Suzie>'));

-- John is the father of Matt.
INSERT INTO family_rdf_data VALUES (2,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/John>',
'<http://www.example.org/family/fatherOf>',
```

```
'<http://www.example.org/family/Matt>'));

-- Janice is the mother of Suzie.
INSERT INTO family_rdf_data VALUES (3,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Janice>',
'<http://www.example.org/family/motherOf>',
'<http://www.example.org/family/Suzie>'));

-- Janice is the mother of Matt.
INSERT INTO family_rdf_data VALUES (4,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Janice>',
'<http://www.example.org/family/motherOf>',
'<http://www.example.org/family/Matt>'));

-- Sammy is the father of Cathy.
INSERT INTO family_rdf_data VALUES (5,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Sammy>',
'<http://www.example.org/family/fatherOf>',
'<http://www.example.org/family/Cathy>'));

-- Sammy is the father of Jack.
INSERT INTO family_rdf_data VALUES (6,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Sammy>',
'<http://www.example.org/family/fatherOf>',
'<http://www.example.org/family/Jack>'));

-- Suzie is the mother of Cathy.
INSERT INTO family_rdf_data VALUES (7,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Suzie>',
'<http://www.example.org/family/motherOf>',
'<http://www.example.org/family/Cathy>'));

-- Suzie is the mother of Jack.
INSERT INTO family_rdf_data VALUES (8,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Suzie>',
'<http://www.example.org/family/motherOf>',
'<http://www.example.org/family/Jack>'));

-- Matt is the father of Tom.
INSERT INTO family_rdf_data VALUES (9,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Matt>',
'<http://www.example.org/family/fatherOf>',
'<http://www.example.org/family/Tom>'));

-- Matt is the father of Cindy
INSERT INTO family_rdf_data VALUES (10,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Matt>',
'<http://www.example.org/family/fatherOf>',
'<http://www.example.org/family/Cindy>'));

-- Martha is the mother of Tom.
INSERT INTO family_rdf_data VALUES (11,
SDO_RDF_TRIPLE_S('family',
```

```
'<http://www.example.org/family/Martha>',
'<http://www.example.org/family/motherOf>',
'<http://www.example.org/family/Tom>'));

-- Martha is the mother of Cindy.
INSERT INTO family_rdf_data VALUES (12,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Martha>',
'<http://www.example.org/family/motherOf>',
'<http://www.example.org/family/Cindy>'));

-- Cathy is the sister of Jack.
INSERT INTO family_rdf_data VALUES (13,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Cathy>',
'<http://www.example.org/family/sisterOf>',
'<http://www.example.org/family/Jack>'));

-- Jack is male.
INSERT INTO family_rdf_data VALUES (14,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Jack>',
'<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
'<http://www.example.org/family/Male>'));

-- Tom is male.
INSERT INTO family_rdf_data VALUES (15,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Tom>',
'<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
'<http://www.example.org/family/Male>'));

-- Cindy is female.
INSERT INTO family_rdf_data VALUES (16,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Cindy>',
'<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
'<http://www.example.org/family/Female>'));

-- Person is a class.
INSERT INTO family_rdf_data VALUES (17,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Person>',
'<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
'<http://www.w3.org/2000/01/rdf-schema#Class>'));

-- Male is a subclass of Person.
INSERT INTO family_rdf_data VALUES (18,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Male>',
'<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
'<http://www.example.org/family/Person>'));

-- Female is a subclass of Person.
INSERT INTO family_rdf_data VALUES (19,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Female>',
'<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
'<http://www.example.org/family/Person>'));

-- siblingOf is a property.
```

```
INSERT INTO family_rdf_data VALUES (20,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/siblingOf>',
'<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
'<http://www.w3.org/1999/02/22-rdf-syntax-ns#Property>'));

-- parentOf is a property.
INSERT INTO family_rdf_data VALUES (21,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/parentOf>',
'<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
'<http://www.w3.org/1999/02/22-rdf-syntax-ns#Property>'));

-- brotherOf is a subproperty of siblingOf.
INSERT INTO family_rdf_data VALUES (22,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/brotherOf>',
'<http://www.w3.org/2000/01/rdf-schema#subPropertyOf>',
'<http://www.example.org/family/siblingOf>'));

-- sisterOf is a subproperty of siblingOf.
INSERT INTO family_rdf_data VALUES (23,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/sisterOf>',
'<http://www.w3.org/2000/01/rdf-schema#subPropertyOf>',
'<http://www.example.org/family/siblingOf>'));

-- A brother is male.
INSERT INTO family_rdf_data VALUES (24,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/brotherOf>',
'<http://www.w3.org/2000/01/rdf-schema#domain>',
'<http://www.example.org/family/Male>'));

-- A sister is female.
INSERT INTO family_rdf_data VALUES (25,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/sisterOf>',
'<http://www.w3.org/2000/01/rdf-schema#domain>',
'<http://www.example.org/family/Female>'));

-- fatherOf is a subproperty of parentOf.
INSERT INTO family_rdf_data VALUES (26,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/fatherOf>',
'<http://www.w3.org/2000/01/rdf-schema#subPropertyOf>',
'<http://www.example.org/family/parentOf>'));

-- motherOf is a subproperty of parentOf.
INSERT INTO family_rdf_data VALUES (27,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/motherOf>',
'<http://www.w3.org/2000/01/rdf-schema#subPropertyOf>',
'<http://www.example.org/family/parentOf>'));

-- A father is male.
INSERT INTO family_rdf_data VALUES (28,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/fatherOf>',
'<http://www.w3.org/2000/01/rdf-schema#domain>',
'<http://www.example.org/family/Male>'));
```

```
-- A mother is female.
INSERT INTO family_rdf_data VALUES (29,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/motherOf>',
'<http://www.w3.org/2000/01/rdf-schema#domain>',
'<http://www.example.org/family/Female>'));

-- Use SET ESCAPE OFF to prevent the caret (^) from being
-- interpreted as an escape character. Two carets (^^) are
-- used to represent typed literals.
SET ESCAPE OFF;

-- Cathy's height is 5.8 (decimal).
INSERT INTO family_rdf_data VALUES (30,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Cathy>',
'<http://www.example.org/family/height>',
'"5.8"^^xsd:decimal'));

-- Jack's height is 6 (integer).
INSERT INTO family_rdf_data VALUES (31,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Jack>',
'<http://www.example.org/family/height>',
'"6"^^xsd:integer'));

-- Tom's height is 05.75 (decimal).
INSERT INTO family_rdf_data VALUES (32,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Tom>',
'<http://www.example.org/family/height>',
'"05.75"^^xsd:decimal'));

-- Cindy's height is 06.00 (decimal).
INSERT INTO family_rdf_data VALUES (33,
SDO_RDF_TRIPLE_S('family',
'<http://www.example.org/family/Cindy>',
'<http://www.example.org/family/height>',
'"06.00"^^xsd:decimal'));

COMMIT;

-- RDFS inferencing in the family model
BEGIN
  SEM_APIS.CREATE_ENTAILMENT(
    'rdfs_rix_family',
    SEM_Models('family'),
    SEM_Rulebases('RDFS'));
END;
/

-- Select all males from the family model, without inferencing.
SELECT m
  FROM TABLE(SEM_MATCH(
    '{?m rdf:type :Male}',
    SEM_Models('family'),
    null,
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

```
-- Select all males from the family model, with RDFS inferencing.
SELECT m
  FROM TABLE(SEM_MATCH(
    '{?m rdf:type :Male}',
    SEM_Models('family'),
    SDO_RDF_Rulebases('RDFS'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));

-- General inferencing in the family model

EXECUTE SEM_APIS.CREATE_RULEBASE('family_rb');

INSERT INTO mdsys.semr_family_rb VALUES(
  'grandparent_rule',
  '(?x :parentOf ?y) (?y :parentOf ?z)',
  NULL,
  '(?x :grandParentOf ?z)',
  SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')));

COMMIT;

-- Because a new rulebase has been created, and it will be used in the
-- entailment, drop the preceding entailment and then re-create it.
EXECUTE SEM_APIS.DROP_ENTAILMENT ('rdfs_rix_family');

-- Re-create the entailment.
BEGIN
  SEM_APIS.CREATE_ENTAILMENT(
    'rdfs_rix_family',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'));
END;
/

-- Select all grandfathers and their grandchildren from the family model,
-- without inferencing. (With no inferencing, no results are returned.)
SELECT x grandfather, y grandchild
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male}',
    SEM_Models('family'),
    null,
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));

-- Select all grandfathers and their grandchildren from the family model.
-- Use inferencing from both the RDFS and family_rb rulebases.
SELECT x grandfather, y grandchild
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male}',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));

-- Set up to find grandfathers of tall (>= 6) grandchildren
-- from the family model, with RDFS inferencing and
-- inferencing using the "family_rb" rulebase.

UPDATE mdsys.semr_family_rb SET
  antecedents = '(?x :parentOf ?y) (?y :parentOf ?z) (?z :height ?h)',
```

```
  filter = '(h >= ''6'')',
  aliases = SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/'))
WHERE rule_name = 'GRANDPARENT_RULE';

-- Because the rulebase has been updated, drop the preceding entailment,
-- and then re-create it.
EXECUTE SEM_APIS.DROP_ENTAILMENT ('rdfs_rix_family');

-- Re-create the entailment.
BEGIN
  SEM_APIS.CREATE_ENTAILMENT(
    'rdfs_rix_family',
    SEM_Models('family'),
    SEM_Rulebases('RDFS','family_rb'));
END;
/

-- Find the entailment that was just created (that is, the
-- one based on the specified model and rulebases).
SELECT SEM_APIS.LOOKUP_ENTAILMENT(SEM_MODELS('family'),
  SEM_RULEBASES('RDFS','family_rb')) AS lookup_entailment FROM DUAL;

-- Select grandfathers of tall (>= 6) grandchildren, and their
-- tall grandchildren.
SELECT x grandfather, y grandchild
  FROM TABLE(SEM_MATCH(
    '{?x :grandParentOf ?y . ?x rdf:type :Male}',
    SEM_Models('family'),
    SEM_RuleBases('RDFS','family_rb'),
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null));
```

1.13 Software Naming Changes Since Release 11.1

Because the support for semantic data has been expanded beyond the original focus on RDF, the names of many software objects (PL/SQL packages, functions and procedures, system tables and views, and so on) have been changed as of Oracle Database Release 11.1. In most cases, the change is to replace the string *RDF* with *SEM*. although in some cases it may be to replace *SDO_RDF* with *SEM*.

All valid code that used the pre-Release 11.1 names will continue to work; your existing applications will not be broken. However, it is suggested that you change old applications to use new object names, and you should use the new names for any new applications. This manual will document only the new names.

Table 1-19 (page 1-102) lists the old and new names for some objects related to support for semantic technologies, in alphabetical order by old name.

Table 1-19 Semantic Technology Software Objects: Old and New Names

Old Name	New Name
RDF_ALIAS data type	SEM_ALIAS
RDF_MODEL$ view	SEM_MODEL$
RDF_RULEBASE_INFO view	SEM_RULEBASE_INFO
RDF_RULES_INDEX_DATASETS view	SEM_RULES_INDEX_DATASETS

Table 1-19 (Cont.) Semantic Technology Software Objects: Old and New Names

Old Name	New Name
RDF_RULES_INDEX_INFO view	SEM_RULES_INDEX_INFO
RDFI_*rules-index-name* view	SEMI_*rules-index-name*
RDFM_*model-name* view	SEMM_*model-name*
RDFR_*rulebase-name* view	SEMR_*rulebase-name*
SDO_RDF package	SEM_APIS
SDO_RDF_INFERENCE package	SEM_APIS
SDO_RDF_MATCH table function	SEM_MATCH
SDO_RDF_MODELS data type	SEM_MODELS
SDO_RDF_RULEBASES data type	SEM_RULEBASES

1.14 For More Information About RDF Semantic Graph

For more information about RDF Semantic Graph support and related topics, you may find the following resources helpful:

- Oracle Spatial and Graph RDF Semantic Graph page (OTN), which includes links for downloads, technical and business white papers, a discussion forum, and other sources of information: `http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/rdfsemantic-graph-1902016.html`
- World Wide Web Consortium (W3C) *RDF Primer*: `http://www.w3.org/TR/rdf-primer/`
- World Wide Web Consortium (W3C) *OWL Web Ontology Language Reference*: `http://www.w3.org/TR/owl-ref/`

2

OWL Concepts

This chapter describes concepts related to the support for a subset of the Web Ontology Language (OWL). It builds on the information in RDF Semantic Graph Overview (page 1-1), and it assumes that you are familiar with the major concepts associated with OWL, such as ontologies, properties, and relationships. For detailed information about OWL, see the *OWL Web Ontology Language Reference* at `http://www.w3.org/TR/owl-ref/`.

This chapter contains the following major sections:

- Ontologies (page 2-1)
- Using OWL Inferencing (page 2-5)
- Using Semantic Operators to Query Relational Data (page 2-19)

2.1 Ontologies

An **ontology** is a shared conceptualization of knowledge in a particular domain. It consists of a collection of classes, properties, and optionally instances. Classes are typically related by class hierarchy (subclass/ superclass relationship). Similarly, the properties can be related by property hierarchy (subproperty/ superproperty relationship). Properties can be symmetric or transitive, or both. Properties can also have domain, ranges, and cardinality constraints specified for them.

RDFS-based ontologies only allow specification of class hierarchies, property hierarchies, `instanceOf` relationships, and a domain and a range for properties.

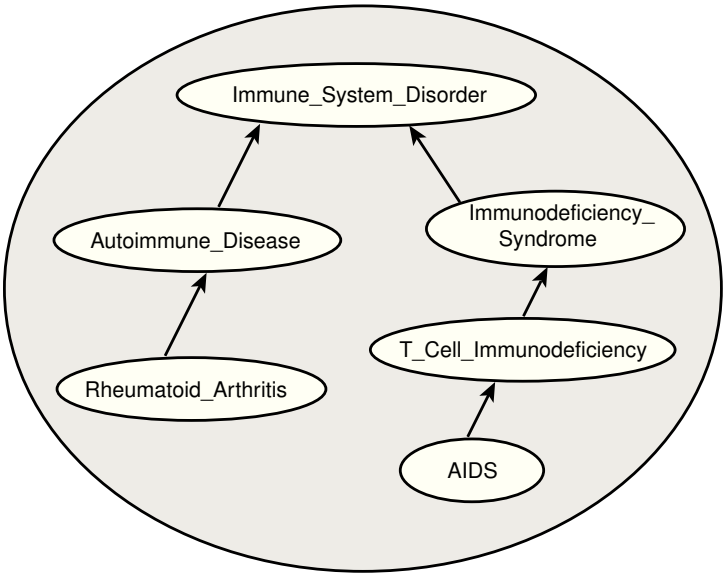
OWL ontologies build on RDFS-based ontologies by additionally allowing specification of property characteristics. OWL ontologies can be further classified as OWL-Lite, OWL-DL, and OWL Full. OWL-Lite restricts the cardinality minimum and maximum values to 0 or 1. OWL-DL relaxes this restriction by allowing minimum and maximum values. OWL Full allows instances to be also defined as a class, which is not allowed in OWL-DL and OWL-Lite ontologies.

Supported OWL Subsets (page 2-2) describes OWL capabilities that are supported and not supported with semantic data.

2.1.1 Example: Cancer Ontology

Figure 2-1 (page 2-2) shows part of a cancer ontology, which describes the classes and properties related to cancer. One requirement is to have a PATIENTS data table with a column named DIAGNOSIS, which must contain a value from the `Diseases_and_Disorders` class hierarchy.

Figure 2-1 Cancer Ontology Example



In the cancer ontology shown in Figure 2-1 (page 2-2), the diagnosis `Immune_System_Disorder` includes two subclasses, `Autoimmune_Disease` and `Immunodeficiency_Syndrome`. The `Autoimmune_Disease` diagnosis includes the subclass `Rheumatoid_Arthritis`; and the `Immunodeficiency_Syndrome` diagnosis includes the subclass `T_Cell_Immunodeficiency`, which includes the subclass `AIDS`.

The data in the PATIENTS table might include the PATIENT_ID and DIAGNOSIS column values shown in Table 2-1 (page 2-2).

Table 2-1 PATIENTS Table Example Data

PATIENT_ID	DIAGNOSIS
1234	Rheumatoid_Arthritis
2345	Immunodeficiency_Syndrome
3456	AIDS

To query ontologies, you can use the SEM_MATCH table function (described in Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22)) or the SEM_RELATED operator and its ancillary operators (described in Using Semantic Operators to Query Relational Data (page 2-19)).

2.1.2 Supported OWL Subsets

This section describes OWL vocabulary subsets that are supported.

Oracle Database supports the RDFS++, OWLSIF, and OWLPrime vocabularies, which have increasing expressivity, as well as OWL 2 RL. Each supported vocabulary has a corresponding rulebase; however, these rulebases do not need to be populated because the underlying entailment rules of these three vocabularies are internally implemented. The supported vocabularies are as follows:

- RDFS++: A minimal extension to RDFS; which is RDFS plus `owl:sameAs` and `owl:InverseFunctionalProperty`.

- OWLSIF: OWL with IF Semantic, with the vocabulary and semantics proposed for pD* semantics in *Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary*, by H.J. Horst, Journal of Web Semantics 3, 2 (2005), 79–115.
- OWLPrime: The following OWL capabilities:
  - Basics: class, subclass, property, subproperty, domain, range, type
  - Property characteristics: transitive, symmetric, functional, inverse functional, inverse
  - Class comparisons: equivalence, disjointness
  - Property comparisons: equivalence
  - Individual comparisons: same, different
  - Class expressions: complement
  - Property restrictions: `hasValue`, `someValuesFrom`, `allValuesFrom`

    As with pD*, the supported semantics for these value restrictions are only intensional (IF semantics).
- OWL 2 RL: Described in the "OWL 2 RL" section of the W3C *OWL 2 Web Ontology Language Profiles* recommendation (`http://www.w3.org/TR/owl2-profiles/#OWL_2_RL`) as: "The OWL 2 RL profile is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It is designed to accommodate both OWL 2 applications that can trade the full expressivity of the language for efficiency, and RDF(S) applications that need some added expressivity from OWL 2."

  The system-defined rulebase `OWL2RL` supports all the standard production rules defined for OWL 2 RL. As with `OWLPRIME`, users will not see any rules in this OWL2RL rulebase. The rulebase `OWL2RL` will be created automatically if it does not already exist.

  The following code excerpt uses the OWL2RL rulebase:

```
CREATE TABLE m1_tpl (triple SDO_RDF_TRIPLE_S) COMPRESS;
EXECUTE sem_apis.create_sem_model('m1','m1_tpl','triple');
-- Insert data into model M1. Details omitted
...
-- Now run inference using the OWL2RL rulebase
EXECUTE
sem_apis.create_entailment('m1_inf',sem_models('m1'),sem_rulebases('owl2rl'));
```

  Note that inference-related optimizations, such as parallel inference and RAW8, are all applicable when the `OWL2RL` rulebase is used.
- OWL 2 EL: Described in the "OWL 2 EL" section of the W3C *OWL 2 Web Ontology Language Profiles* recommendation (`http://www.w3.org/TR/owl2-profiles/#OWL_2_EL`) as: "The OWL 2 EL profile is designed as a subset of OWL 2 that
  - is particularly suitable for applications employing ontologies that define very large numbers of classes and/or properties,
  - captures the expressive power used by many such ontologies, and

– for which ontology consistency, class expression subsumption, and instance checking can be decided in polynomial time."

A prime example of OWL 2 EL ontology is the biomedical ontology SNOMED Clinical Terms (SNOMED CT). For information about SNOMED CT, see: http://www.ihtsdo.org/snomed-ct/

The system-defined rulebase OWL2EL supports the EL syntax.

As with OWLPRIME and OWL2RL, users will not see any rules in this OWL2EL rulebase, and the OWL2EL rulebase will be created automatically if it does not already exist.

The following code excerpt uses the OWL2EL rulebase against the well known SNOMED ontology:

```
CREATE TABLE snomed_tpl (triple SDO_RDF_TRIPLE_S) COMPRESS;
EXECUTE sem_apis.create_sem_model('snomed','snomed_tpl','triple') compress;
-- Insert data into model SNOMED. Details omitted
...
-- Now run inference using the OWL2EL rulebase
EXECUTE
sem_apis.create_entailment('snomed_inf',sem_models('snomed'),sem_rulebases('owl2e
l'));
```

Note that the OWL2EL rulebase support does not include reflexive object properties (ReflexiveObjectProperty) simply because a reflexive object property will link every individual with itself, which would probably cause an unnecessary and costly expansion of the inference graph.

Table 2-2 (page 2-4) lists the RDFS/OWL vocabulary constructs included in each supported rulebase.

Table 2-2 RDFS/OWL Vocabulary Constructs Included in Each Supported Rulebase

Rulebase Name	RDFS/OWL Constructs Included
RDFS++	all RDFS vocabulary constructs owl:InverseFunctionalProperty owl:sameAs
OWLSIF	all RDFS vocabulary constructs owl:FunctionalProperty owl:InverseFunctionalProperty owl:SymmetricProperty owl:TransitiveProperty owl:sameAs owl:inverseOf owl:equivalentClass owl:equivalentProperty owl:hasValue owl:someValuesFrom owl:allValuesFrom

Table 2-2 (Cont.) RDFS/OWL Vocabulary Constructs Included in Each Supported Rulebase

Rulebase Name	RDFS/OWL Constructs Included
OWLPrime	rdfs:subClassOf rdfs:subPropertyOf rdfs:domain rdfs:range owl:FunctionalProperty owl:InverseFunctionalProperty owl:SymmetricProperty owl:TransitiveProperty owl:sameAs owl:inverseOf owl:equivalentClass owl:equivalentProperty owl:hasValue owl:someValuesFrom owl:allValuesFrom owl:differentFrom owl:disjointWith owl:complementOf
OWL2RL	(As described in http://www.w3.org/TR/owl2-profiles/#OWL_2_RL)
OWL2EL	(As described in http://www.w3.org/TR/owl2-profiles/#OWL_2_EL)

2.2 Using OWL Inferencing

You can use entailment rules to perform native OWL inferencing. This section creates a simple ontology, performs native inferencing, and illustrates some more advanced features.

2.2.1 Creating a Simple OWL Ontology

Example 2-1 (page 2-5) creates a simple OWL ontology, inserts one statement that two URIs refer to the same entity, and performs a query using the SEM_MATCH table function.

Example 2-1 Creating a Simple OWL Ontology

```
SQL> CREATE TABLE owltst(id number, triple sdo_rdf_triple_s);
Table created.

SQL> EXECUTE sem_apis.create_sem_model('owltst','owltst','triple');
PL/SQL procedure successfully completed.

SQL> INSERT INTO owltst VALUES (1, sdo_rdf_triple_s('owltst',
       'http://example.com/name/John', 'http://www.w3.org/2002/07/owl#sameAs',
       'http://example.com/name/JohnQ'));
1 row created.
```

```
SQL> commit;

SQL> -- Use SEM_MATCH to perform a simple query.
SQL> select s,p,o from table(SEM_MATCH('(?s ?p  ?o)', SEM_Models('OWLTST'),
           null,  null, null ));
```

2.2.2 Performing Native OWL inferencing

Example 2-2 (page 2-6) calls the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure. You do not need to create the rulebase and add rules to it, because the OWL rules are already built into the RDF Semantic Graph inferencing engine.

Example 2-2 Performing Native OWL Inferencing

```
SQL> -- Invoke the following command to run native OWL inferencing that
SQL> -- understands the vocabulary defined in the preceding section.
SQL>
SQL>  EXECUTE sem_apis.create_entailment('owltst_idx', sem_models('owltst'),
sem_rulebases('OWLPRIME'));
PL/SQL procedure successfully completed.

SQL> -- The following view is generated to represent the entailed graph (rules
index).
SQL> desc mdsys.semi_owltst_idx;

SQL> -- Run the preceding query with an additional rulebase parameter to list
SQL> -- the original graph plus the inferred triples.
SQL> SELECT s,p,o FROM table(SEM_MATCH('(?s ?p  ?o)', SEM_MODELS('OWLTST'),
           SEM_RULEBASES('OWLPRIME'),  null, null ));
```

2.2.3 Performing OWL and User-Defined Rules Inferencing

Example 2-3 (page 2-7) creates a user-defined rulebase, inserts a simplified `uncleOf` rule (stating that the brother of one's father is one's uncle), and calls the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure.

For performance, the inference engine by default executes each user rule without checking the syntax legality of inferred triples (for example, literal value as a subject, blank node as a predicate) until after the last round of entailment. After completing the last entailment round, the inference engine removes all syntactically illegal triples without throwing any errors for these triples. However, because triples with illegal syntax may exist during multiple rounds of inference, rules can use these triples as part of their antecedents. For example, consider the following user-defined rules:

- Rule 1:

  ```
  (?s :account ?y)
  (?s :country :Spain) --> (?y rdf:type :SpanishAccount)
  ```

- Rule 2:

  ```
  (?s :account ?y)
  (?y rdf:type :SpanishAccount) --> (?s :language "es_ES")
  ```

Rule 1 finds all Spanish users and designates their accounts as Spanish accounts. Rule 2 sets the language for all users with Spanish accounts to `es_ES` (Spanish). Consider the following data, displayed in Turtle format:

```
:Juan      :account "123ABC4Z"
           :country :Spain
```

```
:Alejandro :account "5678DEF9Y"
           :country :Spain
```

Applying Rule 1 and Rule 2 produces the following inferred triples:

```
(:Juan      :language "es_ES")
(:Alejandro :language "es_ES")
```

Note there are no triples specifying which accounts are of type `:SpanishAccount`. The user-defined rules infer those triples during creation of the entailment, but the inference engine removes them after the last round of inference because they contain illegal syntax. The accounts are the literal values, which cannot be used as subjects in an RDF triple.

To force the checking of syntax legality of inferred triples, add the `/*+ ENABLE_SYNTAX_CHECKING */` optimizer hint to the beginning of the rule's FILTER expression. Forcing syntax checking for a rule can result in a performance penalty and will throw an exception for any syntactically illegal triples. The following example, similar to Rule 1, forces syntax checking. (In addition, merely to illustrate the use of a filter expression, the example restricts accounts to those that do not end with the letter 'Z'.)

```
INSERT INTO mdsys.semr_user_rulebase VALUES (
  'spanish_account_rule',
  '(?s <http://example.com/account> ?y)(?y <http://example.com/account> <http://
example.com/Spain>)',
  '/*+ ENABLE_SYNTAX_CHECKING */ y not like ''%Z'' ',
  '(?y <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://example.com/
SpanishAccount>)',
  NULL
);
```

Example 2-3 Performing OWL and User-Defined Rules Inferencing

```
SQL> -- First, insert the following assertions.

SQL> INSERT INTO owltst VALUES (1, sdo_rdf_triple_s('owltst',
       'http://example.com/name/John', 'http://example.com/rel/fatherOf',
       'http://example.com/name/Mary'));


SQL> INSERT INTO owltst VALUES (1, sdo_rdf_triple_s('owltst',
       'http://example.com/name/Jack', 'http://example.com/rel/brotherOf',
       'http://example.com/name/John'));

SQL> -- Create a user-defined rulebase.

SQL> EXECUTE sem_apis.create_rulebase('user_rulebase');

SQL> -- Insert a simple "uncle" rule.

SQL> INSERT INTO mdsys.semr_user_rulebase VALUES ('uncle_rule',
'(?x <http://example.com/rel/brotherOf> ?y)(?y <http://example.com/rel/fatherOf> ?
z)',
NULL, '(?x <http://example.com/rel/uncleOf> ?z)', null);

SQL>  -- In the following statement, 'USER_RULES=T' is required, to
SQL> --  include the original graph plus the inferred triples.
SQL> EXECUTE sem_apis.create_entailment('owltst2_idx', sem_models('owltst'),
          sem_rulebases('OWLPRIME','USER_RULEBASE'),
          SEM_APIS.REACH_CLOSURE, null, 'USER_RULES=T');
```

```
SQL> -- In the result of the following query, :Jack :uncleOf :Mary is inferred.
SQL> SELECT s,p,o FROM table(SEM_MATCH('(?s ?p  ?o)',
           SEM_MODELS('OWLTST'),
           SEM_RULEBASES('OWLPRIME','USER_RULEBASE'),  null, null ));
```

2.2.4 Generating OWL inferencing Proofs

OWL inference can be complex, depending on the size of the ontology, the actual vocabulary (set of language constructs) used, and the interactions among those language constructs. To enable you to find out how a triple is derived, you can use proof generation during inference. (Proof generation does require additional CPU time and disk resources.)

To generate the information required for proof, specify `PROOF=T` in the call to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, as shown in the following example:

```
EXECUTE sem_apis.create_entailment('owltst_idx', sem_models('owltst'), -
  sem_rulebases('owlprime'), SEM_APIS.REACH_CLOSURE, 'SAM', 'PROOF=T');
```

Specifying `PROOF=T` causes a view to be created containing proof for each inferred triple. The view name is the entailment name prefixed by `MDSYS.SEMI_`. Two relevant columns in this view are LINK_ID and EXPLAIN (the proof). The following example displays the LINK_ID value and proof of each generated triple (with LINK_ID values shortened for simplicity):

```
SELECT link_id || ' generated by ' || explain as
          triple_and_its_proof FROM mdsys.semi_owltst_idx;

TRIPLE_AND_ITS_PROOF
--------------------------------------------------------------------
8_5_5_4 generated by 4_D_5_5 : SYMM_SAMH_SYMM
8_4_5_4 generated by 8_5_5_4 4_D_5_5 : SAM_SAMH
. . .
```

A proof consists of one or more triple (link) ID values and the name of the rule that is applied on those triples:

link-id1 [*link-id2* ... l*ink-idn*] : *rule-name*

Example 2-4 Displaying Proof Information

To get the full subject, predicate, and object URIs for proofs, you can query the model view and the entailment (rules index) view. Example 2-4 (page 2-8) displays the LINK_ID value and associated triple contents using the model view MDSYS.SEMM_OWLTST and the entailment view MDSYS.SEMI_OWLTST_IDX.

```
SELECT to_char(x.triple.rdf_m_id, 'FMXXXXXXXXXXXXXXXX') ||'_'||
       to_char(x.triple.rdf_s_id, 'FMXXXXXXXXXXXXXXXX') ||'_'||
       to_char(x.triple.rdf_p_id, 'FMXXXXXXXXXXXXXXXX') ||'_'||
       to_char(x.triple.rdf_c_id, 'FMXXXXXXXXXXXXXXXX'),
       x.triple.get_triple()
  FROM (
   SELECT sdo_rdf_triple_s(
           t.canon_end_node_id,
           t.model_id,
           t.start_node_id,
           t.p_value_id,
           t.end_node_id) triple
     FROM (select * from mdsys.semm_owltst union all
```

```
          select * from mdsys.semi_owltst_idx
         ) t
    WHERE t.link_id IN ('4_D_5_5','8_5_5_4')
  ) x;

   LINK_ID  X.TRIPLE.GET_TRIPLE()(SUBJECT, PROPERTY, OBJECT)
----------  ------------------------------------------------------------
4_D_5_5 SDO_RDF_TRIPLE('<http://example.com/name/John>', '<http://www.w3.org/2002/07/
owl#sameAs>', '<http://example.com/name/JohnQ>')
8_5_5_4 SDO_RDF_TRIPLE('<http://example.com/name/JohnQ>', '<http://www.w3.org/
2002/07/owl#sameAs>', '<http://example.com/name/John>')
```

In Example 2-4 (page 2-8), for the proof entry 8_5_5_4 generated by 4_D_5_5 : SYMM_SAMH_SYMM for the triple with LINK_ID = 8_5_5_4, it is inferred from the triple with 4_D_5_5 using the symmetricity of `owl:sameAs`.

If the entailment status is INCOMPLETE and if the last entailment was generated without proof information, you cannot invoke SEM_APIS.CREATE_ENTAILMENT (page 11-15) with `PROOF=T`. In this case, you must first drop the entailment and create it again specifying `PROOF=T`.

2.2.5 Validating OWL Models and Entailments

An OWL ontology may contain errors, such as unsatisfiable classes, instances belonging to unsatisfiable classes, and two individuals asserted to be same and different at the same time. You can use the SEM_APIS.VALIDATE_MODEL (page 11-74) and SEM_APIS.VALIDATE_ENTAILMENT (page 11-73) functions to detect inconsistencies in the original data model and in the entailment, respectively.

Example 2-5 Validating an Entailment

Example 2-5 (page 2-9) shows uses the SEM_APIS.VALIDATE_ENTAILMENT (page 11-73) function, which returns a null value if no errors are detected or a VARRAY of strings if any errors are detected.

```
SQL>  -- Insert an offending triple.
SQL>  insert into owltst values (1, sdo_rdf_triple_s('owltst',
            'urn:C1', 'http://www.w3.org/2000/01/rdf-schema#subClassOf', 'http://
www.w3.org/2002/07/owl#Nothing'));

SQL> -- Drop entailment first.
SQL>  exec sem_apis.drop_entailment('owltst_idx');
PL/SQL procedure successfully completed.

SQL> -- Perform OWL inferencing.
SQL> exec sem_apis.create_entailment('owltst_idx', sem_models('OWLTST'),
sem_rulebases('OWLPRIME'));
PL/SQL procedure successfully completed.

SQL > set serveroutput on;
SQL > -- Now invoke validation API: sem_apis.validate_entailment
SQL >
declare
  lva mdsys.rdf_longVarcharArray;
  idx int;
begin
  lva := sem_apis.validate_entailment(sem_models('OWLTST'),
sem_rulebases('OWLPRIME')) ;

  if (lva is null) then
   dbms_output.put_line('No errors found.');
```

```
  else
    for idx in 1..lva.count loop
      dbms_output.put_line('Offending entry := ' || lva(idx)) ;
    end loop ;
  end if;
end ;
/

SQL> -- NOTE: The LINK_ID value and the numbers in the following
SQL> -- line are shortened for simplicity in this example. --

          Offending entry  := 1 10001 (4_2_4_8 2 4 8) Unsatisfiable class.
```

Each item in the validation report array includes the following information:

- Number of triples that cause this error (`1` in Example 2-5 (page 2-9))
- Error code (`10001` Example 2-5 (page 2-9))
- One or more triples (shown in parentheses in the output; `(4_2_4_8 2 4 8)` in Example 2-5 (page 2-9)).

  These numbers are the LINK_ID value and the ID values of the subject, predicate, and object.
- Descriptive error message (`Unsatisfiable class.` in Example 2-5 (page 2-9))

The output in Example 2-5 (page 2-9) indicates that the error is caused by one triple that asserts that a class is a subclass of an empty class `owl:Nothing`.

2.2.6 Using SEM_APIS.CREATE_ENTAILMENT for RDFS Inference

In addition to accepting OWL vocabularies, the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure accepts RDFS rulebases. The following example shows RDFS inference (all standard RDFS rules are defined in `http://www.w3.org/TR/rdf-mt/`):

```
EXECUTE sem_apis.create_entailment('rdfstst_idx', sem_models('my_model'),
sem_rulebases('RDFS'));
```

Because rules RDFS4A, RDFS4B, RDFS6, RDFS8, RDFS10, RDFS13 may not generate meaningful inference for your applications, you can deselect those components for faster inference. The following example deselects these rules.

```
EXECUTE sem_apis.create_entailment('rdfstst_idx', sem_models('my_model'),
sem_rulebases('RDFS'), SEM_APIS.REACH_CLOSURE, -
 'RDFS4A-, RDFS4B-, RDFS6-, RDFS8-, RDFS10-, RDFS13-');
```

2.2.7 Enhancing Inference Performance

This section describes suggestions for improving the performance of inference operations.

- Collect statistics before inferencing. After you load a large RDF/OWL data model, you should execute the SEM_PERF.GATHER_STATS (page 13-3) procedure. See the Usage Notes for that procedure (in SEM_PERF Package Subprograms (page 13-1)) for important usage information.

- Allocate sufficient temporary tablespace for inference operations. OWL inference support in Oracle relies heavily on table joins, and therefore uses significant temporary tablespace.
- Use the appropriate implementations of the SVFH and AVFH inference components.

  The default implementations of the SVFH and AVFH inference components work best when the number of restriction classes defined by `owl:someValuesFrom` and/or `owl:allValuesFrom` is low (as in the LUBM data sets). However, when the number of such classes is high (as in the Gene Ontology `http://www.geneontology.org/`), using non-procedural implementations of SVFH and AVFH may significantly improve performance.

  To disable the procedural implementations and to select the non-procedural implementations of SVFH and AVFH, include `'PROCSVFH=F'` and/or `'PROCAVFH=F'` in the options to SEM_APIS.CREATE_ENTAILMENT (page 11-15). Using the appropriate implementation for an ontology can provide significant performance benefits. For example, selecting the non-procedural implementation of SVFH for the NCI Thesaurus ontology (see `http://www.cancer.gov/research/resources/terminology`) produced a 960% performance improvement for the SVFH inference component (tested on a dual-core, 8GB RAM desktop system with 3 SATA disks tied together with Oracle ASM).

See also Optimizing owl:sameAs Inference (page 2-11).

2.2.8 Optimizing owl:sameAs Inference

You can optimize inference performance for large `owl:sameAs` cliques by specifying `'OPT_SAMEAS=T'` in the `options` parameter when performing OWLPrime entailment. (A **clique** is a graph in which every node of it is connected to, bidirectionally, every other node in the same graph.)

According to OWL semantics, the `owl:sameAs` construct is treated as an equivalence relation, so it is reflexive, symmetric, and transitive. As a result, during inference a full materialization of `owl:sameAs`-related entailments could significantly increase the size of the inferred graph. Consider the following example triple set:

```
:John  owl:sameAs  :John1 .
:John  owl:sameAs  :John2 .
:John2 :hasAge     "32" .
```

Applying OWLPrime inference (with the `SAM` component specified) to this set would generate the following new triples:

```
:John1  owl:sameAs  :John .
:John2  owl:sameAs  :John .
:John1  owl:sameAs  :John2 .
:John2  owl:sameAs  :John1 .
:John   owl:sameAs  :John .
:John1  owl:sameAs  :John1 .
:John2  owl:sameAs  :John2 .
:John   :hasAge     "32" .
:John1  :hasAge     "32" .
```

In the preceding example, `:John`, `:John1` and `:John2` are connected to each other with the `owl:sameAs` relationship; that is, they are members of an `owl:sameAs` **clique**. To provide optimized inference for large `owl:sameAs` cliques, you can consolidate `owl:sameAs` triples without sacrificing correctness by specifying

'OPT_SAMEAS=T' in the options parameter when performing OWLPrime entailment. For example:

```
EXECUTE sem_apis.create_entailment('M_IDX',sem_models('M'),
   sem_rulebases('OWLPRIME'),null,null,'OPT_SAMEAS=T');
```

When you specify this option, for each owl:sameAs clique, one resource from the clique is chosen as a canonical representative and all of the inferences for that clique are consolidated around that resource. Using the preceding example, if :John1 is the clique representative, after consolidation the inferred graph would contain only the following triples:

```
:John1 owl:sameAs :John1 .
:John1 :hasAge    "32" .
```

Some overhead is incurred with owl:sameAs consolidation. During inference, all asserted models are copied into the inference partition, where they are consolidated together with the inferred triples. Additionally, for very large asserted graphs, consolidating and removing duplicate triples incurs a large runtime overhead, so the OPT_SAMEAS=T option is recommended only for ontologies that have a large number of owl:sameAs relationships and large clique sizes.

After the OPT_SAMEAS=T option has been used for an entailment, all subsequent uses of SEM_APIS.CREATE_ENTAILMENT (page 11-15) for that entailment must also use OPT_SAMEAS=T, or an error will be reported. To disable optimized sameAs handling, you must first drop the entailment.

Clique membership information is stored in a view named MDSYS.SEMC_*entailment-name*, where *entailment-name* is the name of the entailment (rules index). Each MDSYS.SEMC_*entailment-name* view has the columns shown in Table 2-3 (page 2-12).

Table 2-3 MDSYS.SEMC_entailment_name View Columns

Column Name	Data Type	Description
MODEL_ID	NUMBER	ID number of the inferred model
VALUE_ID	NUMBER)	ID number of a resource that is a member of the owl:sameAs clique identified by CLIQUE_ID
CLIQUE_ID	NUMBER	ID number of the clique representative for the VALUE_ID resource

To save space, the MDSYS.SEMC_*entailment-name* view does not contain reflexive rows like (CLIQUE_ID, CLIQUE_ID).

2.2.8.1 Querying owl:sameAs Consolidated Inference Graphs

At query time, if the entailment queried was created using the OPT_SAMEAS=T option, the results are returned from an owl:sameAs-consolidated inference partition. The query results are not expanded to include the full owl:sameAs closure.

In the following example query, the only result returned would be :John1, which is the canonical clique representative.

```
SELECT A FROM TABLE (
  SEM_MATCH ('(?A :hasAge "32")',SEM_MODELS('M'),
    SEM_RULEBASES('OWLPRIME'),NULL, NULL));
```

With the preceding example, even though `:John2 :hasAge "32"` occurs in the model, it has been replaced during the inference consolidation phase where redundant triples are removed. However, you can expand the query results by performing a join with the MDSYS.SEMC_*rules-index-name* view that contains the consolidated `owl:sameAs` information. For example, to get expanded result set for the preceding SEM_MATCH query, you can use the following expanded query:

```
SELECT V.VALUE_NAME A_VAL FROM TABLE (
  SEM_MATCH ('(?A :hasAge "32")',SEM_MODELS('M'),
    SEM_RULEBASES('OWLPRIME'), NULL, NULL)) Q,
    MDSYS.RDF_VALUE$ V, MDSYS.SEMC_M_IDX C
  WHERE V.VALUE_ID  = C.VALUE_ID
     AND C.CLIQUE_ID = Q.A$RDFVID
  UNION ALL
    SELECT A A_VAL FROM TABLE (
      SEM_MATCH ('(?A :hasAge "32")',SEM_MODELS('M'),
        SEM_RULEBASES('OWLPRIME'),NULL, NULL));
```

Or, you could rewrite the preceding expanded query using a left outer join, as follows:

```
SELECT V.VALUE_NAME A_VAL FROM TABLE (
  SEM_MATCH ('(?A <http://hasAge> "33")',SEM_MODELS('M'),
   SEM_RULEBASES('OWLPRIME'), NULL, NULL)) Q,
  MDSYS.RDF_VALUE$ V,
   (SELECT value_id, clique_id FROM MDSYS.SEMC_M_IDX
  UNION ALL
    SELECT DISTINCT clique_id, clique_id
      FROM MDSYS.SEMC_M_IDX) C
 WHERE Q.A$RDFVID  = c.clique_id  (+)
   AND V.VALUE_ID  = nvl(C.VALUE_ID, Q.A$RDFVID);
```

2.2.9 Performing Incremental Inference

Incremental inference can be used to update entailments (rules indexes) efficiently after triple additions. There are two ways to enable incremental inference for an entailment:

- Specify the `options` parameter value `INC=T` when creating the entailment. For example:

  ```
  EXECUTE sem_apis.create_entailment ('M_IDX',sem_models('M'),
    sem_rulebases('OWLPRIME'),null,null, 'INC=T');
  ```

- Use the SEM_APIS.ENABLE_INC_INFERENCE (page 11-42) procedure.

  If you use this procedure, the entailment must have a VALID status. Before calling the procedure, if you do not own the models involved in the entailment, you must ensure that the respective model owners have used the SEM_APIS.ENABLE_CHANGE_TRACKING (page 11-41) procedure to enable change tracking for those models.

When incremental inference is enabled for an entailment, the parameter `INC=T` must be specified when invoking the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure for that entailment.

Incremental inference for an entailment depends on triggers for the application tables of the models involved in creating the entailment. This means that incremental inference works only when triples are inserted in the application tables underlying the entailment using conventional path loads, unless you specify the triples by using the `delta_in` parameter in the call to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, as in the following example, in which the triples from model

M_NEW will be added to model M, and entailment M_IDX will be updated with the new inferences:

```
EXECUTE sem_apis.create_entailment('M_IDX', sem_models('M'),
  sem_rulebases('OWLPRIME''), SEM_APIS.REACH_CLOSURE, null, null,
  sem_models('M_NEW'));
```

If multiple models are involved in the incremental inference call, then to specify the destination model to which the delta_in model or models are to be added, specify DEST_MODEL=<*model_name*> in the options parameter. For example, the following causes the semantic data in model M_NEW to be added to model M2:

```
EXECUTE sem_apis.create_entailment('M_IDX', sem_models('M1','M2','M3'),
sem_rulebases('OWLPRIME''), SEM_APIS.REACH_CLOSURE, null, 'DEST_MODEL=M2',
sem_models('M_NEW'));
```

Another way to bypass the conventional path loading requirement when using incremental inference is to set the UNDO_RETENTION parameter to cover the intervals between entailments when you perform bulk loading. For example, if the last entailment was created 6 hours ago, the UNDO_RETENTION value should be set to greater than 6 hours; if it is less than that, then (given a heavy workload and limited undo space) it is not guaranteed that all relevant undo information will be preserved for incremental inference to apply. In such cases, the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure falls back to regular (non-incremental) inference.

To check if change tracking is enabled on a model, use the SEM_APIS.GET_CHANGE_TRACKING_INFO (page 11-48) procedure. To get additional information about incremental inference for an entailment, use the SEM_APIS.GET_INC_INF_INFO (page 11-49) procedure.

The following restrictions apply to incremental inference:

- It does not work with optimized owl:sameAs handling (OPT_SAMEAS), user-defined rules, VPD-enabled models, or version-enabled models.
- It supports only the addition of triples. With updates or deletions, the entailment will be completely rebuilt.
- It depends on triggers on application tables.
- Column types (RAW8 or NUMBER) used in incremental inference must be consistent. For instance, if RAW8=T is used to build the entailment initially, then for every subsequent SEM_APIS.CREATE_ENTAILMENT (page 11-15) call the same option must be used. To change the column type to NUMBER, you must drop and rebuild the entailment.

2.2.10 Using Parallel Inference

Parallel inference can improve inference performance by taking advantage of the capabilities of a multi-core or multi-CPU architectures. To use parallel inference, specify the DOP (degree of parallelism) keyword and an appropriate value when using the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure. For example:

```
EXECUTE sem_apis.create_entailment('M_IDX',sem_models('M'),
      sem_rulebases('OWLPRIME'), sem_apis.REACH_CLOSURE, null, 'DOP=4');
```

Specifying the DOP keyword causes parallel execution to be enabled for an Oracle-chosen set of inference components

The success of parallel inference depends heavily on a good hardware configuration of the system on which the database is running. The key is to have a "balanced" system that implements the best practices for database performance tuning and Oracle SQL parallel execution. For example, do not use a single 1 TB disk for an 800 GB database, because executing SQL statements in parallel on a single physical disk can even be slower than executing SQL statements in serial mode. Parallel inference requires ample memory; for each CPU core, you should have at least 4 GB of memory.

Parallel inference is best suited for large ontologies; however, inference performance can also improve for small ontologies.

There is some transient storage overhead associated with using parallel inference. Parallel inference builds a source table that includes all triples based on all the source RDF/OWL models and existing inferred graph. This table might use an additional 10 to 30 percent of storage compared to the space required for storing data and index of the source models.

2.2.11 Using Named Graph Based Inferencing (Global and Local)

The default inferencing in Oracle Database takes all asserted triples from all the source model or models provided and applies semantic rules on top of all the asserted triples until an inference closure is reached. Even if the given source models contain one or more multiple named graphs, it makes no difference because all assertions, whether part of a named graph or not, are treated the same as if they come from a single graph. (For an introduction to named graph support in RDF Semantic Graph, see Named Graphs (page 1-16).)

This default inferencing can be thought of as completely "global" in that it does not consider named graphs at all.

However, if you use named graphs, you can override the default inferencing and have named graphs be considered by using either of the following features:

- Named graph based *global* inference (NGGI), which treats all specified named graphs as a unified graph. NGGI lets you narrow the scope of triples to be considered, while enabling great flexibility; it is explained in Named Graph Based Global Inference (NGGI) (page 2-15).
- Named graph based *local* inference (NGLI), which treats each specified named graph as a separate entity. NGLI is explained in Named Graph Based Local Inference (NGLI) (page 2-16).

For using NGGI and NGLI together, see a recommended usage flow in Using NGGI and NGLI Together (page 2-17).

You specify NGGI or NGLI through certain parameters and options to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure when you create an entailment (rules index).

2.2.11.1 Named Graph Based Global Inference (NGGI)

Named graph based global inference (NGGI) enables you to narrow the scope of triples used for inferencing at the named graph level (as opposed to the model level). It also enables great flexibility in selecting the scope; for example, you can include triples from zero or more named graphs and/or from the default graph, and you can include all triples with a null graph name from specified models.

For example, in a hospital application you may only want to apply the inference rules on all the information contained in a set of named graphs describing patients of a particular hospital. If the patient-related named graphs contains only instance-related

assertions (ABox), you can specify one or multiple additional schema related-models (TBox), as in Example 2-6 (page 2-16).

Example 2-6 Named Graph Based Global Inference

```
EXECUTE sem_apis.create_entailment(
  'patients_inf',
  models_in         => sem_models('patients','hospital_ontology'),
  rulebases_in      => sem_rulebases('owl2rl'),
  passes            => SEM_APIS.REACH_CLOSURE,
  inf_components_in => null,
  options           => 'DOP=4,RAW8=T',
  include_default_g => sem_models('hospital_ontology'),
  include_named_g   =>
sem_graphs('<urn:hospital1_patient1>','<urn:hospital1_patient2>'),
  inf_ng_name       => '<urn:inf_graph_for_hospital1>'
  );
```

In Example 2-6 (page 2-16):

- Two models are involved: `patients` contains a set of named graphs where each named graph holds triples relevant to a particular patient, and `hospital_ontology` contains schema information describing concepts and relationships that are defined for hospitals. These two models together are the source models, and they set up an overall scope for the inference.
- The `include_default_g` parameter causes all triples with a NULL graph name in the specified models to participate in NGGI. In this example, all triples with a NULL graph name in model `hospital_ontology` will be included in NGGI.
- The `include_named_g` parameter causes all triples from the specified named graphs (across all source models) to participate in NGGI. In this example, triples from named graphs `<urn:hospital1_patient1>` and `<urn:hospital1_patient2>` will be included in NGGI.
- The `inf_ng_name` parameter assigns graph name `<urn:inf_graph_for_hospital1>` to all the new triples inferred by NGGI.

2.2.11.2 Named Graph Based Local Inference (NGLI)

Named graph based local inference (NGLI) treats each named graph as a separate entity instead of viewing the graphs as a single unified graph. Inference logic is performed within the boundary of each entity. You can specify schema-related assertions (TBox) in a default graph, and that default graph will participate the inference of each named graph. For example, inferred triples based on a graph with name `G1` will be assigned the same graph name `G1` in the inferred data partition.

Assertions from any two separate named graphs will never jointly produce any new assertions.

For example, assume the following:

- Graph `G1` includes the following assertion:

  ```
  :John  :hasBirthMother  :Mary .
  ```

- Graph `G2` includes the following assertion:

  ```
  :John  :hasBirthMother  :Bella .
  ```

- The default graph includes the assertion that `:hasBirthMother` is an `owl:FunctionalProperty`. (This assertion has a null graph name.)

In this example, named graph based *local* inference (NGLI) will ***not*** infer that `:Mary` is `owl:sameAs :Bella` because the two assertions are from two distinct graphs, `G1` and `G2`. By contrast, a named graph based *global* inference (NGGI) that includes `G1`, `G2`, and the functional property definition *would* be able to infer that `:Mary` is `owl:sameAs :Bella`.

NGLI currently does not work together with proof generation, user-defined rules, optimized `owl:sameAs` handling, or incremental inference.

Example 2-7 Named Graph Based Local Inference

Example 2-7 (page 2-17) shows NGLI.

```
EXECUTE sem_apis.create_entailment(
  'patients_inf',
  models_in         => sem_models('patients','hospital_ontology'),
  rulebases_in      => sem_rulebases('owl2rl'),
  passes            => SEM_APIS.REACH_CLOSURE,
  inf_components_in => null,
  options           => 'LOCAL_NG_INF=T'
);
```

In Example 2-7 (page 2-17):

- The two models patients and hospital_ontology together are the source models, and they set up an overall scope for the inference, similar to the case of global inference in Example 2-6 (page 2-16). All triples with a null graph name are treated as part of the common schema (TBox). Inference is performed within the boundary of every single named graph combined with the common schema.
- Then `options` parameter keyword-value pair `LOCAL_NG_INF=T` specifies that named graph based local inference (NGLI) is to be performed.

Note that, by design, NGLI does not apply to the default graph itself. However, you can easily apply named graph based global inference (NGGI) on the default graph and set the `inf_ng_name` parameter to null. In this way, the TBox inference is precomputed, improving the overall performance and storage consumption.

NGLI does not allow the following:

- Inferring new relationships based on a mix of triples from multiple named graphs
- Inferring new relationships using only triples from the default graph.

To get the inference that you would normally expect, you should keep schema assertions and instance assertions separate. Schema assertions (for example, `:A rdfs:subClassOf :B` and `:p1 rdfs:subPropertyOf :p2`) should be stored in the default graph as unnamed triples (with null graph names). By contrast, instance assertions (for example, `:X :friendOf :Y`) should be stored in one of the named graphs.

For a discussion and example of using NGLI to perform document-centric inference with semantically indexed documents, see Performing Document-Centric Inference (page 4-16).

2.2.11.3 Using NGGI and NGLI Together

The following is a recommended usage flow for using NGGI and NGLI together. It assumes that TBox and ABox are stored in two separate models, that TBox contains

schema definitions and all triples in the TBox have a null graph name, but that ABox consists of a set of named graphs describing instance-related data.

1. Invoke NGGI on the TBox by itself. For example:

```
EXECUTE sem_apis.create_entailment(
    'TEST_INF',
    sem_models('abox','tbox'),
    sem_rulebases('owl2rl'),
    SEM_APIS.REACH_CLOSURE,
    include_default_g=>sem_models('tbox')
);
```

2. Invoke NGLI for all named graphs. For example:

```
EXECUTE sem_apis.create_entailment(
    'TEST_INF',
    sem_models('abox','tbox'),
    sem_rulebases('owl2rl'),
    SEM_APIS.REACH_CLOSURE,
    options => 'LOCAL_NG_INF=T,ENTAIL_ANYWAY=T'
);
```

   ENTAIL_ANYWAY=T is specified because the NGGI call in step 1will set the status of inferred graph to VALID, and the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure call in step 2 will quit immediately unless ENTAIL_ANYWAY=T is specified.

2.2.12 Performing Selective Inferencing (Advanced Information)

Selective inferencing is component-based inferencing, in which you limit the inferencing to specific OWL components that you are interested in. To perform selective inferencing, use the inf_components_in parameter to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure to specify a comma-delimited list of components. The final inferencing is determined by the *union* of rulebases specified and the components specified.

Example 2-8 Performing Selective Inferencing

Example 2-8 (page 2-18) limits the inferencing to the class hierarchy from subclass (SCOH) relationship and the property hierarchy from subproperty (SPOH) relationship. This example creates an empty rulebase and then specifies the two components ('SCOH,SPOH') in the call to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure.

```
EXECUTE sem_apis.create_rulebase('my_rulebase');

EXECUTE sem_apis.create_entailment('owltst_idx', sem_models('owltst'),
sem_rulebases('my_rulebase'), SEM_APIS.REACH_CLOSURE, 'SCOH,SPOH');
```

The following component codes are available: SCOH, COMPH, DISJH, SYMMH, INVH, SPIH, MBRH, SPOH, DOMH, RANH, EQCH, EQPH, FPH, IFPH, DOM, RAN, SCO, DISJ, COMP, INV, SPO, FP, IFP, SYMM, TRANS, DIF, SAM, CHAIN, HASKEY, ONEOF, INTERSECT, INTERSECTSCOH, MBRLST, PROPDISJH, SKOSAXIOMS, SNOMED, SVFH, THINGH, THINGSAM, UNION, RDFP1, RDFP2, RDFP3, RDFP4, RDFP6, RDFP7, RDFP8AX, RDFP8BX, RDFP9, RDFP10, RDFP11, RDFP12A, RDFP12B, RDFP12C, RDFP13A, RDFP13B, RDFP13C, RDFP14A, RDFP14BX, RDFP15, RDFP16, RDFS2, RDFS3, RDFS4a, RDFS4b, RDFS5, RDFS6, RDFS7, RDFS8, RDFS9, RDFS10, RDFS11, RDFS12, RDFS13

The rules corresponding to components with a prefix of *RDFP* can be found in *Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary*, by H.J. Horst.

The syntax for deselecting a component is *component_name* followed by a minus (-) sign. For example, the following statement performs OWLPrime inference without calculating the `subClassOf` hierarchy:

```
EXECUTE sem_apis.create_entailment('owltst_idx', sem_models('owltst'),
sem_rulebases('OWLPRIME'), SEM_APIS.REACH_CLOSURE, 'SCOH-');
```

By default, the OWLPrime rulebase implements the transitive semantics of owl:sameAs. OWLPrime does not include the following rules (semantics):

```
U   owl:sameAs   V  .
U     p    X  .        ==>   V  p   X   .

U   owl:sameAs   V  .
X     p    U  .        ==>   X   p   V   .
```

The reason for not including these rules is that they tend to generate many assertions. If you need to include these assertions, you can include the SAM component code in the call to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure.

2.3 Using Semantic Operators to Query Relational Data

You can use semantic operators to query relational data in an ontology-assisted manner, based on the semantic relationship between the data in a table column and terms in an ontology. The SEM_RELATED semantic operator retrieves rows based on semantic relatedness. The SEM_DISTANCE semantic operator returns distance measures for the semantic relatedness, so that rows returned by the SEM_RELATED operator can be ordered or restricted using the distance measure. The index type MDSYS.SEM_INDEXTYPE allows efficient execution of such queries, enabling scalable performance over large data sets.

2.3.1 Using the SEM_RELATED Operator

Referring to the cancer ontology example in Example: Cancer Ontology (page 2-1), consider the following query that requires semantic matching: *Find all patients whose diagnosis is of the type 'Immune_System_Disorder'*. A typical database query of the PATIENTS table (described in Example: Cancer Ontology (page 2-1)) involving syntactic match will not return any rows, because no rows have a DIAGNOSIS column containing the exact value `Immune_System_Disorder`. For example the following query will not return any rows:

```
SELECT diagnosis FROM patients WHERE diagnosis = 'Immune_System_Disorder';
```

Example 2-9 SEM_RELATED Operator

However, many rows in the patient data table are relevant, because their diagnoses fall under this class. Example 2-9 (page 2-19) uses the SEM_RELATED operator (instead of lexical equality) to retrieve all the relevant rows from the patient data table. (In this example, the term `Immune_System_Disorder` is prefixed with a namespace, and the default assumption is that the values in the table column also have a namespace prefix. However, that might not always be the case, as explained in Using URIPREFIX When Values Are Not Stored as URIs (page 2-24).)

```
SELECT diagnosis FROM patients
  WHERE SEM_RELATED (diagnosis,
```

```
    '<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
    '<http://www.example.org/medical_terms/Immune_System_Disorder>',
    sem_models('medical_ontology'), sem_rulebases('owlprime')) = 1;
```

The SEM_RELATED operator has the following attributes:

```
SEM_RELATED(
  sub  VARCHAR2,
  predExpr  VARCHAR2,
  obj  VARCHAR2,
  ontologyName  SEM_MODELS,
  ruleBases  SEM_RULEBASES,
  index_status  VARCHAR2,
  lower_bound INTEGER,
  upper_bound INTEGER
 ) RETURN INTEGER;
```

The `sub` attribute is the name of table column that is being searched. The terms in the table column are typically the subject in a <subject, predicate, object> triple pattern.

The `predExpr` attribute represents the predicate that can appear as a label of the edge on the path from the subject node to the object node.

The `obj` attribute represents the term in the ontology for which related terms (related by the `predExpr` attribute) have to be found in the table (in the column specified by the `sub` attribute). This term is typically the object in a <subject, predicate, object> triple pattern. (In a query with the equality operator, this would be the query term.)

The `ontologyName` attribute is the name of the ontology that contains the relationships between terms.

The `rulebases` attribute identifies one or more rulebases whose rules have been applied to the ontology to infer new relationships. The query will be answered based both on relationships from the ontology and the inferred new relationships when this attribute is specified.

The `index_status` optional attribute lets you query the data even when the relevant entailment (created when the specified rulebase was applied to the ontology) does not have a valid status. If this attribute is null, the query returns an error if the entailment does not have a valid status. If this attribute is not null, it must be the string `VALID`, `INCOMPLETE`, or `INVALID`, to specify the minimum status of the entailment for the query to succeed. Because OWL does not guarantee monotonicity, the value `INCOMPLETE` should not be used when an OWL Rulebase is specified.

The `lower_bound` and `upper_bound` optional attributes let you specify a bound on the distance measure of the relationship between terms that are related. See Using the SEM_DISTANCE Ancillary Operator (page 2-20) for the description of the distance measure.

The SEM_RELATED operator returns 1 if the two input terms are related with respect to the specified `predExpr` relationship within the ontology, and it returns 0 if the two input terms are not related. If the lower and upper bounds are specified, it returns 1 if the two input terms are related with a distance measure that is greater than or equal to `lower_bound` and less than or equal to `upper_bound`.

2.3.2 Using the SEM_DISTANCE Ancillary Operator

The SEM_DISTANCE ancillary operator computes the distance measure for the rows filtered using the SEM_RELATED operator. The SEM_DISTANCE operator has the following format:

```
SEM_DISTANCE (number) RETURN NUMBER;
```

The `number` attribute can be any number, as long as it matches the number that is the last attribute specified in the call to the SEM_RELATED operator (see Example 2-10 (page 2-21)). The number is used to match the invocation of the ancillary operator SEM_DISTANCE with a specific SEM_RELATED (primary operator) invocation, because a query can have multiple invocations of primary and ancillary operators.

Example 2-10 SEM_DISTANCE Ancillary Operator

Example 2-10 (page 2-21) expands Example 2-9 (page 2-19) to show several statements that include the SEM_DISTANCE ancillary operator, which gives a measure of how closely the two terms (here, a patient's diagnosis and the term `Immune_System_Disorder`) are related by measuring the distance between the terms. Using the cancer ontology described in Example: Cancer Ontology (page 2-1), the distance between `AIDS` and `Immune_System_Disorder` is 3.

```
SELECT diagnosis, SEM_DISTANCE(123) FROM patients
  WHERE SEM_RELATED (diagnosis,
    '<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
    '<http://www.example.org/medical_terms/Immune_System_Disorder>',
    sem_models('medical_ontology'), sem_rulebases('owlprime'), 123) = 1;

SELECT diagnosis FROM patients
  WHERE SEM_RELATED (diagnosis,
    '<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
    '<http://www.example.org/medical_terms/Immune_System_Disorder>',
    sem_models('medical_ontology'), sem_rulebases('owlprime'), 123) = 1
  ORDER BY SEM_DISTANCE(123);

SELECT diagnosis, SEM_DISTANCE(123) FROM patients
  WHERE SEM_RELATED (diagnosis,
    '<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
    '<http://www.example.org/medical_terms/Immune_System_Disorder>',
    sem_models('medical_ontology'), sem_rulebases('owlprime'), 123) = 1
  AND SEM_DISTANCE(123) <= 3;
```

Example 2-11 Using SEM_DISTANCE to Restrict the Number of Rows Returned

Example 2-11 (page 2-21) uses distance information to restrict the number of rows returned by the primary operator. All rows with a term related to the object attribute specified in the SEM_RELATED invocation, but with a distance of greater than or equal to 2 and less than or equal to 4, are retrieved.

```
SELECT diagnosis FROM patients
  WHERE SEM_RELATED (diagnosis,
    '<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
    '<http://www.example.org/medical_terms/Immune_System_Disorder>',
    sem_models('medical_ontology'), sem_rulebases('owlprime'), 2, 4) = 1;
```

In Example 2-11 (page 2-21), the lower and upper bounds are specified using the `lower_bound` and `upper_bound` parameters in the SEM_RELATED operator instead of using the SEM_DISTANCE operator. The SEM_DISTANCE operator can be also be used for restricting the rows returned, as shown in the last SELECT statement in Example 2-10 (page 2-21).

2.3.2.1 Computation of Distance Information

Distances are generated for the following properties during inference (entailment): OWL properties defined as transitive properties, and RDFS `subClassOf` and RDFS `subPropertyOf` properties. The distance between two terms linked through these

properties is computed as the shortest distance between them in a hierarchical class structure. Distances of two terms linked through other properties are undefined and therefore set to null.

Each transitive property link in the original model (viewed as a hierarchical class structure) has a distance of 1, and the distance of an inferred triple is generated according to the number of links between the two terms. Consider the following hypothetical sample scenarios:

- If the original graph contains `C1 rdfs:subClassOf C2` and `C2 rdfs:subClassOf C3`, then `C1 rdfs:subClassof of C3` will be derived. In this case:
  - `C1 rdfs:subClassOf C2`: distance = 1, because it exists in the model.
  - `C2 rdfs:subClassOf C3`: distance = 1, because it exists in the model.
  - `C1 rdfs:subClassOf C3`: distance = 2, because it is generated during inference.
- If the original graph contains `P1 rdfs:subPropertyOf P2` and `P2 rdfs:subPropertyOf P3`, then P1 rdfs:subPropertyOf P3 will be derived. In this case:
  - `P1 rdfs:subPropertyOf P2`: distance = 1, because it exists in the model.
  - `P2 rdfs:subPropertyOf P3`: distance = 1, because it exists in the model.
  - `P1 rdfs:subPropertyOf P3`: distance = 2, because it is generated during inference.
- If the original graph contains `C1 owl:equivalentClass C2` and `C2 owl:equivalentClass C3`, then `C1 owl:equivalentClass C3` will be derived. In this case:
  - `C1 owl:equivalentClass C2`: distance = 1, because it exists in the model.
  - `C2 owl:equivalentClass C3`: distance = 1, because it exists in the model.
  - `C1 owl:equivalentClass C3`: distance = 2, because it is generated during inference.

The SEM_RELATED operator works with user-defined rulebases. However, using the SEM_DISTANCE operator with a user-defined rulebase is not yet supported, and will raise an error.

2.3.3 Creating a Semantic Index of Type MDSYS.SEM_INDEXTYPE

When using the SEM_RELATED operator, you can create a semantic index of type MDSYS.SEM_INDEXTYPE on the column that contains the ontology terms. Creating such an index will result in more efficient execution of the queries. The CREATE INDEX statement must contain the `INDEXTYPE IS MDSYS.SEM_INDEXTYPE` clause, to specify the type of index being created.

Example 2-12 Creating a Semantic Index

Example 2-12 (page 2-22) creates a semantic index named DIAGNOSIS_SEM_IDX on the DIAGNOSIS column of the PATIENTS table using the `Cancer_Ontology` ontology.

```
CREATE INDEX diagnosis_sem_idx
  ON patients (diagnosis)
  INDEXTYPE IS MDSYS.SEM_INDEXTYPE;
```

The column on which the index is built (DIAGNOSIS in Example 2-12 (page 2-22)) must be the first parameter to the SEM_RELATED operator, in order for the index to be used. If it not the first parameter, the index is not used during the execution of the query.

Example 2-13 Creating a Semantic Index Specifying a Model and Rulebase

To improve the performance of certain semantic queries, you can cause statistical information to be generated for the semantic index by specifying one or more models and rulebases when you create the index. Example 2-13 (page 2-23) creates an index that will also generate statistics information for the specified model and rulebase. The index can be used with other models and rulebases during query, but the statistical information will be used only if the model and rulebase specified during the creation of the index are the same model and rulebase specified in the query.

```
CREATE INDEX diagnosis_sem_idx
  ON patients (diagnosis)
  INDEXTYPE IS MDSYS.SEM_INDEXTYPE('ONTOLOGY_MODEL(medical_ontology),
    RULEBASE(OWLPrime)');
```

Example 2-14 Query Benefitting from Generation of Statistical Information

The statistical information is useful for queries that return top-k results sorted by semantic distance. Example 2-14 (page 2-23) shows such a query.

```
SELECT /*+ FIRST_ROWS */ diagnosis FROM patients
  WHERE SEM_RELATED (diagnosis,
    '<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
    '<http://www.example.org/medical_terms/Immune_System_Disorder>',
    sem_models('medical_ontology'), sem_rulebases('owlprime'), 123) = 1
  ORDER BY SEM_DISTANCE(123);
```

2.3.4 Using SEM_RELATED and SEM_DISTANCE When the Indexed Column Is Not the First Parameter

If an index of type MDSYS.SEM_INDEXTYPE has been created on a table column that is the first parameter to the SEM_RELATED operator, the index will be used. For example, the following query retrieves all rows that have a value in the DIAGNOSIS column that is a subclass of (`rdfs:subClassOf`) `Immune_System_Disorder`.

```
SELECT diagnosis FROM patients
  WHERE SEM_RELATED (diagnosis,
    '<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
    '<http://www.example.org/medical_terms/Immune_System_Disorder>',
    sem_models('medical_ontology'), sem_rulebases('owlprime')) = 1;
```

Assume, however, that this query instead needs to retrieve all rows that have a value in the DIAGNOSIS column for which `Immune_System_Disorder` is a subclass. You could rewrite the query as follows:

```
SELECT diagnosis FROM patients
  WHERE SEM_RELATED
    ('<http://www.example.org/medical_terms/Immune_System_Disorder>',
    '<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
    diagnosis,
    sem_models('medical_ontology'), sem_rulebases('owlprime')) = 1;
```

However, in this case a semantic index on the DIAGNOSIS column will not be used, because it is not the first parameter to the SEM_RELATED operator. To cause the index to be used, you can change the preceding query to use the `inverseOf` keyword, as follows:

```
SELECT diagnosis FROM patients
  WHERE SEM_RELATED (diagnosis,
    'inverseOf(http://www.w3.org/2000/01/rdf-schema#subClassOf)',
    '<http://www.example.org/medical_terms/Immune_System_Disorder>',
    sem_models('medical_ontology'), sem_rulebases('owlprime')) = 1;
```

This form causes the table column (on which the index is built) to be the first parameter to the SEM_RELATED operator, and it retrieves all rows that have a value in the DIAGNOSIS column for which `Immune_System_Disorder` is a subclass.

2.3.5 Using URIPREFIX When Values Are Not Stored as URIs

By default, the semantic operator support assumes that the values stored in the table are URIs. These URIs can be from different namespaces. However, if the values in the table do not have URIs, you can use the URIPREFIX keyword to specify a URI when you create the semantic index. In this case, the specified URI is prefixed to the value in the table and stored in the index structure. (One implication is that multiple URIs cannot be used).

Example 2-15 (page 2-24) creates a semantic index that uses a URI prefix.

Example 2-15 Specifying a URI Prefix During Semantic Index Creation

```
CREATE INDEX diagnosis_sem_idx
  ON patients (diagnosis)
  INDEXTYPE IS MDSYS.SEM_INDEXTYPE
  PARAMETERS('URIPREFIX(<http://www.example.org/medical/>)');
```

The slash (/) character at the end of the URI is important, because the URI is prefixed to the table value (in the index structure) without any parsing.

3

Simple Knowledge Organization System (SKOS) Support

You can perform inferencing based on a core subset of the Simple Knowledge Organization System (SKOS) data model, which is especially useful for representing thesauri, classification schemes, taxonomies, and other types of controlled vocabulary. SKOS is based on standard semantic web technologies including RDF and OWL, which makes it easy to define the formal semantics for those knowledge organization systems and to share the semantics across applications.

Support is provided for most, but not all, of the features of SKOS, the detailed specification of which is available at `http://www.w3.org/TR/skos-reference/`.

Around 40 SKOS-specific terms are included in the RDF Semantic Graph support, such as `skos:broader`, `skos:relatedMatch`, and `skos:Concept`. Over 100 SKOS axiomatic triples have been added, providing the basic coverage of SKOS semantics. However, support is not included for the integrity conditions described in the SKOS specification.

To perform SKOS-based inferencing, specify the system-defined `SKOSCORE` rulebase in the `rulebases_in` parameter in the call to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, as in the following example:

```
EXECUTE sem_apis.create_entailment('tstidx',sem_models('tst'),
sem_rulebases('skoscore'));
```

Example 3-1 (page 3-1) defines, in Turtle format, a simple electronics scheme and two relevant concepts, cameras and digital cameras. Its meaning is straightforward and its representation is in RDF. It can be managed by Oracle Database in the same way as other RDF and OWL data.

Example 3-1 SKOS Definition of an Electronics Scheme

```
ex1:electronicsScheme rdf:type skos:ConceptScheme;

ex1:cameras rdf:type skos:Concept;
   skos:prefLabel "cameras"@en;
   skos:inScheme ex1:electronicsScheme.

ex1:digitalCameras rdf:type skos:Concept;
   skos:prefLabel "digital cameras"@en;
   skos:inScheme ex1:electronicsScheme.

ex1:digitalCameras skos:broader ex1:cameras.
```

3.1 Supported and Unsupported SKOS Semantics

This section describes features of SKOS semantics that are and are not supported by Oracle Database.

3.1.1 Supported SKOS Semantics

All terms defined in SKOS and SKOS extension for labels are recognized. When the SKOSCORE rulebase is chosen for inference, the recognized terms include the following:

```
skos:altLabel
skos:broader
skos:broaderTransitive
skos:broadMatch
skos:changeNote
skos:closeMatch
skos:Collection
skos:Concept
skos:ConceptScheme
skos:definition
skos:editorialNote
skos:exactMatch
skos:example
skos:hasTopConcept
skos:hiddenLabel
skos:historyNote
skos:inScheme
skos:mappingRelation
skos:member
skos:memberList
skos:narrower
skos:narrowerTransitive
skos:narrowMatch
skos:notation
skos:note
skos:OrderedCollection
skos:prefLabel
skos:related
skos:relatedMatch
skos:scopeNote
skos:semanticRelation
skos:topConceptOf
skosxl:altLabel
skosxl:hiddenLabel
skosxl:Label
skosxl:labelRelation
skosxl:literalForm
skosxl:prefLabel
```

Most SKOS axioms and definitions are supported including the following: S1-S8, S10-S11, S15-S26, S28-S31, S33-S36, S38-S45, S47-S50, and S53-S54. (See the SKOS detailed specification for definitions.)

Most SKOS integrity conditions are supported, including S9, S13, S27, S37, and S46.

S52 is partially supported.

S55, S56, and S57 are not supported by default.

- S55, the property chain (`skosxl:prefLabel, skosxl:literalForm`), is a subproperty of `skos:prefLabel`.
- S56, the property chain (`skosxl:altLabel, skosxl:literalForm`), is a subproperty of `skos:altLabel`.

- S57, the property chain (`skosxl:hiddenLabel, skosxl:literalForm`), is a subproperty of `skos:hiddenLabel.chains`.

However, S55, S56, and S57 can be implemented using the OWL 2 subproperty chain construct. For information about property chain handling, see Property Chain Handling (page 3-4).

3.1.2 Unsupported SKOS Semantics

The following features of SKOS semantics are not supported:

- S12 and S51: The `rdfs:range` of the relevant predicates is the class of RDF plain literals. There is no check that the object values of these predicates are indeed plain literals; however, applications can perform such a check.
- S14: A resource has no more than one value of `skos:prefLabel` per language tag. This integrity condition is even beyond OWL FULL semantics, and it is not enforced in the current release.
- S32: The `rdfs:range of skos:member` is the union of classes `skos:Concept` and `skos:Collection`. This integrity condition is not enforced.
- S55, S56, and S57 are not supported by default, but they can be implemented using the OWL 2 subproperty chain construct, as explained in Supported SKOS Semantics (page 3-2).

3.2 Performing Inference on SKOS Models

To create an SKOS model, use the same procedure (SEM_APIS.CREATE_SEM_MODEL (page 11-27)) as for creating a semantic model. You can load data into an SKOS model in the same way as for semantic models.

To infer new relationships for one or more SKOS models, use the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure with the system-defined rulebase `SKOSCORE`. For example:

```
EXECUTE sem_apis.create_entailment('tstidx',sem_models('tst'),
sem_rulebases('skoscore'));
```

The inferred data will include many of the axioms defined in the SKOS detailed specification. Like other system-defined rulebases, SKOSCORE has no explicit rules; all the semantics supported are coded into the implementation.

3.2.1 Validating SKOS Models and Entailments

You can use the SEM_APIS.VALIDATE_ENTAILMENT (page 11-73) and SEM_APIS.VALIDATE_MODEL (page 11-74) procedures to validate the supported integrity conditions. The output will include any inconsistencies caused by the supported integrity conditions, such as OWL 2 `propertyDisjointWith` and `S52`.

Example 3-2 (page 3-3) validates an SKOS entailment.

Example 3-2 Validating an SKOS Entailment

```
set serveroutput on
declare
  lva mdsys.rdf_longVarcharArray;
  idx int;
begin
  lva := sem_apis.validate_entailment(sdo_rdf_models('tstskos'),
```

```
sem_rulebases('skoscore'));
  if (lva is null) then
    dbms_output.put_line('No conflicts');
  else
  for idx in 1..lva.count loop
    dbms_output.put_line('entry ' || idx || ' ' || lva(idx));
  end loop;
  end if;
end;
 /
```

3.2.2 Property Chain Handling

The SKOS S55, S56, and S57 semantics are not supported by default. However, you can add support for them by using the OWL 2 subproperty chain construct.

Example 3-3 (page 3-4) inserts the necessary chain definition triples for S55 into an SKOS model. After the insertion, an invocation of SEM_APIS.CREATE_ENTAILMENT (page 11-15) that specifies the SKOSCORE rulebase will include the semantics defined in S55.

Example 3-3 Property Chain Insertions to Implement S55

```
INSERT INTO tst VALUES(sdo_rdf_triple_s('tst','<http://www.w3.org/2004/02/skos/
core#prefLabel>', '<http://www.w3.org/2002/07/owl#propertyChainAxiom>', '_:jA1'));
INSERT INTO tst VALUES(sdo_rdf_triple_s('tst','_:jA1', '<http://www.w3.org/
1999/02/22-rdf-syntax-ns#first>', '<http://www.w3.org/2008/05/skos-xl#prefLabel>'));
INSERT INTO tst VALUES(sdo_rdf_triple_s('tst','_:jA1', '<http://www.w3.org/
1999/02/22-rdf-syntax-ns#rest>', '_:jA2'));
INSERT INTO tst VALUES(sdo_rdf_triple_s('tst','_:jA2', '<http://www.w3.org/
1999/02/22-rdf-syntax-ns#first>', '<http://www.w3.org/2008/05/skos-
xl#literalForm>'));
INSERT INTO tst VALUES(sdo_rdf_triple_s('tst','_:jA2', '<http://www.w3.org/
1999/02/22-rdf-syntax-ns#rest>', '<http://www.w3.org/1999/02/22-rdf-syntax-
ns#nil>'));
```

4

Semantic Indexing for Documents

Information extractors locate and extract meaningful information from unstructured documents. The ability to search for documents based on this extracted information is a significant improvement over the keyword-based searches supported by the full-text search engines.

Semantic indexing for documents introduces an index type that can make use of information extractors and annotators to semantically index documents stored in relational tables. Documents indexed semantically can be searched using SEM_CONTAINS operator within a standard SQL query. The search criteria for these documents are expressed using SPARQL query patterns that operate on the information extracted from the documents, as in the following example.

```
SELECT docId
FROM   Newsfeed
WHERE  SEM_CONTAINS (article,
     ' { ?org    rdf:type            typ:Organization  .
         ?org    pred:hasCategory    cat:BusinessFinance } ', ..) = 1
```

The key components that facilitate Semantic Indexing for documents in an Oracle Database include:

- Extensible information extractor framework, which allows third-party information extractors to be plugged into the database
- SEM_CONTAINS operator to identify documents of interest, based on their extracted information, using standard SQL queries
- SEM_CONTAINS_SELECT ancillary operator to return relevant information about the documents identified using SEM_CONTAINS operator
- SemContext index type to interact with the information extractor and manage the information extracted from a document set in an index structure and to facilitate semantically meaningful searches on the documents

The application program interface (API) for managing extractor policies and semantic indexes created for documents is provided in the SEM_RDFCTX PL/SQL package. SEM_RDFCTX Package Subprograms (page 14-1) provides the reference information about the subprograms in SEM_RDFCTX package.

This chapter contains the following major sections:

- Information Extractors for Semantically Indexing Documents (page 4-2)
- Extractor Policies (page 4-4)
- Semantically Indexing Documents (page 4-5)
- SEM_CONTAINS and Ancillary Operators (page 4-6)

- Searching for Documents Using SPARQL Query Patterns (page 4-7)
- Bindings for SPARQL Variables in Matching Subgraphs in a Document (SEM_CONTAINS_SELECT Ancillary Operator) (page 4-8)
- Improving the Quality of Document Search Operations (page 4-9)
- Indexing External Documents (page 4-10)
- Configuring the Calais Extractor type (page 4-12)
- Working with General Architecture for Text Engineering (GATE) (page 4-12)
- Creating a New Extractor Type (page 4-13)
- Creating a Local Semantic Index on a Range-Partitioned Table (page 4-14)
- Altering a Semantic Index (page 4-15)
- Passing Extractor-Specific Parameters in CREATE INDEX and ALTER INDEX (page 4-16)
- Performing Document-Centric Inference (page 4-16)
- Metadata Views for Semantic Indexing (page 4-17)
- Default Style Sheet for GATE Extractor Output (page 4-19)

4.1 Information Extractors for Semantically Indexing Documents

Information extractors process unstructured documents and extract meaningful information from them, often using natural-language processing engines with the aid of ontologies. The quality and the completeness of information extracted from a document vary from one extractor to another. Some extractors simply identify the entities (such as names of persons, organizations, and geographic locations from a document), while the others attempt to identify the relationships among the identified entities and additional description for those entities. You can search for a specific document from a large set when the information extracted from the documents is maintained as a semantic index.

You can use an information extractor to create a semantic index on the documents stored in a column of a relational table. An extensible framework allows any third-party information extractor that is accessible from the database to be plugged into the database. An object type created for an extractor encapsulates the extraction logic, and has methods to configure the extractor and receive information extracted from a given document in RDF/XML format.

An abstract type MDSYS.RDFCTX_EXTRACTOR defines the common interfaces to all information extractors. An implementation of this abstract type interacts with a specific information extractor to produce RDF/XML for a given document. An implementation for this type can access a third-party information extractor that either is available as a database application or is installed on the network (accessed using Web service callouts). Example 4-1 (page 4-2) shows the definition of the RDFCTX_EXTRACTOR abstract type.

Example 4-1 RDFCTX_EXTRACTOR Abstract Type Definition

```
create or replace type rdfctx_extractor authid current_user as object (
  extr_type        VARCHAR2(32),
  member function  getDescription return VARCHAR2,
```

```
  member function  rdfReturnType return VARCHAR2,
  member function  getContext(attribute VARCHAR2) return VARCHAR2,
  member procedure startDriver,
  member function  extractRDF(document CLOB,
                              docId    VARCHAR2) return CLOB,
  member function  extractRdf(document CLOB,
                              docId    VARCHAR2,
                              params   VARCHAR2,
                              options  VARCHAR2 default NULL) return CLOB
  member function  batchExtractRdf(docCursor        SYS_REFCURSOR,
                              extracted_info_table  VARCHAR2,
                              params                VARCHAR2,
                              partition_name        VARCHAR2 default NULL,
                              docId                 VARCHAR2 default NULL,
                              preferences           SYS.XMLType default NULL,
                              options               VARCHAR2 default NULL)
                              return CLOB,
  member procedure closeDriver
) not instantiable not final
/
```

A specific implementation of the RDFCTX_EXTRACTOR type sets an identifier for the extractor type in the `extr_type` attribute, and it returns a short description for the extractor type using `getDescription` method. All implementations of this abstract type return the extracted information as RDF triples. In the current release, the RDF triples are expected to be serialized using RDF/XML format, and therefore the `rdfReturnType` method should return '`RDF/XML`'.

An extractor type implementation uses the `extractRDF` method to encapsulate the extraction logic, possibly by invoking external information extractor using proprietary interfaces, and returns the extracted information in RDF/XML format. When a third-party extractor uses some proprietary XML Schema to capture the extracted information, an XML style sheet can be used to generate an equivalent RDF/XML. The `startDriver` and `closeDriver` methods can perform any housekeeping operations pertaining to the information extractor. The optional `params` parameter allows the extractor to obtain additional information about the type of extraction needed (for example, the desired quality of extraction).

Optionally, an extractor type implementation may support a batch interface by providing an implementation of the `batchExtractRdf` member function. This function accepts a cursor through the input parameter `docCursor` and typically uses that cursor to retrieve each document, extract information from the document, and then insert the extracted information into (the specified partition identified by the `partition_name` partition of the `extracted_info_table` table. The `preferences` parameter is used to obtain the preferences value associated with the policy (as described in Indexing External Documents (page 4-10) and in the SEM_RDFCTX.CREATE_POLICY (page 14-2) reference section).

The `getContext` member function accepts an attribute name and returns the value for that attribute. Currently this function is used only for extractors supporting the batch interface. The attribute names and corresponding possible return values are the following:

- For the `BATCH_SUPPORT` attribute, the return values are '`YES`' or '`NO`' depending on whether the extractor supports the batch interface.
- For the `DBUSER` attribute, the return value is the name of a database user that will connect to the database to retrieve rows from the cursor (identified by the

`docCursor` parameter) and that will write to the table `extracted_info_table`.

This information is used for granting appropriate privileges to the table being indexed and the table `extracted_info_table`.

The `startDriver` and `closeDriver` methods can perform any housekeeping operations pertaining to the information extractor.

An extractor type for the General Architecture for Text Engineering (GATE) engine is defined as a subtype of the RDFCTX_EXTRACTOR type. The implementation of this extractor type sends the documents to a GATE engine over a TCP connection, receives annotations extracted by the engine in XML format, and converts this proprietary XML document to an RDF/XML document. For more information on configuring a GATE engine to work with Oracle Database, see Working with General Architecture for Text Engineering (GATE) (page 4-12). For an example of creating a new information extractor, see Creating a New Extractor Type (page 4-13).

Information extractors that are deployed as Web services can be invoked from the database by extending the RDFCTX_WS_EXTRACTOR type, which is a subtype of the RDFCTX_EXTRACTOR type. The RDFCTX_WS_EXTRACTOR type encapsulates the Web service callouts in the `extractRDF` method; specific implementations for network-based extractors can reuse this implementation by setting relevant attribute values in the type constructor.

Thomson Reuters Calais is an example of a network-based information extractor that can be accessed using web-service callouts. The CALAIS_EXTRACTOR type, which is a subtype of the RDFCTX_WS_EXTRACTOR type, encapsulates the Calais extraction logic, and it can be used to semantically index the documents. The CALAIS_EXTRACTOR type must be configured for the database instance before it can be used to create semantic indexes, as explained in Configuring the Calais Extractor type (page 4-12).

4.2 Extractor Policies

An **extractor policy** is a named dictionary entity that determines the characteristics of a semantic index that is created using the policy. Each extractor policy refers, directly or indirectly, to an instance of an extractor type. An extractor policy with a direct reference to an extractor type instance can be used to compose other extractor policies that include additional RDF models for ontologies.

The following example creates a basic extractor policy created using the GATE extractor type:

```
begin
  sem_rdfctx.create_policy (policy_name => 'SEM_EXTR',
                            extractor   => mdsys.gatenlp_extractor());
end;
/
```

The following example creates a dependent extractor policy that combines the metadata extracted by the policy in the preceding example with a user-defined RDF model named `geo_ontology`:

```
begin
  sem_rdfctx.create_policy (policy_name => 'SEM_EXTR_PLUS_GEOONT',
                            base_policy => 'SEM_EXTR',
                            user_models => SEM_MODELS ('geo_ontology'));
end;
/
```

You can use an extractor policy to create one or more semantic indexes on columns that store unstructured documents, as explained in Semantically Indexing Documents (page 4-5).

4.3 Semantically Indexing Documents

Textual documents stored in a CLOB or VARCHAR2 column of a relational table can be indexed using the MDSYS.SEMCONTEXT index type, to facilitate semantically meaningful searches. The extractor policy specified at index creation determines the information extractor used to semantically index the documents. The extracted information, captured as a set of RDF triples for each document, is managed in the semantic data store. Each instance of the semantic index is associated with a system-generated RDF model, which maintains the RDF triples extracted from the corresponding documents.

The following example creates a semantic index named `ArticleIndex` on the textual documents in the ARTICLE column of the NEWSFEED table, using the extractor policy named `SEM_EXTR`:

```
CREATE INDEX ArticleIndex on Newsfeed (article)
   INDEXTYPE IS mdsys.SemContext PARAMETERS ('SEM_EXTR');
```

The RDF model created for an index is managed internally and it is not associated with an application table. The triples stored in such model are automatically maintained for any modifications (such as update, insert, or delete) made to the documents stored in the table column. Although a single RDF model is used to index all documents stored in a table column, the triples stored in the model maintain references to the documents from which they are extracted; therefore, all the triples extracted from a specific document form an individual graph within the RDF model. The documents that are semantically indexed can then be searched using a SPARQL query pattern that operates on the triples extracted from the documents.

When creating a semantic index for documents, you can use a basic extractor policy or a dependent policy, which may include one or more user-defined RDF models. When you create an index with a dependent extractor policy, the document search pattern specified using SPARQL could span the triples extracted from the documents as well as those defined in user-defined models.

You can create an index using multiple extractor policies, in which case the triples extracted by the corresponding extractors are maintained separately in distinct RDF models. A document search query using one such index can select the specific policy to be used for answering the query. For example, an extractor policy named `CITY_EXTR` can be created to extract the names of the cities from a given document, and this extractor policy can be used in combination with the SEM_EXTR policy to create a semantic index, as in the following example:

```
CREATE INDEX ArticleIndex on Newsfeed (article)
   INDEXTYPE IS mdsys.SemContext PARAMETERS ('SEM_EXTR CITY_EXTR');
```

The first extractor policy in the PARAMETERS list is considered to be the default policy if a query does not refer to a specific policy; however, you can change the default extractor policy for a semantic index by using the SEM_RDFCTX.SET_DEFAULT_POLICY (page 14-5) procedure, as in the following example:

```
begin
  sem_rdfctx.set_default_policy (index_name => 'ArticleIndex',
                                 policy_name => 'CITY_EXTR');
```

```
end;
/
```

4.4 SEM_CONTAINS and Ancillary Operators

You can use the SEM_CONTAINS operator in a standard SQL statement to search for documents or document references that are stored in relational tables. This operator has the following syntax:

```
SEM_CONTAINS(
  column   VARCHAR2 / CLOB,
  sparql   VARCHAR2,
  policy   VARCHAR2,
  aliases  SEM_ALIASES,
  index_status  NUMBER,
  ancoper  NUMBER
 ) RETURN NUMBER;
```

The `column` and `sparql` attributes attribute are required. The other attributes are optional (that is, each can be a null value).

The `column` attribute identifies a VARCHAR2 or CLOB column in a relational table that stores the documents or references to documents that are semantically indexed. An index of type MDSYS.SEMCONTEXT must be defined in this column for the SEM_CONTAINS operator to use.

The `sparql` attribute is a string literal that defines the document search criteria, expressed in SPARQL format.

The optional `policy` attribute specifies the name of an extractor policy, usually to override the default policy. A semantic document index can have one or more extractor policies specified at index creation, and one of these policies is the default, which is used if the `policy` attribute is null in the call to SEM_CONTAINS.

The optional `aliases` attribute identifies one or more namespaces, including a default namespace, to be used for expansion of qualified names in the query pattern. Its data type is SEM_ALIASES, which has the following definition: `TABLE OF SEM_ALIAS`, where each SEM_ALIAS element identifies a namespace ID and namespace value. The SEM_ALIAS data type has the following definition: `(namespace_id VARCHAR2(30), namespace_val VARCHAR2(4000))`

The optional `index_status` attribute is relevant only when a dependent policy involving one or more entailments is being used for the SEM_CONTAINS invocation. The `index_status` value identifies the minimum required validity status of the entailments. The possible values are `0` (for VALID, the default), `1` (for INCOMPLETE), and `2` (for INVALID).

The optional `ancoper` attribute specifies a number as the binding to be used when the SEM_CONTAINS_SELECT ancillary operator is used with this operator in a query. The number specified for the `ancoper` attribute should be the same as number specified for the `operbind` attribute in the SEM_CONTAINS_SELECT ancillary operator.

The SEM_CONTAINS operator returns 1 for each document instance matching the specified search criteria, and returns 0 for all other cases.

For more information about using the SEM_CONTAINS operator, including an example, see Searching for Documents Using SPARQL Query Patterns (page 4-7).

4.4.1 SEM_CONTAINS_SELECT Ancillary Operator

You can use the SEM_CONTAINS_SELECT ancillary operator to return additional information about each document that matches some search criteria. This ancillary operator has a single numerical attribute (`operbind`) that associates an instance of the SEM_CONTAINS_SELECT ancillary operator with a SEM_CONTAINS operator by using the same value for the binding. This ancillary operator returns an object of type CLOB that contains the additional information from the matching document, formatted in SPARQL Query Results XML format.

The SEM_CONTAINS_SELECT ancillary operator has the following syntax:

```
SEM_CONTAINS_SELECT(
  operbind  NUMBER
 ) RETURN CLOB;
```

For more information about using the SEM_CONTAINS_SELECT ancillary operator, including examples, see Bindings for SPARQL Variables in Matching Subgraphs in a Document (SEM_CONTAINS_SELECT Ancillary Operator) (page 4-8).

4.4.2 SEM_CONTAINS_COUNT Ancillary Operator

You can use the SEM_CONTAINS_COUNT ancillary operator for a SEM_CONTAINS operator invocation. For each matched document, it returns the count of matching subgraphs for the SPARQL graph pattern specified in the SEM_CONTAINS invocation.

The SEM_CONTAINS_COUNT ancillary operator has the following syntax:

```
SEM_CONTAINS_COUNT(
  operbind  NUMBER
 ) RETURN NUMBER;
```

The following example excerpt shows the use of the SEM_CONTAINS_COUNT ancillary operator to return the count of matching subgraphs for each matched document:

```
SELECT docId, SEM_CONTAINS_COUNT(1) as matching_subgraph_count
FROM   Newsfeed
WHERE  SEM_CONTAINS (article,
  '{ ?org   rdf:type          class:Organization  .
     ?org   pred:hasCategory  cat:BusinessFinance }', ..,
   1)= 1;
```

4.5 Searching for Documents Using SPARQL Query Patterns

Documents that are semantically indexed (that is, indexed using the mdsys.SemContext index type) can be searched using SEM_CONTAINS operator within a standard SQL query. In the query, the SEM_CONTAINS operator must have at least two parameters, the first specifying the column in which the documents are stored and the second specifying the document search criteria expressed as a SPARQL query pattern, as in the following example:

```
SELECT docId FROM Newsfeed
WHERE  SEM_CONTAINS (article,
  '{ ?org  rdf:type  <http://www.example.com/classes/Organization>  .
     ?org  <http://example.com/pred/hasCategory>
             <http://www.example.com/category/BusinessFinance> }'
           )= 1;
```

The SPARQL query pattern specified with the SEM_CONTAINS operator is matched against the individual graphs corresponding to each document, and a document is considered to match a search criterion if the triples from the corresponding graph satisfy the query pattern. In the preceding example, the SPARQL query pattern identifies the individual graphs (thus, the documents) that refer to an `Organization` that belong to `BusinessFinance` category. The SQL query returns the rows corresponding to the matching documents in its result set. The preceding example assumes that the URIs used in the query are generated by the underlying extractor, and that you (the user searching for documents) are aware of the properties and terms that are generated by the extractor in use.

When you create an index using a dependent extractor policy that includes one or more user-defined RDF models, the triples asserted in the user models are considered to be common to all the documents. Document searches involving such policies test the search criteria against the triples in individual graphs corresponding to the documents, combined with the triples in the user models. For example, the following query identifies all articles referring to organizations in the state of New Hampshire, using the geographical ontology (`geo_ontology` RDF Model from a preceding example) that maps cities to states:

```
SELECT docId FROM   Newsfeed
WHERE  SEM_CONTAINS (article,
        '{ ?org     rdf:type          class:Organization  .
           ?org     pred:hasLocation  ?city .
           ?city    geo:hasState      state:NewHampshire }',
        'SEM_EXTR_PLUS_GEOONT',
               sem_aliases(
                  sem_alias('class', 'http://www.myorg.com/classes/'),
                  sem_alias('pred', 'http://www.myorg.com/pred/'),
                  sem_alias('geo', 'http://geoont.org/rel/'),
                  sem_alias('state', 'http://geoont.org/state/'))) = 1;
```

The preceding query, with a reference to the extractor policy SEM_EXTR_PLUS_GEOONT (created in an example in Extractor Policies (page 4-4)), combines the triples extracted from the indexed documents and the triples in the user model to find matching documents. In this example, the name of the extractor policy is optional if the corresponding index is created with just this policy or if this is the default extractor policy for the index. When the query pattern uses some qualified names, an optional parameter to the SEM_CONTAINS operator can specify the namespaces to be used for expanding the qualified names.

SPARQL-based document searches can make use of the SPARQL syntax that is supported through SEM_MATCH queries.

4.6 Bindings for SPARQL Variables in Matching Subgraphs in a Document (SEM_CONTAINS_SELECT Ancillary Operator)

You can use the SEM_CONTAINS_SELECT ancillary operator to return additional information about each document matched using the SEM_CONTAINS operator. Specifically, the bindings for the variables used in SPARQL-based document search criteria can be returned using this operator. This operator is ancillary to the SEM_CONTAINS operator, and a literal number is used as an argument to this operator to associate it with a specific instance of SEM_CONTAINS operator, as in the following example:

```
SELECT docId, SEM_CONTAINS_SELECT(1) as result
FROM   Newsfeed
WHERE  SEM_CONTAINS (article,
```

```
  '{ ?org   rdf:type          class:Organization  .
    ?org   pred:hasCategory  cat:BusinessFinance }', ..,
   1)= 1;
```

The SEM_CONTAINS_SELECT ancillary operator returns the bindings for the variables in SPARQL Query Results XML format, as CLOB data. The variables may be bound to multiple data instances from a single document, in which case all bindings for the variables are returned. The following example is an excerpt from the output of the preceding query: a value returned by the SEM_CONTAINS_SELECT ancillary operator for a document matching the specified search criteria.

```
<results>
  <result>
     <binding name="ORG">
        <uri>http://newscorp.com/Org/AcmeCorp</uri>
     </binding>
  </result>
  <result>
     <binding name="ORG">
       <uri>http://newscorp.com/Org/ABCCorp</uri>
     </binding>
  </result>
</results>
```

You can rank the search results by creating an instance of XMLType for the CLOB value returned by the SEM_CONTAINS_SELECT ancillary operator and applying an XPath expression to sort the results on some attribute values.

By default, the SEM_CONTAINS_SELECT ancillary operator returns bindings for all variables used in the SPARQL-based document search criteria. However, when the values for only a subset of the variables are relevant for a search, the SPARQL pattern can include a SELECT clause with space-separated list of variables for which the values should be returned, as in the following example:

```
SELECT docId, SEM_CONTAINS_SELECT(1) as result
FROM   Newsfeed
WHERE  SEM_CONTAINS (article,
        'SELECT ?org  ?city
         WHERE { ?org     rdf:type          class:Organization  .
                 ?org     pred:hasLocation  ?city .
                 ?city    geo:hasState      state:NewHampshire }', ..,
        1) = 1;
```

4.7 Improving the Quality of Document Search Operations

The quality of a document search operation depends on the quality of the information produced by the extractor used to index the documents. If the information extracted is incomplete, you may want to add some annotations to a document. You can use the SEM_RDFCTX.MAINTAIN_TRIPLES (page 14-4) procedure to add annotations, in the form of RDF triples, to specific documents in order to improve the quality of search, as shown in the following example:

```
begin
  sem_rdfctx.maintain_triples(
     index_name      => 'ArticleIndex',
     where_clause    => 'docid in (1,15,20)',
     rdfxml_content => sys.xmltype(
      '<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
                xmlns:pred="http://example.com/pred/">
```

```
      <rdf:Description rdf:about=" http://newscorp.com/Org/ExampleCorp">
        <pred:hasShortName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            Example
        </pred:hasShortName>
     </rdf:Description>
    </rdf:RDF>'));
end;
/
```

The index name and the WHERE clause specified in the preceding example identify specific instances of the document to be annotated, and the RDF/XML content passed in is used to add additional triples to the individual graphs corresponding to those documents. This allows domain experts and user communities to improve the quality of search by adding relevant triples to annotate some documents.

4.8 Indexing External Documents

You can use semantic indexing on documents that are stored in a file system or on the network. In such cases, you store the references to external documents in a table column, and you create a semantic index on the column using an appropriate extractor policy.

To index external documents, define an extractor policy with appropriate preferences, using an XML document that is assigned to the `preferences` parameter of the SEM_RDFCTX.CREATE_POLICY (page 14-2) procedure, as in the following example:

```
begin
  sem_rdfctx.create_policy (
       policy_name => 'SEM_EXTR_FROM_FILE',
       extractor   => mdsys.gatenlp_extractor()),
       preferences => sys.xmltype('<RDFCTXPreferences>
                                     <Datastore type="FILE">
                                        <Path>EXTFILES_DIR</Path>
                                     </Datastore>
                                   </RDFCTXPreferences>'));
end;
/
```

The `<Datastore>` element in the preferences document specifies the type of repository used for the documents to be indexed. When the value for the `type` attribute is set to `FILE`, the `<Path>` element identifies a directory object in the database (created using the SQL statement CREATE DIRECTORY). A table column indexed using the specified extractor policy is expected to contain relative paths to individual files within the directory object, as shown in the following example:

```
CREATE TABLE newsfeed (docid       number,
                       articleLoc  VARCHAR2(100));
INSERT INTO into newsfeed (docid, articleLoc) values
                    (1, 'article1.txt');
INSERT INTO newsfeed (docid, articleLoc) values
                    (2, 'folder/article2.txt');

CREATE INDEX ArticleIndex on newsfeed (articleLoc)
   INDEXTYPE IS mdsys.SemContext PARAMETERS ('SEM_EXTR_FROM_FILE');
```

To index documents that are accessed using HTTP protocol, create a extractor policy with preferences that set the `type` attribute of the `<Datastore>` element to `URL` and that list one or more hosts in the `<Path>` elements, as shown in the following excerpt:

```
<RDFCTXPreferences>
   <Datastore type="URL">
       <Path>http://cnn.com</Path>
       <Path>http://abc.com</Path>
   </Datastore>
</RDFCTXPreferences>
```

The schema in which a semantic index for external documents is created must have the necessary privileges to access the external objects, including access to any proxy server used to access documents outside the firewall, as shown in the following example:

```
-- Grant read access to the directory object for FILE data store --
grant read on directory EXTFILES_DIR to SEMUSR;

-- Grant connect access to set of hosts for URL data store --
begin
  dbms_network_acl_admin.create_acl (
                acl          => 'network_docs.xml',
                description  => 'Normal Access',
                principal    => 'SEMUSR',
                is_grant     => TRUE,
                privilege    => 'connect');
end;
/

begin
  dbms_network_acl_admin.assign_acl (
               acl        => 'network_docs.xml',
               host       =>  'cnn.com',
               lower_port => 1,
               upper_port => 10000);
end;
/
```

External documents that are semantically indexed in the database may be in one of the well-known formats such as Microsoft Word, RTF, and PDF. This takes advantage of the Oracle Text capability to extract plain text version from formatted documents using filters (see the CTX_DOC.POLICY_FILTER procedure, described in *Oracle Text Reference*). To semantically index formatted documents, you must specify the name of a CTX policy in the extractor preferences, as shown in the following excerpt:

```
<RDFCTXPreferences>
   <Datastore type="FILE" filter="CTX_FILTER_POLICY">
       <Path>EXTFILES_DIR</Path>
   </Datastore>
</RDFCTXPreferences>
```

In the preceding example, the `CTX_FILTER_POLICY` policy, created using the CTX_DDL.CREATE_POLICY procedure, must exist in your schema. The table columns that are semantically indexed using this preferences document can store paths to formatted documents, from which plain text is extracted using the specified CTX policy. The information extractor associated with the extractor policy then processes the plain text further, to extract the semantics in RDF/XML format.

4.9 Configuring the Calais Extractor type

The CALAIS_EXTRACTOR type, which is a subtype of the RDFCTX_WS_EXTRACTOR type, enables you to access a Web service end point anywhere on the network, including the one that is publicly accessible (`OpenCalais.com`). To do so, you must connect with SYSDBA privileges and configure the Calais extractor type with Web service end point, the SOAP action, and the license key by setting corresponding parameters, as shown in the following example:

```
begin
  sem_rdfctx.set_extractor_param (
     param_key   => 'CALAIS_WS_ENDPOINT',
     param_value => 'http://api1.opencalais.com/enlighten/calais.asmx',
     param_desc  => 'Calais web service end-point');

  sem_rdfctx.set_extractor_param (
     param_key   => 'CALAIS_KEY',
     param_value => '<Calais license key goes here>',
     param_desc  => 'Calais extractor license key');

  sem_rdfctx.set_extractor_param (
     param_key   => 'CALAIS_WS_SOAPACTION',
     param_value => 'http://clearforest.com/Enlighten',
     param_desc  => 'Calais web service SOAP Action');
end;
```

To enable access to a Web service outside the firewall, you must also set the parameter for the proxy host, as in the following example:

```
begin
  sem_rdfctx.set_extractor_param (
      param_key   => 'HTTP_PROXY',
      param_value => 'www-proxy.acme.com',
      param_desc  => 'Proxy server');
end;
```

4.10 Working with General Architecture for Text Engineering (GATE)

General Architecture for Text Engineering (GATE) is an open source natural language processor and information extractor (see `http://gate.ac.uk`). You can use GATE to perform semantic indexing of documents stored in the database. The extractor type `mdsys.gatenlp_extractor` is defined as a subtype of the RDFCTX_EXTRACTOR type. The implementation of this extractor type sends an unstructured document to a GATE engine over a TCP connection, receives corresponding annotations, and converts them into RDF following a user-specified XML style sheet.

The requests for information extraction are handled by a server socket implementation, which instantiates the GATE components and listens to extraction requests at a pre-determined port. The host and the post for the GATE listener are recorded in the database, as shown in the following example, for all instances of the `mdsys.gatenlp_extractor` type to use.

```
begin
  sem_rdfctx.set_extractor_param (
     param_key   => 'GATE_NLP_HOST',
     param_value => 'gateserver.acme.com',
     param_desc  => 'Host for GATE NLP Listener ');
```

```
  sem_rdfctx.set_extractor_param (
     param_key   => 'GATE_NLP_PORT',
     param_value => '7687',
     param_desc  => 'Port for Gate NLP Listener');
end;
```

The server socket application receives an unstructured document and constructs an annotation set with the desired types of annotations. Each annotation in the set may be customized to include additional features, such as the relevant phrase from the input document and some domain specific features. The resulting annotation set is serialized into XML (using the `annotationSetToXml` method in the `gate.corpora.DocumentXmlUtils` Java package) and returned back to the socket client.

A sample Java implementation for the GATE listener is available for download from the code samples and examples page on OTN (see Semantic Data Examples (PL/SQL and Java) (page 1-93) for information about this page).

The `mdsys.gatenlp_extractor` implementation in the database receives the annotation set encoded in XML, and converts it to RDF/XML using an XML style sheet. You can replace the default style sheet (listed in Default Style Sheet for GATE Extractor Output (page 4-19)) used by the `mdsys.gatenlp_extractor` implementation with a custom style sheet when you instantiate the type.

The following example creates an extractor policy that uses a custom style sheet to generate RDF from the annotation set produced by the GATE extractor:

```
begin
  sem_rdfctx.create_policy (policy_name => 'GATE_EXTR',
                            extractor   => mdsys.gatenlp_extractor(
      sys.XMLType('<?xml version="1.0"?>
                 <xsl:stylesheet version="2.0"
                    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
                   ..
                 </xsl:stylesheet>')));
end;
/
```

4.11 Creating a New Extractor Type

You can create a new extractor type by extending the RDFCTX_EXTRACTOR or RDFCTX_WS_EXTRACTOR extractor type. The extractor type to be extended must be accessible using Web service calls. The schema in which the new extractor type is created must be granted additional privileges to allow creation of the subtype. For example, if a new extractor type is created in the schema RDFCTXU, you must enter the following commands to grant the UNDER and RDFCTX_ADMIN privileges to that schema:

```
GRANT under ON mdsys.rdfctx_extractor TO rdfctxu;
GRANT rdfctx_admin TO rdfctxu;
```

As an example, assume that an information extractor can process an incoming document and return an XML document that contains extracted information. To enable the information extractor to be invoked using a PL/SQL wrapper, you can create the corresponding extractor type implementation, as in the following example:

```
create or replace type rdfctxu.info_extractor under rdfctx_extractor (
  xsl_trans   sys.XMLtype,
  constructor function info_extractor (
```

```
                xsl_trans  sys.XMLType ) return self as result,
  overriding member function getDescription return VARCHAR2,
  overriding member function rdfReturnType return VARCHAR2,
  overriding member function extractRDF(document CLOB,
                                        docId    VARCHAR2) return CLOB
)
/

create or replace type body rdfctxu.info_extractor as
  constructor function info_extractor (
                xsl_trans  sys.XMLType ) return self as result is
  begin
    self.extr_type := 'Info Extractor Inc.';
    -- XML style sheet to generate RDF/XML from proprietary XML documents
    self.xsl_trans := xsl_trans;
    return;
  end info_extractor;

  overriding member function getDescription return VARCHAR2 is
  begin
    return 'Extactor by Info Extractor Inc.';
  end getDescription;

  overriding member function rdfReturnType return VARCHAR2 is
  begin
    return 'RDF/XML';
  end rdfReturnType;

  overriding member function extractRDF(document CLOB,
                                        docId    VARCHAR2) return CLOB is
    ce_xmlt  sys.xmltype;
  begin
    EXECUTE IMMEDIATE
      'begin :1 = info_extract_xml(doc => :2); end;'
       USING IN OUT ce_xmlt, IN document;

    -- Now pass the ce_xmlt through RDF/XML transformation --
    return ce_xmlt.transform(self.xsl_trans).getClobVal();
  end extractRdf;

end;
```

In the preceding example:

- The implementation for the created `info_extractor` extractor type relies on the XML style sheet, set in the constructor, to generate RDF/XML from the proprietary XML schema used by the underlying information extractor.
- The `extractRDF` function assumes that the `info_extract_xml` function contacts the desired information extractor and returns an XML document with the information extracted from the document that was passed in.
- The XML style sheet is applied on the XML document to generate equivalent RDF/XML, which is returned by the `extractRDF` function.

4.12 Creating a Local Semantic Index on a Range-Partitioned Table

A local index can be created on a VARCHAR2 or CLOB column of a range-partitioned table by using the following syntax:

```
CREATE INDEX <index-name> … LOCAL;
```

The following example creates a range-partitioned table and a local semantic index on that table:

```
CREATE TABLE part_newsfeed (
  docid number, article CLOB, cdate DATE)
partition by range (cdate)
(partition p1 values less than (to_date('01-Jan-2001')),
 partition p2 values less than (to_date('01-Jan-2004')),
 partition p3 values less than (to_date('01-Jan-2008')),
 partition p4 values less than (to_date('01-Jan-2012'))
);

CREATE INDEX ArticleLocalIndex on part_newsfeed (article)
   INDEXTYPE IS mdsys.SemContext PARAMETERS ('SEM_EXTR')
LOCAL;
```

Note that every partition of the local semantic index will have content generated for the same set of policies. When you use the ALTER INDEX statement on a local index to add or drop policies associated with a semantic index partition, you should try to keep the same set of policies associated with each partition. You can achieve this result by using ALTER INDEX statements in a loop over the set of partitions. (For more information about altering semantic indexes, see Altering a Semantic Index (page 4-15),)

4.13 Altering a Semantic Index

This section discusses using the ALTER INDEX statement with a semantic index. For a local semantic index, the ALTER INDEX statement applies to a specified partition. The general syntax of the ALTER INDEX command for a semantic index is as follows:

```
ALTER INDEX <index-name> REBUILD [PARTITION <index-partition-name>]
  [PARAMETERS ('-<action_for_policy> <policy-name>')];
```

4.13.1 Rebuilding Content for All Existing Policies in a Semantic Index

If the PARAMETERS clause is not included in the ALTER INDEX statement, the content of the semantic index (or index partition) is rebuilt for every policy presently associated with the index. The following are two examples:

```
ALTER INDEX ArticleIndex REBUILD;
ALTER INDEX ArticleLocalIndex REBUILD PARTITION p1;
```

4.13.2 Rebuilding to Add Content for a New Policy to a Semantic Index

Using `add_policy` for *<action_for_policy>*, you can add content for a new base policy or a dependent policy to a semantic index (or index partition). If a dependent policy is being added and if its base policy is not already a part of the index, then content for the base policy is also added implicitly (by invoking the extractor specified as part of the base policy definition). The following is an example:

```
ALTER INDEX ArticleIndex REBUILD PARAMETERS ('-add_policy MY_POLICY');
```

4.13.3 Rebuilding Content for an Existing Policy from a Semantic Index

Using `rebuild_policy` for *<action_for_policy>*, you can rebuild the content of the semantic index (or index partition) for an existing policy presently associated with the index. The following is an example:

```
ALTER INDEX ArticleIndex REBUILD PARAMETERS ('-rebuild_policy MY_POLICY');
```

4.13.4 Rebuilding to Drop Content for an Existing Policy from a Semantic Index

Using `drop_policy` for *<action_for_policy>*, you can drop content corresponding to an existing base policy or a dependent policy from a semantic index (or index partition). Note that dropping the content for a base policy will fail if it is the only policy for the index (or index partition) or if it is used by dependent policies associated with this index (or index partition).

The following example drops the content for a policy from an index:

```
ALTER INDEX ArticleIndex REBUILD PARAMETERS ('-drop_policy MY_POLICY');
```

4.14 Passing Extractor-Specific Parameters in CREATE INDEX and ALTER INDEX

The CREATE INDEX and ALTER INDEX statements allow the passing of parameters needed by extractors. These parameters are passed on to the extractor using the `params` parameter of the `extractRdf` and `batchExtractRdf` methods. The following two examples show their use:

```
CREATE INDEX ArticleIndex on Newsfeed (article)
  INDEXTYPE IS mdsys.SemContext PARAMETERS ('SEM_EXTR=(NE_ONLY)');

ALTER INDEX ArticleIndex REBUILD
  PARAMETERS ('-add_policy MY_POLICY=(NE_ONLY)');
```

4.15 Performing Document-Centric Inference

Document-centric inference refers to the ability to infer from each document individually. It does not allow triples extracted from two different documents to be used together for inference. It contrasts with the more common corpus-centric inference, where new triples can be inferred from combinations of triples extracted from multiple documents.

Document-centric inference can be desirable in document search applications because inclusion of a document in the search result is based on the extracted and/or inferred triples for that document only, that is, triples extracted and/or inferred from any other documents in the corpus do not play any role in the selection of this document. (Document-centric inference might be preferred, for example, if there is inconsistency among documents because of differences in the reliability of the data or in the biases of the document creators.)

To perform document-centric inference, use named graph based local inference (explained in Named Graph Based Local Inference (NGLI) (page 2-16)) by specifying `options => 'LOCAL_NG_INF=T'` in the call to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure.

Entailments created through document-centric inference can be included as content of a semantic index by creating a dependent policy and adding that policy to the semantic index, as shown in Example 4-2 (page 4-16).

Example 4-2 Using Document-Centric Inference

```
-- Create entailment 'extr_data_inf' using document-centric inference
-- assuming:
--   model_name for semantic index based on base policy: 'RDFCTX_MOD_1'
--    (model name is available from the RDFCTX_INDEX_POLICIES view;
```

```
--     see RDFCTX_INDEX_POLICIES View (page 4-18))
--   ontology: dataOntology
--   rulebase: OWL2RL
-- options: 'LOCAL_NG_INF=T' (for document-centric inference)
BEGIN
sem_apis.create_entailment('extr_data_inf',
  models_in    => sem_models('RDFCTX_MOD_1', 'dataOntology'),
  rulebases_in => sem_rulebases('OWL2RL'),
  options      => 'LOCAL_NG_INF=T');
END;
/
-- Create a dependent policy to augment data extracted using base policy
-- with content of entailment extr_data_inf (computed in previous statement)
BEGIN
sem_rdfctx.create_policy (
  policy_name => 'SEM_EXTR_PLUS_DATA_INF',
  base_policy => 'SEM_EXTR',
  user_models => NULL,
  user_entailments => sem_models('extr_data_inf'));
END;
/
-- Add the dependent policy to the ARTICLEINDEX index.
EXECUTE sem_rdfctx.add_dependent_policy('ARTICLEINDEX','SEM_EXTR_PLUS_DATA_INF');
```

4.16 Metadata Views for Semantic Indexing

This section describes views that contain metadata about semantic indexing.

4.16.1 MDSYS.RDFCTX_POLICIES View

Information about extractor policies defined in the current schema is maintained in the MDSYS.RDFCTX_POLICIES view, which has the columns shown in Table 4-1 (page 4-17) and one row for each extractor policy.

Table 4-1 MDSYS.RDFCTX_POLICIES View Columns

Column Name	Data Type	Description
POLICY_OWNER	VARCHAR2(32)	Owner of the extractor policy
POLICY_NAME	VARCHAR2(32)	Name of the extractor policy
EXTRACTOR	MDSYS.RDFCTX_EXTRACTOR	Instance of extractor type
IS_DEPENDENT	VARCHAR2(3)	Contains `YES` if the extractor policy is dependent on a base policy; contains `NO` if the extractor policy is not dependent on a base policy.
BASE_POLICY	VARCHAR2(32)	For a dependent policy, the name of the base policy
USER_MODELS	MDSYS.RDF_MODELS	For a dependent policy, a list of the RDF models included in the policy

4.16.2 RDFCTX_INDEX_POLICIES View

Information about semantic indexes defined in the current schema and the extractor policies used to create the index is maintained in the MDSYS.RDFCTX_POLICIES view, which has the columns shown in Table 4-2 (page 4-18) and one row for each combination of semantic index and extractor policy.

Table 4-2 MDSYS.RDFCTX_INDEX_POLICIES View Columns

Column Name	Data Type	Description
INDEX_OWNER	VARCHAR2(32)	Owner of the semantic index
INDEX_NAME	VARCHAR2(32)	Name of the semantic index
INDEX_PARTITION	VARCHAR2(32)	Name of the index partition (for LOCAL index only)
POLICY_NAME	VARCHAR2(32)	Name of the extractor policy
EXTR_PARAMETERS	VARCHAR2(100)	Parameters specified for the extractor
IS_DEFAULT	VARCHAR2(3)	Contains `YES` if POLICY_NAME is the default extractor policy for the index; contains `NO` if POLICY_NAME is not the default extractor policy for the index.
STATUS	VARCHAR2(10)	Contains `VALID` if the index is valid, `INPROGRESS` if the index is being created, or `FAILED` if a system failure occurred during the creation of the index.
RDF_MODEL	VARCHAR2(32)	Name of the RDF model maintaining the index data

4.16.3 RDFCTX_INDEX_EXCEPTIONS View

Information about exceptions encountered while creating or maintaining semantic indexes in the current schema is maintained in the MDSYS.RDFCTX_INDEX_EXCEPTIONS view, which has the columns shown in Table 4-3 (page 4-18) and one row for each exception.

Table 4-3 MDSYS.RDFCTX_INDEX_EXCEPTIONS View Columns

Column Name	Data Type	Description
INDEX_OWNER	VARCHAR2(32)	Owner of the semantic index associated with the exception
INDEX_NAME	VARCHAR2(32)	Name of the semantic index associated with the exception

Table 4-3 (Cont.) MDSYS.RDFCTX_INDEX_EXCEPTIONS View Columns

Column Name	Data Type	Description
POLICY_NAME	VARCHAR2(32)	Name of the extractor policy associated with the exception
DOC_IDENTIFIER	VARCHAR2(38)	Row identifier (rowid) of the document associated with the exception
EXCEPTION_TYPE	VARCHAR2(13)	Type of exception
EXCEPTION_CODE	NUMBER	Error code associated with the exception
EXCEPTION_TEXT	CLOB	Text associated with the exception
EXTRACTED_AT	TIMESTAMP	Time at which the exception occurred

4.17 Default Style Sheet for GATE Extractor Output

This section lists the default XML style sheet that the `mdsys.gatenlp_extractor` implementation uses to convert the annotation set (encoded in XML) into RDF/XML. (This extractor is explained in Working with General Architecture for Text Engineering (GATE) (page 4-12).)

```
<?xml version="1.0"?>
  <xsl:stylesheet version="2.0"
                  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
    <xsl:output encoding="utf-8" indent="yes"/>
    <xsl:param name="docbase">http://xmlns.oracle.com/rdfctx/</xsl:param>
    <xsl:param name="docident">0</xsl:param>
    <xsl:param name="classpfx">
      <xsl:value-of select="$docbase"/>
      <xsl:text>class/</xsl:text>
    </xsl:param>
    <xsl:template match="/">
       <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
                xmlns:owl="http://www.w3.org/2002/07/owl#"
                xmlns:prop="http://xmlns.oracle.com/rdfctx/property/">
       <xsl:for-each select="AnnotationSet/Annotation">
         <rdf:Description>
           <xsl:attribute name="rdf:about">
             <xsl:value-of select="$docbase"/>
             <xsl:text>docref/</xsl:text>
             <xsl:value-of select="$docident"/>
             <xsl:text>/</xsl:text>
             <xsl:value-of select="@Id"/>
           </xsl:attribute>
           <xsl:for-each select="./Feature">
             <xsl:choose>
               <xsl:when test="./Name[text()='majorType']">
                 <rdf:type>
                   <xsl:attribute name="rdf:resource">
                      <xsl:value-of select="$classpfx"/>
```

```
          <xsl:text>major/</xsl:text>
          <xsl:value-of select="translate(./Value/text(),
                                          ' ', '#')"/>
        </xsl:attribute>
      </rdf:type>
    </xsl:when>
    <xsl:when test="./Name[text()='minorType']">
      <xsl:element name="prop:hasMinorType">
        <xsl:attribute name="rdf:resource">
          <xsl:value-of select="$docbase"/>
          <xsl:text>minorType/</xsl:text>
          <xsl:value-of select="translate(./Value/text(),
                                          ' ', '#')"/>
        </xsl:attribute>
      </xsl:element>
    </xsl:when>
    <xsl:when test="./Name[text()='kind']">
      <xsl:element name="prop:hasKind">
        <xsl:attribute name="rdf:resource">
          <xsl:value-of select="$docbase"/>
          <xsl:text>kind/</xsl:text>
          <xsl:value-of select="translate(./Value/text(),
                                          ' ', '#')"/>
        </xsl:attribute>
      </xsl:element>
    </xsl:when>
    <xsl:when test="./Name[text()='locType']">
      <xsl:element name="prop:hasLocType">
        <xsl:attribute name="rdf:resource">
          <xsl:value-of select="$docbase"/>
          <xsl:text>locType/</xsl:text>
          <xsl:value-of select="translate(./Value/text(),
                                          ' ', '#')"/>
        </xsl:attribute>
      </xsl:element>
    </xsl:when>
    <xsl:when test="./Name[text()='entityValue']">
      <xsl:element name="prop:hasEntityValue">
        <xsl:attribute name="rdf:datatype">
          <xsl:text>
            http://www.w3.org/2001/XMLSchema#string
          </xsl:text>
        </xsl:attribute>
        <xsl:value-of select="./Value/text()"/>
      </xsl:element>
    </xsl:when>
    <xsl:otherwise>
      <xsl:element name="prop:has{translate(
                           substring(./Name/text(),1,1),
                           'abcdefghijklmnopqrstuvwxyz',
                           'ABCDEFGHIJKLMNOPQRSTUVWXYZ')}{
                         substring(./Name/text(),2)}">
        <xsl:attribute name="rdf:datatype">
          <xsl:text>
            http://www.w3.org/2001/XMLSchema#string
          </xsl:text>
        </xsl:attribute>
        <xsl:value-of select="./Value/text()"/>
      </xsl:element>
    </xsl:otherwise>
  </xsl:choose>
```

```
          </xsl:for-each>
        </rdf:Description>
      </xsl:for-each>
      </rdf:RDF>
    </xsl:template>
</xsl:stylesheet>
```

5

Fine-Grained Access Control for RDF Data

The default control of access to the Oracle Database semantic data store is at the model level: the owner of a model can grant select, delete, and insert privileges on the model to other users by granting appropriate privileges on the view named RDFM_<*model_name*>. However, for applications with stringent security requirements, you can enforce a fine-grained access control mechanism by using the Oracle Label Security option of Oracle Database:

Oracle Label Security (OLS) for RDF data allows sensitivity labels to be associated with individual triples stored in an RDF model. For each query, access to specific triples is granted by comparing their labels with the user's session labels. Furthermore, a minimum sensitivity label for all triple describing a specific resource or all triples defined with a specific predicate can be enforced by assigning a sensitivity label directly to the resource or the predicate, respectively.

For information about using OLS, see *Oracle Label Security Administrator's Guide*.

Note:

Effective with Oracle Database Release 12*c* (12.1), Virtual Private Database (VPD) support in RDF Semantic Graph is deprecated for providing fine-grained access control, and will be removed in an upcoming major release. (Meanwhile, Virtual Private Database Support in RDF Semantic Graph (page C-1) contains information about this deprecated support.)

You should not develop new RDF Semantic Graph applications that depend on VPD, and you should transition existing RDF Semantic Graph applications that depend on VPD to use Oracle Label Security (OLS) instead.

For more information, see My Oracle Support Note 1468273.1.

Oracle Label Security (OLS) for RDF data provides two options for securing semantic data:

- Triple-level security (explained in Triple-Level Security (page 5-2)), which is highly recommended for its performance and ease of use
- Resource-level security (explained in Resource-Level Security (page 5-11)), which is generally not recommended

To specify an option, use the SEM_RDFSA.APPLY_OLS_POLICY (page 15-3) procedure with the appropriate `rdfsa_options` parameter value.

To switch from one option to the other, remove the existing policy by using the SEM_RDFSA.REMOVE_OLS_POLICY (page 15-12) procedure, and then apply the new policy by using the SEM_RDFSA.APPLY_OLS_POLICY (page 15-3) procedure with the appropriate `rdfsa_options` parameter value.

5.1 Triple-Level Security

The triple-level security option provides a thin layer of RDF-specific capabilities on top of the Oracle Database native support for label security. This option provides better performance and is easier to use than the resource-level security (described in Resource-Level Security (page 5-11)), especially for performing inference while using OLS. The main difference is that with triple-level security there is no need to assign labels, explicitly or implicitly, to individual triple resources (subjects, properties, objects).

To use triple-level security, specify `SEM_RDFSA.TRIPLE_LEVEL_ONLY` as the `rdfsa_options` parameter value when you execute the SEM_RDFSA.APPLY_OLS_POLICY (page 15-3) procedure. For example:

```
EXECUTE sem_rdfsa.apply_ols_policy('defense', SEM_RDFSA.TRIPLE_LEVEL_ONLY);
```

Do not specify any of the other available parameters for the SEM_RDFSA.APPLY_OLS_POLICY (page 15-3) procedure.

When you use triple-level security, OLS is applied to each semantic model in the network. That is, label security is applied to the relevant internal tables and to all the application tables; there is no need to manually apply policies to the application tables of existing semantic models. However, if you need to create additional models after applying the OLS policy, you must use the SEM_OLS.APPLY_POLICY_TO_APP_TAB (page 12-1) procedure to apply OLS to the application table before creating the model. Similarly, if you have dropped a semantic model and you no longer need to protect the application table, you can use the SEM_OLS.REMOVE_POLICY_FROM_APP_TAB (page 12-2) procedure. (These procedures are described in SEM_OLS Package Subprograms (page 12-1).)

With triple-level security, duplicate triples with different labels can be inserted in the semantic model. (Such duplicates are not allowed with resource-level security.) For example, assume that you have a triple with a very sensitive label, such as:

```
(<urn:X>,<urn:P>,<urn:Y>, "TOPSECRET")
```

This does not prevent a low-privileged (`UNCLASSIFIED`) user from inserting the triple `(<urn:X>,<urn:P>,<urn:Y>, "UNCLASSIFIED")`. Because SPARQL and SEM_MATCH do not return label information, a query will return both rows (assuming the user has appropriate privileges), and it will not be easy to distinguish between the `TOPSECRET` and `UNCLASSIFIED` triples.

To filter out such low-security triples when querying the semantic models, you can one or more the following options with SEM_MATCH:

- `POLICY_NAME` specifies the OLS policy name.
- `MIN_LABEL` specifies the minimum label for triples that are included in the query

In other words, every triple that contains a label that is strictly dominated by `MIN_LABEL` is not included in the query. For example, to filter out the "UNCLASSIFIED" triple, you could use the following query (assuming the OLS policy name is `DEFENSE` and that the query user has read privileges over `UNCLASSIFIED` and `TOPSECRET` triples):

```
SELECT s,p,y FROM table(sem_match('{?s ?p ?y}' ,
  sem_models(TEST'), null, null, null, null,
  'MIN_LABEL=TOPSECRET POLICY_NAME=DEFENSE'));
```

Note that the filtering in the preceding example occurs in addition to the security checks performed by the native OLS software.

After a triple has been inserted, you can view and update the label information through the `CTXT1` column in the application table for the semantic model (assuming that you have the `WRITEUP` and `WRITEDOWN` privileges to modify the labels).

There are no restrictions on who can perform inference or bulk loading with triple-level security; all of the inferred or bulk loaded triples are inserted with the user's session row label. Note that you can change the session labels by using the SA_UTL package. (For more information, see *Oracle Label Security Administrator's Guide*.)

5.1.1 Fine-Grained Security for Inferred Data and Ladder-Based Inference (LBI)

When triple-level security is turned on for RDF data stored in Oracle Database, asserted facts are tagged with data labels to enforce mandatory access control. In addition, when a user invokes the forward-chaining based inference function through the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, the newly inferred relationships will be tagged with the current row label (`SA_UTL.NUMERIC_ROW_LABEL`).

These newly inferred relationships are derived solely based on the information that the user is allowed to access. These relationships do, however, share the same data label. This is understandable because a SEM_APIS.CREATE_ENTAILMENT (page 11-15) call can be viewed as a three-step process: read operation, followed by a logical inference computation, followed by a write operation. The read operation gathers information upon which inference computation is based, and it is restricted by access privileges, the user's label, and the data labels; the logical inference computation step is purely mathematical; and the final write of inferred information into the entailed graph is no different from the same user asserting some new facts (which happen to be calculated by the previous step).

Having all inferred assertions tagged with a single label is sufficient if a user only owns a single label. It is, however, not fine-grained enough when there are multiple labels owned by the same user, which is a common situation in a multitenancy setup.

For example, assume a user sets its user label and data label as `TopSecret`, invokes SEM_APIS.CREATE_ENTAILMENT (page 11-15), switches to a weaker label named `Secret`, and finally performs a SPARQL query. The query will not be able to see any of those newly inferred relationships because they were all tagged with the `TopSecret` label. However, if the user switches back to the `TopSecret` label, now every single inferred relationship is visible. It is "all or nothing" (that is, all visible or nothing visible) as far as inferred relationships are concerned.

When multiple labels are available for use by a given user, you normally want to assign different labels to different inferred relationships. There are two ways to achieve this goal:

- Invoking SEM_APIS.CREATE_ENTAILMENT Multiple Times (page 5-3)
- Using Ladder-Based Inference (LBI) (page 5-4)

Ladder-based inference, effective with Oracle Database 12*c* Release 1 (12.1), is probably the simpler and more convenient of the two approaches.

Invoking SEM_APIS.CREATE_ENTAILMENT Multiple Times

Assume a security policy named DEFENSE, a user named SCOTT, and a sequence of user labels Label1, Label2,..., Label*n* owned by SCOTT. The following call by SCOTT

sets the label as Label1, runs the inference for the first time, and tags the newly inferred triples with Label1:

```
EXECUTE sa_utl.set_label('defense',char_to_label('defense','Label1'));
EXECUTE sa_utl.set_row_label('defense',char_to_label('defense','Label1'));
EXECUTE sem_apis.create_entailment('inf', sem_models('contracts'),
sem_rulebases('owlprime'), SEM_APIS.REACH_CLOSURE, null,'');
```

Now, SCOTT switches the label to Label2, runs the inference a second time, and tags the newly inferred triples with Label2. Obviously, if Label2 is dominated by Label1, then no new triples will be inferred because Label2 cannot see anything beyond what Label1 is allowed to see. If Label2 is not dominated by Label1, the read step of the inference process will probably see a different set of triples, and consequently the inference call can produce some new triples, which will in turn be tagged with Label2.

For the purpose of this example, assume the following condition holds true: for any 1 <= *i* < *j* <= *n*, Label*j* is not dominated by Label*i*.

```
EXECUTE sa_utl.set_label('defense',char_to_label('defense','Label2'));
EXECUTE sa_utl.set_row_label('defense',char_to_label('defense','Label2'));
EXECUTE sem_apis.create_entailment('inf', sem_models('contracts'),
sem_rulebases('owlprime'), SEM_APIS.REACH_CLOSURE, null, 'ENTAIL_ANYWAY=T');
```

SCOTT continues the preceding actions using the rest of the labels in the label sequence: Label1, Label2, ..., Label*n*. The last step will be as follows:

```
EXECUTE sa_utl.set_label('defense',char_to_label('defense','Labeln'));
EXECUTE sa_utl.set_row_label('defense',char_to_label('defense','Labeln'));
EXECUTE sem_apis.create_entailment('inf', sem_models('contracts'),
sem_rulebases('owlprime'), SEM_APIS.REACH_CLOSURE, null, 'ENTAIL_ANYWAY=T');
```

After all these actions are performed, the inference graph probably consists of triples tagged with various different labels.

Using Ladder-Based Inference (LBI)

Basically, ladder-based inference (LBI) wraps in one API call all the actions described in the Invoking SEM_APIS.CREATE_ENTAILMENT Multiple Times (page 5-3) approach. Visually, those actions are like climbing up a ladder. When proceeding from one label to the next, more asserted facts become visible or accessible (assuming the new label is not dominated by any of the previous ones), and therefore new relationships can be inferred.

The syntax to invoke LBI is shown in the following example.

```
EXECUTE sem_apis.create_entailment('inf',
  sem_models('contracts'),
  sem_rulebases('owlprime'),
  SEM_APIS.REACH_CLOSURE,
  null,
  null,
  ols_ladder_inf_lbl_seq=>'numericLabel1 numericLabel2 numericLabel3 numericLabel4'
);
```

The parameter `ols_ladder_inf_lbl_seq` specifies a sequence of labels. This sequence is provided as a list of numeric labels delimited by spaces. When using LBI, it is a good practice to arrange the sequence of labels so that weaker labels are put before stronger labels. This will reduce the size of the inferred graph. (If labels do not dominate each other, they can be specified in any order.)

5.1.2 Extended Example: Applying OLS Triple-Level Security on Semantic Data

This section presents an extended example illustrating how to apply OLS triple-level security to semantic data. It assumes that OLS has been configured and enabled. The examples are very simplified, and do not reflect recommended practices regarding user names and passwords.

Unless otherwise indicated, perform the steps while connected AS SYSDBA.

1. Perform some necessary setup steps.

   a. As SYSDBA, create database users named A, B, and C.

   ```
   create user a identified by <password-for-a>;
   grant connect, unlimited tablespace, resource to a;
   create user b identified by <password-for-b>;
   grant connect, unlimited tablespace, resource to b;
   create user c identified by <password-for-c>;
   grant connect, unlimited tablespace, resource to c;
   ```

   b. As SYSDBA, create a security administrator and grant privileges.

   ```
   CREATE USER fgac_admin identified by <password-for-fgac_admin>;
   GRANT connect, unlimited tablespace,resource to fgac_admin;
   GRANT SELECT ON mdsys.rdf_link$ to fgac_admin;
   GRANT EXECUTE ON sa_components TO fgac_admin;
   GRANT EXECUTE ON sa_user_admin TO fgac_admin;
   GRANT EXECUTE ON sa_label_admin TO fgac_admin;
   GRANT EXECUTE ON sa_policy_admin TO fgac_admin;
   GRANT EXECUTE ON sa_sysdba to fgac_admin;
   GRANT EXECUTE ON TO_LBAC_DATA_LABEL to fgac_admin;
   GRANT lbac_dba to fgac_admin;
   ```

   c. Connect as the security administrator and create a policy named defense.

   ```
   CONNECT fgac_admin/<password-for-fgac_admin>
   EXECUTE SA_SYSDBA.CREATE_POLICY('defense','ctxt1');
   ```

   d. Create three security levels (For simplicity, compartments and groups are omitted.)

   ```
   EXECUTE SA_COMPONENTS.CREATE_LEVEL('defense',3000,'TS','TOP SECRET');
   EXECUTE SA_COMPONENTS.CREATE_LEVEL('defense',2000,'SE','SECRET');
   EXECUTE SA_COMPONENTS.CREATE_LEVEL('defense',1000,'UN','UNCLASSIFIED');
   ```

   e. Create three labels.

   ```
   EXECUTE SA_LABEL_ADMIN.CREATE_LABEL('defense',1000,'UN');
   EXECUTE SA_LABEL_ADMIN.CREATE_LABEL('defense',1500,'SE');
   EXECUTE SA_LABEL_ADMIN.CREATE_LABEL('defense',3100,'TS');
   ```

   f. Assign labels and privileges.

   ```
   EXECUTE SA_USER_ADMIN.SET_USER_LABELS('defense','A','UN');
   EXECUTE SA_USER_ADMIN.SET_USER_LABELS('defense','B','SE');
   EXECUTE SA_USER_ADMIN.SET_USER_LABELS('defense','C','TS');
   EXECUTE SA_USER_ADMIN.SET_USER_LABELS('defense','fgac_admin','TS');
   EXECUTE SA_USER_ADMIN.SET_USER_PRIVS('defense','FGAC_ADMIN', 'full');
   ```

2. Create a semantic model.

   a. Create a model and share it with some other users.

```
CONNECT a/<password-for-a>
CREATE TABLE project_tpl (triple sdo_rdf_triple_s) compress for oltp;
EXECUTE sem_apis.create_sem_model('project', 'project_tpl', 'triple');
GRANT select on mdsys.rdfm_project to B;
GRANT select on mdsys.rdfm_project to C;
GRANT select, insert, update, delete on project_tpl to B, C;
```

b. Ensure that the bulk loading API can be executed.

```
GRANT insert on project_tpl to mdsys;
```

3. Apply the OLS policy for RDF.

```
CONNECT fgac_admin/fgac_admin
BEGIN
  sem_rdfsa.apply_ols_policy('defense', sem_rdfsa.TRIPLE_LEVEL_ONLY);
END;
/
```

Note that the application table now has an extra column named CTXT1:

```
CONNECT a/<password-for-a>a
DESCRIBE project_tpl;
Name                                      Null?    Type
----------------------------------------- -------- ------------------------
 TRIPLE                                            PUBLIC.SDO_RDF_TRIPLE_S
 CTXT1                                             NUMBER(10)
```

4. Add data to the semantic model.

```
-- User A uses incremental APIs to add semantic data
connnect a/<password-for a)
INSERT INTO project_tpl(triple) values
  (sdo_rdf_triple_s('project','<urn:A>','<urn:hasManager>','<urn:B>'));
INSERT INTO project_tpl(triple) values
  (sdo_rdf_triple_s('project','<urn:B>','<urn:hasManager>','<urn:C>'));
INSERT INTO project_tpl(triple) values
   (sdo_rdf_triple_s('project','<urn:A>','<urn:expenseReportAmount>','"100"'));
INSERT INTO project_tpl(triple) values

(sdo_rdf_triple_s('project','<urn:expenseReportAmount>','rdfs:subPropertyOf','<ur
n:projExp>'));
COMMIT;

-- User B uses bulk API to add semantic data
connect b/<password-for-b>
CREATE TABLE  project_stab(RDF$STC_GRAPH varchar2(4000),
RDF$STC_sub varchar2(4000),
RDF$STC_pred varchar2(4000),
RDF$STC_obj varchar2(4000)) compress;
GRANT select on project_stab to mdsys;

-- For simplicity, data types are omitted.
INSERT INTO project_stab values(null,
'<urn:B>','<urn:expenseReportAmount>','"200"');
INSERT INTO project_stab values(null,
'<urn:proj1>','<urn:deadline>','"2012-12-25"');
EXECUTE sem_apis.bulk_load_from_staging_table('project','b','project_stab');

-- As User B, check the contents in the application table
 connect b/<password-for-b>
SELECT * from a.project_tpl order by ctxt1;
```

```
SDO_RDF_TRIPLE_S(8.5963E+18, 7, 1.4711E+18, 2.0676E+18, 8.5963E+18)    1000
SDO_RDF_TRIPLE_S(5.1676E+18, 7, 8.5963E+18, 2.0676E+18, 5.1676E+18)    1000
SDO_RDF_TRIPLE_S(2.3688E+18, 7, 1.4711E+18, 4.6588E+18, 2.3688E+18)    1000
SDO_RDF_TRIPLE_S(7.6823E+18, 7, 4.6588E+18, 1.1911E+18, 7.6823E+18)    1000
SDO_RDF_TRIPLE_S(6.6322E+18, 7, 8.5963E+18, 4.6588E+18, 6.6322E+18)    1500
SDO_RDF_TRIPLE_S(8.4800E+18, 7, 6.2294E+18, 5.4118E+18, 8.4800E+18)    1500

6 rows selected.
SELECT count(1) from mdsys.rdfm_project;
6

-- As User A, check the contents in the application table
-- As expected, A can only see 4 triples
SQL> conn a/<password>
SQL> select * from a.project_tpl order by ctxt1;
SDO_RDF_TRIPLE_S(8.5963E+18, 7, 1.4711E+18, 2.0676E+18, 8.5963E+18)    1000

SDO_RDF_TRIPLE_S(5.1676E+18, 7, 8.5963E+18, 2.0676E+18, 5.1676E+18)    1000

SDO_RDF_TRIPLE_S(2.3688E+18, 7, 1.4711E+18, 4.6588E+18, 2.3688E+18)    1000

SDO_RDF_TRIPLE_S(7.6823E+18, 7, 4.6588E+18, 1.1911E+18, 7.6823E+18)    1000

SQL> select count(1) from mdsys.rdfm_project;
4

-- User C uses incremental APIs to add semantic data including 2 quads
connect c/<password-for-c>
INSERT INTO a.project_tpl(triple) values
  (sdo_rdf_triple_s('project','<urn:C>','<urn:expenseReportAmount>','"400"'));
INSERT INTO a.project_tpl(triple) values
  (sdo_rdf_triple_s('project','<urn:proj1>','<urn:hasBudget>','"10000"'));
INSERT INTO a.project_tpl(triple) values

(sdo_rdf_triple_s('project:<urn:proj2>','<urn:proj2>','<urn:hasBudget>','"20000"'
));
INSERT INTO a.project_tpl(triple) values

(sdo_rdf_triple_s('project:<urn:proj2>','<urn:proj2>','<urn:dependsOn>','<urn:pro
j1>'));
COMMIT;
```

5. Query the data as different users using the default label.

```
-- Now as user A, B, C, execute the following query
select lpad(nvl(g, ' '), 20) || ' ' || s || ' ' || p || ' ' || o from
table(sem_match('{ graph ?g { ?s ?p ?o }}',
sem_models('project'),
null,
null,
null,
null,
'GRAPH_MATCH_UNNAMED=T'
            ))
order by g, s, p, o;

connect  a/<password-for-a>
-- Repeat the preceding query
SQL> /

urn:A urn:expenseReportAmount 100
urn:A urn:hasManager urn:B
```

```
urn:B urn:hasManager urn:C
urn:expenseReportAmount http://www.w3.org/2000/01/rdf-schema#subPropertyOf
urn:projExp
SQL> connect  b/<password-for-b>
SQL> /

urn:A urn:expenseReportAmount 100
urn:A urn:hasManager urn:B
urn:B urn:expenseReportAmount 200
urn:B urn:hasManager urn:C
urn:expenseReportAmount http://www.w3.org/2000/01/rdf-schema#subPropertyOf
urn:projExp
urn:proj1 urn:deadline 2012-12-25
SQL> connect  c/<password-for-c>
SQL> /

urn:proj2 urn:proj2 urn:dependsOn urn:proj1
urn:proj2 urn:proj2 urn:hasBudget 20000
urn:A urn:expenseReportAmount 100
urn:A urn:hasManager urn:B
urn:B urn:expenseReportAmount 200
urn:B urn:hasManager urn:C
urn:C urn:expenseReportAmount 400
urn:expenseReportAmount http://www.w3.org/2000/01/rdf-schema#subPropertyOf
urn:projExp
urn:proj1 urn:deadline 2012-12-25
urn:proj1 urn:hasBudget 10000
```

As expected, different users (with different labels) can see different sets of triples in the project RDF graph.

6. Query the same data as a single user using different labels.

   The same query used in the preceding step produces just 6 matches:

```
urn:A urn:expenseReportAmount 100
urn:A urn:hasManager urn:B
urn:B urn:expenseReportAmount 200
urn:B urn:hasManager urn:C
urn:expenseReportAmount http://www.w3.org/2000/01/rdf-schema#subPropertyOf
urn:projExp
urn:proj1 urn:deadline 2012-12-25

6 rows selected.
```

   If user C picks the weakest label ("unclassified"), then user C sees even less

```
exec sa_utl.set_label('defense',char_to_label('defense','UN'));
exec sa_utl.set_row_label('defense',char_to_label('defense','UN'));
```

   The same query used in the preceding step produces just 4 matches:

```
urn:A urn:expenseReportAmount 100
urn:A urn:hasManager urn:B
urn:B urn:hasManager urn:C
urn:expenseReportAmount http://www.w3.org/2000/01/rdf-schema#subPropertyOf
urn:projExp
```

   If user C wants to run the query only against triples/quads with data label that dominates "Secret":

```
-- First set the label back
exec sa_utl.set_label('defense',char_to_label('defense','TS')); exec
sa_utl.set_row_label('defense',char_to_label('defense','TS'));
select lpad(nvl(g, ' '), 20) || ' ' || s || ' ' || p || ' ' || o
from table(sem_match('{ graph ?g { ?s ?p ?o }}',
sem_models('project'),
null,
null,
null,
null,
'MIN_LABEL=SE POLICY_NAME=DEFENSE GRAPH_MATCH_UNNAMED=T'
           ))
order by g, s, p, o;
```

The query response excludes those assertions made by user A:

```
urn:proj2 urn:proj2 urn:dependsOn urn:proj1
urn:proj2 urn:proj2 urn:hasBudget 20000
urn:B urn:expenseReportAmount 200
urn:C urn:expenseReportAmount 400
urn:proj1 urn:deadline 2012-12-25
urn:proj1 urn:hasBudget 10000

6 rows selected.
```

The same query can be executed as User A. However, no matches are returned, as expected.

You can delete semantic data when OLS is enabled for RDF. In the following example, assume that SEM_RDFSA.APPLY_OLS_POLICY (page 15-3) has been executed successfully, and that the same user setup and label designs are used as in the preceding example.

```
-- First, create a test model as user A and grant access to users B and C
connect a/<password-for-a>

create table test_tpl (triple sdo_rdf_triple_s) compress for oltp;
grant select on mdsys.rdfm_test to B,C;
grant select, insert, update, delete on test_tpl to B, C;

-- The following will fail with an error message
-- "Error while creating triggers: If OLS
-- is enabled,  you have to apply table policy
-- before creating an OLS-enabled model"
--
EXECUTE sem_apis.create_sem_model('test', 'test_tpl', 'triple');

-- You need to run this API first

connect fgac_admin/<password-for-fgac_admin>

EXECUTE sem_ols.apply_policy_to_app_tab('defense', 'A', 'TEST_TPL');

-- Now model creation (after OLS policy has been applied) can go through
connect a/<password-for-a>
EXECUTE sem_apis.create_sem_model('test', 'test_tpl', 'triple');

-- Add a triple as User A
INSERT INTO test_tpl(triple) values
  (sdo_rdf_triple_s('test','<urn:A>','<urn:p>','<urn:B>'));
COMMIT;
```

```
-- Add the same triple as User B
connect b/<password-for-b>
INSERT INTO a.test_tpl(triple) values
  (sdo_rdf_triple_s('test','<urn:A>','<urn:p>','<urn:B>'));
COMMIT;

-- Now User B can see both triples in the application table as well as the model view
set numwidth 20
SELECT * from a.test_tpl;

SDO_RDF_TRIPLE_S(8596269297967065604, 19, 1471072612573670395, 28121856352072361
78, 8596269297967065604)
                1000

SDO_RDF_TRIPLE_S(8596269297967065604, 19, 1471072612573670395, 28121856352072361
78, 8596269297967065604)
                1500

SELECT count(1) from mdsys.rdfm_test;
                   2

-- User A can only see one triple due to A's label assignment, as expected.

SELECT * from a.test_tpl;

SDO_RDF_TRIPLE_S(8596269297967065604, 19, 1471072612573670395, 28121856352072361
78, 8596269297967065604)
                1000

SELECT count(1) from mdsys.rdfm_test;
                   1


-- User A issues a delete to remove A's assertions
SQL> delete from a.test_tpl;
1 row deleted.

COMMIT;
Commit complete.


-- Now user A has no assertions left.

SELECT * from a.test_tpl;
no rows selected

SELECT count(1) from mdsys.rdfm_test;
                   0

-- Note that the preceding delete does not affect the same assertion made by B.
connect b/<password-for-b>
SELECT * from a.test_tpl;

SDO_RDF_TRIPLE_S(8596269297967065604, 19, 1471072612573670395, 28121856352072361
78, 8596269297967065604)
                1500

SELECT count(1) from mdsys.rdfm_test;
                   1

-- User B can remove this assertion using a DELETE statement.
```

```
-- The following DELETE statement uses the oracle_orardf_res2vid function
-- to narrow down the scope to triples with a particular subject.
DELETE FROM a.test_tpl app_tab
       where app_tab.triple.rdf_s_id =
             sdo_sem_inference.oracle_orardf_res2vid('<urn:A>');

1 row deleted.
```

5.2 Resource-Level Security

> **Note:**
>
> Oracle recommends that you generally use triple-level security rather than resource-level security. Triple-level security is described in Triple-Level Security (page 5-2).

The resource-level security option enables you to assign one or more security labels that define a security level for table rows. Conceptually, a table in a relational data model can be mapped to an equivalent RDF graph. Specifically, a row in a relational table can be mapped to a set of triples, each asserting some facts about a specific Subject. In this scenario, the subject represents the primary key for the row and each non-key column-value combination from the row is mapped to a predicate-object value combination for the corresponding triples.

A row in a relational data model is identified by its key, and OLS, as a row-level access control mechanism, effectively restricts access to the values associated with the key. With this conceptual mapping between relational and RDF data models, restricting access to a row in a relational table is equivalent to restricting access to a subgraph involving a specific subject. In a model that supports sensitivity labels for each triple, this is enforced by applying the same label to all the triples involving the given subject. However, you can also achieve greater flexibility by allowing the individual triples to have different labels, while maintaining a minimum bound for all the labels.

OLS support for RDF data employs a multilevel approach in which sensitivity labels associated with the triple components (subject, predicate, object) collectively form a minimum bound for the sensitivity label for the triple. With this approach, a data sensitivity label associated with an RDF resource (used as subject, predicate, or object) restricts unauthorized users from accessing any triples involving the resource and from creating new triples with the resource. For example, `projectHLS` as a subject may have a minimum sensitivity label, which ensures that all triples describing this subject have a sensitivity label that at least covers the label for `projectHLS`. Additionally, `hasContractValue` as a predicate may have a higher sensitivity label; and when this predicate is used with `projectHLS` to form a triple, that triple minimally has a label that covers both the subject and the predicate labels, as in the following example:

```
Triple 1: <http://www.myorg.com/contract/projectHLS> :ownedBy
                                <http://www.myorg.com/department/Dept1>
Triple 2: <http://www.myorg.com/contract/projectHLS> :hasContractValue
                                "100000"^^xsd:integer
```

Sensitivity labels are associated with the RDF resources (URIs) based on the position in which they appear in a triple. For example, the same RDF resource may appear in different positions (subject, predicate, or object) in different triples. Three unique labels can be assigned to each resource, so that the appropriate label is used to determine the label for a triple based on the position of the resource in the triple. You

can choose the specific resource positions to be secured in a database instance when you apply an OLS policy to the RDF repository. You can secure subjects, objects, predicates, or any combination, as explained in separate sections to follow. The following example applies an OLS policy named `defense` to the RDF repository and allows sensitivity labels to be associated with RDF subjects and predicates.

```
begin
  sem_rdfsa.apply_ols_policy(
        policy_name   => 'defense',
        rdfsa_options => sem_rdfsa.SECURE_SUBJECT+
                         sem_rdfsa.SECURE_PREDICATE);
end;
/
```

The same RDF resource can appear in both the subject and object positions (and sometime even as the predicate), and such a resource can have distinct sensitivity labels based on its position. A triple using the resource at a specific position should have a label that covers the label corresponding to the resource's position. In such cases, the triple can be asserted or accessed only by the users with labels that cover the triple and the resource labels.

In a specific RDF repository, security based on data classification techniques can be turned on for subjects, predicates, objects, or a combination of these. This ensures that all the triples added to the repository automatically conform to the label relationships described above.

5.2.1 Securing RDF Subjects

An RDF resource (typically a URI) appears in the subject position of a triple when an assertion is made *about* the resource. In this case, a sensitivity label associated with the resource has following characteristics:

- The label represents the minimum sensitivity label for any triple using the resource as a subject. In other words, the sensitivity label for the triple should dominate or cover the label for the subject.
- The label for a newly added triple is initialized to the user initial row label or is generated using the label function, if one is specified. Such operations are successful only if the triple's label dominates the label associated with the triple's subject.
- Only a user with an access label that dominates the subject's label and the triple's label can read the triple.

By default, the sensitivity label for a subject is derived from the user environment when an RDF resource is used in the subject position of a triple for the first time. The default sensitivity label in this case is set to the user's initial row label (the default that is assigned to all rows inserted by the user).

However, you can categorize an RDF resource as a subject and assign a sensitivity label to it even before it is used in a triple. The following example assigns a sensitivity label named `SECRET:HLS:US` to the `projectHLS` resource, thereby restricting the users who are able to define new triples about this resource and who are able to access existing triples with this resource as the subject:

```
begin
  sem_rdfsa.set_resource_label(
         model_name   => 'contracts',
         resource_uri => '<http://www.myorg.com/contract/projectHLS>',
```

```
        label_string => 'SECRET:HLS:US',
        resource_pos => 'S');
end;
```

5.2.2 Securing RDF Predicates

An RDF predicate defines the relationship between a subject and an object. You can use sensitivity labels associated with RDF predicates to restrict access to specific types of relationships with all subjects.

RDF predicates are analogous to columns in a relational table, and the ability to restrict access to specific predicates is equivalent to securing relational data at the column level. As in the case of securing the subject, the predicate's sensitivity label creates a minimum bound for any triples using this predicate. It is also the minimum authorization that a user must posses to define a triple with the predicate or to access a triple with the predicate.

The following example assigns the label `HSECRET:FIN` (in this scenario, a label that is Highly Secret and that also belongs to the Finance department) to triples with the `hasContractValue` predicate, to ensure that only a user with such clearance can define the triple or access it:

```
begin
  sem_rdfsa.set_predicate_label(
        model_name   => 'contracts',
        predicate    => '<http://www.myorg.com/pred/hasContractValue>',
        label_string => 'HSECRET:FIN');
end;
/
```

You can secure predicates in combination with subjects. In such cases, the triples using a subject and a predicate are ensured to have a sensitivity label that at least covers the labels for both the subject and the predicate. Extending the preceding example, if `projectHLS` as a subject is secured with label `SECRET:HLS:US` and if `hasContractValue` as a predicate is secured with label `HSECRET:FIN:`, a triple assigning a monetary value for `projectHLS` should at least have a label `HSECRET:HLS,FIN:US`. Effectively, a user's label must dominate this triple's label to be able to define or access the triple.

5.2.3 Securing RDF Objects

An RDF triple can have an URI or a literal in its object position. The URI in object position of a triple represents some resource. You can secure a resource in the object position by associating a sensitivity label to it, to restrict the ability to use the resource as an object in triples.

Typically, a resource (URI or non-literal) appearing in the object position of a triple may itself be described using additional RDF statements. Effectively, an RDF resource in the object position could appear in the subject position in some other triples. When the RDF resources are secured at the object position without explicit sensitivity labels, the label associated with the same resource in the subject position is used as the default label for the object.

5.2.4 Generating Labels for Inferred Triples

RDF data model allows for specification of declarative rules, enabling it to *infer* the presence of RDF statements that are not explicitly added to the repository. The following shows some simple declarative rules associated with the logic that projects

can be owned by departments and departments have Vice Presidents, and in such cases the project leader is by default the Vice President of the department that owns the project.

```
RuleID -> projectLedBy
Antecedent Expression -> (?proj :ownedBy ?dept) (?dept :hasVP ?person)
Consequent Expression -> (?proj :isLedBy ?person)
```

An RDF rule uses some explicitly asserted triples as well as previously inferred triples as antecedents, and infers one or more consequent triples. Traditionally, the inference process is executed as an offline operation to pregenerate all the inferred triples and to make them available for subsequent query operations.

When the underlying RDF graph is secured using OLS, any additional data inferred from the graph should also be secured to avoid exposing the data to unauthorized users. Additionally, the inference process should run with higher privileges, specifically with full access to data, in order to ensure completeness.

OLS support for RDF data offers techniques to generate sensitivity labels for inferred triples based on labels associated with one or more RDF artifacts. It provides label generation techniques that you can invoke at the time of inference. Additionally, it provides an extensibility framework, which allows an extensible implementation to receive a set of possible labels for a specific triple and determine the most appropriate sensitivity label for the triple based on some application-specific logic. The techniques that you can use for generating the labels for inferred triples include the following (each technique, except for Use Antecedent Labels, is associated with a SEM_RDFSA package constant):

- Use Rule Label (`SEM_RDFSA.LABELGEN_RULE`): An inferred triple is directly generated by a specific rule, and it may be indirectly dependent on other rules through its antecedents. Each rule may have a sensitivity label, which is used as the sensitivity label for all the triples directly inferred by the rule.

- Use Subject Label (`SEM_RDFSA.LABELGEN_SUBJECT`): Derives the label for the inferred triple by considering any sensitivity labels associated with the subject in the new triple. Each inferred triple has a subject, which could in turn be a subject, predicate, or object in any of the triple's antecedents. When such RDF resources are secured, the subject in the newly inferred triple may have one or more labels associated with it. With the Use Subject Label technique, the label for the inferred triple is set to the unique label associated with the RDF resource. When more than one label exists for the resource, you can implement the extensible logic to determine the most relevant label for the new triple.

- Use Predicate Label (`SEM_RDFSA.LABELGEN_PREDICATE`): Derives the label for the inferred triple by considering any sensitivity labels associated with the predicate in the new triple. Each inferred triple has a predicate, which could in turn be a subject, predicate, or object in any of the triple's antecedents. When such RDF resources are secured, the predicate in the newly inferred triple may have one or more labels associated with it. With the Use Predicate Label technique, the label for the inferred triple is set to the unique label associated with the RDF resource. When more than one label exists for the resource, you can implement the extensible logic to determine the most relevant label for the new triple.

- Use Object Label (`SEM_RDFSA.LABELGEN_OBJECT`): Derives the label for the inferred triple by considering any sensitivity labels associated with the object in the new triple. Each inferred triple has an object, which could in turn be a subject, predicate, or object in any of the triple's antecedents. When such RDF resources are secured, the object in the newly inferred triple may have one or more labels

associated with it. With the Use Object Label technique, the label for the inferred triple is set to the unique label associated with the RDF resource. When more than one label exists for the resource, you can implement the extensible logic to determine the most relevant label for the new triple.

- Use Dominating Label (`SEM_RDFSA.LABELGEN_DOMINATING`): Each inferred triple minimally has four direct components: subject, predicate, object, and the rule that produced the triple. With the Use Dominating Label technique, at the time of inference the label generator computes the most dominating of the sensitivity labels associated with each of the component and assigns it as the sensitivity label for the inferred triple. Exception labels are assigned when a clear dominating relationship cannot be established between various labels.

- Use Antecedent Labels: In addition to the four direct components for each inferred triple (subject, predicate, object, and the rule that produced the triple), a triple may have one or more antecedent triples, which are instrumental in deducing the new triple. With the Use Antecedent Labels technique, the labels for all the antecedent triples are considered, and conflict resolution criteria are implemented to determine the most appropriate label for the new triple. Since an inferred triple may be dependent on other inferred triples, a strict order is followed while generating the labels for all the inferred triples.

  The Use Antecedent Labels technique requires that you use a custom label generator. For information about creating and using a custom label generator, see Using Labels Based on Application Logic (page 5-16).

The following example creates an entailment (rules index) for the contracts data using a specific rulebase. This operation can only be performed by a user with FULL access privilege with the OLS policy applied to the RDF repository. In this case, the labels generated for the inferred triples are based on the labels associated with their predicates, as indicated by the use of the `SEM_RDFSA.LABELGEN_PREDICATE` package constant in the `label_gen` parameter.

```
begin
  sem_rdfsa.create_entailment(
         index_name_in   => 'contracts_inf',
         models_in       => SDO_RDF_Models('contracts'),
         rulebases_in    => SDO_RDF_Rulebases('contracts_rb'),
         options         => 'USER_RULES=T',
         label_gen       => sem_rdfsa.LABELGEN_PREDICATE);
end;
```

When the predefined or extensible label generation implementation cannot compute a unique label to be applied to an inferred triple, an exception label is set for the triple. Such triples are not accessible by any user other than the user with full access to RDF data (also the user initiating the inference process). The triples with exception labels are clearly marked, so that a privileged user can access them and apply meaningful labels manually. After the sensitivity labels are applied to inferred triples, only users with compatible labels can access these triples. The following example updates the sensitivity label for triples for which an exception label was set:

```
update mdsys.rdfi_contracts_inf
     set ctxt1 = char_to_label('defense', 'SECRET:HLS:US')
     where ctxt1 = -1;
```

Inferred triples accessed through generated labels might not be same as conceptual triples inferred directly from the user accessible triples and rules. The labels generated using system-defined or custom implementations cannot be guaranteed to be precise. See the information about Fine-Grained Access Control (OLS and VPD)

Considerations (page 11-23) in the Usage Notes for the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure in SEM_APIS Package Subprograms (page 11-1) for details.

5.2.5 Using Labels Based on Application Logic

The MDSYS.RDFSA_LABELGEN type is used to apply appropriate label generator logic at the time of index creation; however, you can also extend this type to implement a custom label generator and generate labels based on application logic. The label generator is specified using the `label_gen` parameter with the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure. To use a system-defined label generator, specify a SEM_RDFSA package constant, as explained in Generating Labels for Inferred Triples (page 5-13); to use a custom label generator, you must implement a custom label generator type and specify an instance of that type instead of a SEM_RDFSA package constant.

To create a custom label generator type, you must have the UNDER privilege on the RDFSA_LABELGEN type. In addition, to create an index for RDF data , you must should have the EXECUTE privilege on this type. The following example grants these privileges to a user named RDF_ADMIN:

```
GRANT under, execute ON mdsys.rdfsa_labelgen TO rdf_admin;
```

The custom label generator type must implement a constructor, which should set the dependent resources and specify the getNumericLabel method to return the label computed from the information passed in, as shown in the following example:

```
CREATE OR REPLACE TYPE CustomSPORALabel UNDER mdsys.rdfsa_labelgen  (
   constructor function CustomSPORALabel return self as result,
   overriding member function getNumericLabel (
                                    subject   rdfsa_resource,
                                    predicate rdfsa_resource,
                                    object    rdfsa_resource,
                                    rule      rdfsa_resource,
                                    anteced   rdfsa_resource)
        return number);
```

Example 5-1 Creating a Custom Label Generator Type

```
CREATE OR REPLACE TYPE BODY CustomSPORALabel AS

   constructor function CustomSPORALabel return self as result as
   begin
     self.setDepResources(sem_rdfsa.USE_SUBJECT_LABEL+
                          sem_rdfsa.USE_PREDICATE_LABEL+
                          sem_rdfsa.USE_OBJECT_LABEL+
                          sem_rdfsa.USE_RULE_LABEL+
                          sem_rdfsa.USE_ANTECED_LABELS);
     return;
   end CustomSPORALabel;

   overriding member function getNumericLabel (
                                    subject   rdfsa_resource,
                                    predicate rdfsa_resource,
                                    object    rdfsa_resource,
                                    rule      rdfsa_resource,
                                    anteced   rdfsa_resource)
        return number as
     labellst mdsys.int_array := mdsys.int_array();
   begin
```

```
    -- Find dominating label of S P O R A -
    -- Application specific logic for computing the triple label -
    -- Copy over all labels to labellst --
    for li in 1 .. subject.getLabelCount() loop
      labellst.extend;
      labellst(labellst.COUNT) = subject.getLabel(li);
    end loop;
    --- Copy over other labels as well ---
    --- Find a dominating of all the labels. Generates -1 if no
    --- dominating label within the set
    return self.findDominatingOf(labellst);
   end getNumericLabel;
  end CustomSPORALabel;
  /
```

The label generator constructor uses a set of constants defined in the SEM_RDFSA package to indicate the list of resources on which the label generator relies. The dependent resources are identified as an inferred triple's subject, its predicate, its object, the rule that produced the triple, and its antecedents. A custom label generator can rely on any subset of these resources for generating the labels, and you can specify this in its constructor by using the constants defined in SEM_RDFSA package : USE_SUBJECT_LABEL, USE_PREDICATE_LABEL, USE_OBJECT_LABEL, USE_RULE_LABEL, USE_ANTCED_LABEL. The following example creates the type body and specifies the constructor:

Example 5-1 (page 5-16) creates the type body, specifying the constructor function and the getNumericLabel member function. (Application-specific logic is not included in this example.)

In Example 5-1 (page 5-16), the sample label generator implementation uses all the resources contributing to the inferred triple for generating a sensitivity label for the triple. Thus, the constructor uses the `setDepResources` method defined in the superclass to set all its dependent components. The list of dependent resources set with this step determines the exact list of values passed to the label generating routine.

The `getNumericLabel` method is the label generation routine that has one argument for each resource that an inferred triple may depend on. Some arguments may be null values if the corresponding dependent resource is not set in the constructor implementation.

The label generator implementation can make use of a general-purpose static routine defined in the RDFSA_LABELGEN type to find a domination label for a given set of labels. A set of labels is passed in an instance of MDSYS.INT_ARRAY type, and the method finds a dominating label among them. If no such label exists, an exception label –1 is returned.

After you have implemented the custom label generator type, you can use the custom label generator for inferred data by assigning an instance of this type to the `label_gen` parameter in the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, as shown in the following example:

```
begin
  sem_apis.create_entailment(
         index_name_in  => 'contracts_rdfsinf',
         models_in      => SDO_RDF_Models('contracts'),
         rulebases_in   => SDO_RDF_Rulebases('RDFS'),
         options        => '',
         label_gen      => CustomSPORALabel());
end;
/
```

5.2.6 RDFOLS_SECURE_RESOURCE View

The MDSYS.RDFOLS_SECURE_RESOURCE view contains information about resources secured with Oracle Label Security (OLS) policies and the sensitivity labels associated with these resources.

Select privileges on this view can be granted to appropriate users. To view the resources associated with a specific model, you must also have select privileges on the model (or the corresponding RDFM_*model-name* view).

The MDSYS.RDFOLS_SECURE_RESOURCE view contains the columns shown in Table 5-1 (page 5-18).

Table 5-1 MDSYS.RDFOLS_SECURE_RESOURCE View Columns

Column Name	Data Type	Description
MODEL_NAME	VARCHAR2(25)	Name of the model.
MODEL_ID	NUMBER	Internal identifier for the model.
RESOURCE_ID	NUMBER	Internal identifier for the resource; to be joined with MDSYS.RDF_VALUE$.VALUE_ID column for information about the resource.
RESOURCE_TYPE	VARCHAR2(16)	One of the following string values to indicate the resource type for which the label is assigned: `SUBJECT`, `PREDICATE`, `OBJECT`, `GLOBAL`.
CTXT1	NUMBER	Sensitivity label assigned to the resource.

6

Workspace Manager Support for RDF Data

Note:

Effective with Oracle Database Release 12*c* (12.1), Workspace Manager support in RDF Semantic Graph is deprecated, and will be removed in an upcoming major release.

You should not develop new RDF Semantic Graph applications that depend on version-enabled models, and you should transition existing RDF Semantic Graph applications that depend on Workspace Manager to use an alternative.

The specific alternative to using Workspace Manager with semantic data depends on the purpose of the application. For information about use cases and recommended alternatives, see My Oracle Support Note 1468273.1.

RDF data stored in the Oracle Database semantic data store can be version-enabled using Oracle Database Workspace Manager. Using Workspace Manager, you can create new versions of RDF data through data modification operations while maintaining a copy of the old data. For detailed usage and reference information about Workspace Manager, see *Oracle Database Workspace Manager Developer's Guide*.

The unit of versioning for the semantic data store is a model, which is in turn associated with an application table that resides in a user schema. Traditional Workspace Manager interfaces are used to manage a version-enabled RDF model. (However, you cannot use version-enabling on a model that participates in a virtual model. Virtual models are described in Virtual Models (page 1-13))

Creating the first version-enabled RDF model in a database re-creates an index on the underlying triple store that holds data for all RDF models. Thus, the time needed to create the first version-enabled RDF model depends on the amount of existing data in these models.

This chapter discuss the key characteristics of a version-enabled RDF model and provides some examples. It contains the following major sections:

- Enabling Workspace Manager Support for RDF Data (page 6-2)
- Version-Enabling an RDF Model (page 6-2)
- Inferring from Version-Enabled RDF Models (page 6-3)
- Merging and Refreshing Workspaces in Version-Enabled RDF Models (page 6-4)
- Special Considerations When Using Workspace Manager Support for RDF Data (page 6-5)
- Usage Flow Example: Versioning Semantic Models (page 6-5)

6.1 Enabling Workspace Manager Support for RDF Data

Workspace Manager support for RDF data is not installed in the database by default. Instead, you must run a script named `sdordfwm.sql` *after* you enable RDF Semantic Graph support in the database (explained in Enabling RDF Semantic Graph Support (page A-1)). You only need to run `sdordfwm.sql` once for the database. To run this script, connect to the database as SYSDBA and enter the following statement as appropriate for your operating system:

- Linux: `SQL> @$ORACLE_HOME/md/admin/sdordfwm.sql`
- Windows: `SQL> @%ORACLE_HOME%\md\admin\sdordfwm.sql`

If you want to check whether Workspace Manager support is enabled for RDF data, enter the following query and see if the value returned is `INSTALLED`:

```
SELECT value FROM mdsys.rdf_parameter
  WHERE namespace = 'COMPONENT' and attribute = 'RDFOWM';
```

6.1.1 Removing Workspace Manager Support for RDF Data

After you have enabled Workspace Manager support for RDF data, you might need to remove that support in certain circumstances. For example, before you downgrade or remove RDF Semantic Graph support in the database, you must remove the Workspace Manager support for RDF data. (For information about downgrading and removing RDF Semantic Graph support, see Enabling_ Downgrading_ or Removing RDF Semantic Graph Support (page A-1).)

To remove Workspace Manager support for RDF data, perform the following steps:

1. Connect to the database as the SYS user with SYSDBA privileges (SYS AS SYSDBA, and enter the SYS account password when prompted).
2. Start SQL*Plus, and enter the following statement:
   - Linux: `@$ORACLE_HOME/md/admin/sdordfwm_rm.sql`
   - Windows: `@%ORACLE_HOME%\md\admin\sdordfwm_rm.sql`

After the `sdordfwm_rm.sql` script completes successfully, you can downgrade or remove RDF Semantic Graph support in the database.

6.2 Version-Enabling an RDF Model

To version-enable an RDF model, you must create any necessary entailments (rules indexes) on it (using the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure), and then version-enable its associated application table. For example, if an RDF model named `contracts` is associated with an application table named CONTRACTS_RDF_DATA, you can perform the version enabling as shown in the following example:

```
begin
  dbms_wm.enableVersioning (table_name => 'contracts_rdf_data');
end;
/
```

The application table must have a primary key, and you cannot specify the HISTORY or VALID TIME options.

The data stored in the application table and the corresponding RDF data in the semantic data store are versioned logically. The standard data manipulation operations (insert, update, delete) on a version-enabled application table maintain the version information for both the affected application table rows and the corresponding RDF triples in the semantic data store. Queries accessing the RDF data using the SEM_MATCH operator take the state of the active workspace and the relevant versions of data into account when constructing the result set.

You cannot use continually refreshed workspaces and multiparent workspaces when working with RDF data. However, other workspace operations (such as creating new workspaces, creating savepoints, setting access modes, and refreshing and merging workspaces) proceed as you would expect.

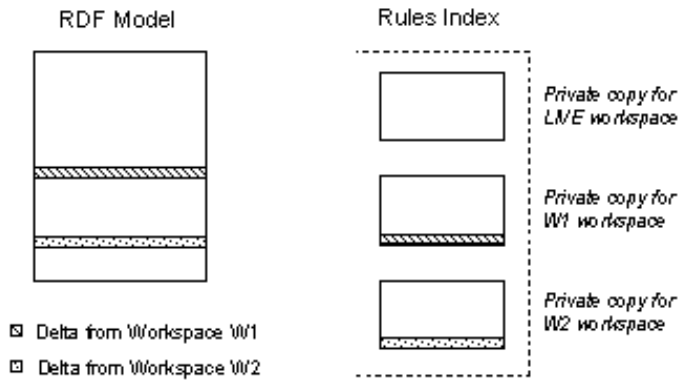
You can use Workspace Manager locks on the application table rows to avoid conflicting data manipulation operations in concurrent workspaces. The conflicts in such cases are detected based on the primary keys for the locked rows, and they may not directly translate to any meaningful locks on the RDF data, such as a lock on a subgraph or a node. (See Merging and Refreshing Workspaces in Version-Enabled RDF Models (page 6-4) for best practices involving version-enabled RDF data.)

6.3 Inferring from Version-Enabled RDF Models

You can use an RDF model to create one or more entailments through inference using system-defined and user-defined rulebases. When such a model is version-enabled, the associated entailments that store the inferred data are also version-enabled.

However, unlike the RDF models, which are versioned logically, the entailments are versioned physically. Thus, a workspace with some unmerged changes to an RDF model may maintain a private version of the entailment, which duplicates all the inferred triples that may already exist in the entailment corresponding to the parent workspace. Figure 6-1 (page 6-3) shows a physically versioned entailment where private copies of the index are maintained for workspaces W1 and W2, which are child workspaces of the LIVE workspace.

Figure 6-1 Physical Versioning of Entailment (Rules Index)



A workspace created from a parent workspace with a valid entailment will share the entailment with its parent until some workspace-specific data manipulation operations occur on the corresponding models. The first data manipulation operation within a workspace creates a private copy of the entailment and marks it INVALID. Workspace-private versions of inference data can be rebuilt using the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, as shown in the following example:

```
begin
  sem_apis.create_entailment(
            index_name_in   => 'contracts_rdfs',
            models_in       => SDO_RDF_Models('contracts'),
            rulebases_in    => SDO_RDF_Rulebases('RDFS'));
end;
/
```

Any subsequent additions to the RDF model within the workspace will mark the entailment as INCOMPLETE, although you can return it to a VALID status using the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure. When a workspace is merged, the workspace-private entailment is dropped, and the corresponding index in the parent workspace is marked INVALID.

6.4 Merging and Refreshing Workspaces in Version-Enabled RDF Models

A workspace modifying data stored in a version-enabled RDF model may contain additional changes to other RDF models or relational tables. The semantics for the merging or refreshing relational data modified in a workspace remain unchanged. Similarly, the merging or refreshing of the application table is subject to conflict analysis based on the primary key defined in the application table. Triples added and deleted in concurrent workspaces are not flagged as conflicts unless they are associated with the same primary key in the application table. For example, a parent and a child workspace may assert the same triple with different primary keys, and during a workspace merge both the triples are accepted. You can avoid such conflicts by generating the application table primary keys from the triple components.

When you work with RDF data, the types of conflicts that must be flagged vary depending on the application requirements. The conflicts include physical conflicts that exist between two triples, and logical conflicts that exist in a graph to which the triples belong. For example, two concurrent workspaces asserting a triple that captures the value of a specific contract physically conflict with each other. Similarly, a workspace that assigned a new project to a department (with an appropriate triple) may logically conflict with other workspace that reduced the budget for the department.

You can avoid these conflicts by acquiring appropriate Workspace Manager locks on the application table rows using the DBMS_WM.lockRows subprogram. For example, while adding a new triple that describes a specific resource, the application logic may acquire row locks for all application table rows that describe the same resource; in this case, failure to acquire the lock indicates a conflict.

If business conflicts between concurrent workspaces can be manually detected and reconciled, you can use the <*application table name*>_DIFF Workspace Manager view to obtain a summary of all the changes in two workspaces. For a version-enabled application table named `contracts_rdf_data`, the query in following example fetches the triples for all the rows that are modified either in `W2` workspace or the `LIVE` workspace:

```
begin
  dbms_wm.SetDiffVersions(workspace1 => 'W2',
                          workspace2 => 'LIVE');
end;
/

SELECT appt.apptable_pkey, appt.triple.get_triple() triple,
       wm_diffver, wm_code FROM contracts_rdf_data_diff appt;
```

For an explanation of the possible values for the WM_DIFFVER and WM_CODE columns, see the section about *xxx*_DIFF views in *Oracle Database Workspace Manager Developer's Guide*.

For any conflicts detected using this technique, you can resolve them by executing compensating data manipulation operations for the conflicting data modifications.

6.5 Special Considerations When Using Workspace Manager Support for RDF Data

Certain RDF Semantic Graph features are not compatible with Workspace Manager for RDF data.

The following network-level features are not supported on semantic networks containing version-enabled RDF models:

- Oracle Label Security (OLS) for RDF data
- Virtual Private Database (VPD) for RDF data
- Semantic indexing for documents
- SEM_APIS.DROP_SEM_NETWORK (page 11-39) with the `cascade` option

The following model-level features are not supported on version-enabled RDF models:

- Participation in virtual models
- Incremental inference
- SEM_APIS.ALTER_MODEL (page 11-4)
- SEM_APIS.MERGE_MODELS (page 11-62)
- SEM_APIS.REMOVE_DUPLICATES (page 11-65)
- SEM_APIS.RENAME_ENTAILMENT (page 11-66)
- SEM_APIS.RENAME_MODEL (page 11-67)
- SEM_APIS.SWAP_NAMES (page 11-69)

Version-enabled RDF models cannot be used in a CDB (multitenant container database) environment.

6.6 Usage Flow Example: Versioning Semantic Models

Example 6-1 (page 6-6) shows how entailments can be handled in different workspaces. Comments within the example explain the individual operations.

Note the following restrictions and considerations for version-enabled RDF models, which are reflected in Example 6-1 (page 6-6):

- Incremental inference is not supported for entailments involving version-enabled RDF models.
- Although an already created entailment can be made VALID by calling SEM_APIS.CREATE_ENTAILMENT (page 11-15), a new entailment cannot be created once the application table corresponding to an RDF model has been version-enabled.

- The pattern argument for SEM_MATCH needs to use the old (non-curly-brace) syntax.
- The following option must be used in the SEM_APIS.CREATE_ENTAILMENT (page 11-15) invocation involving version-enabled models: `'EPT=1'` (or `options => 'USER_RULES=T,EPT=1'` when user-defined rules are used) .

Example 6-1 Versioning a Semantic Model

```
set pagesize 10000

-- 1. Enabling Workspace Manager Support for RDF Data
conn / as sysdba
@?/md/admin/sdordfwm.sql

set echo on
column x format a10
column y format a10

column owner format a10
column index_name format a12
column index_view_name format a18
column status format a10

-- create sem network and a test user
exec sem_apis.create_sem_network('SYSAUX');
grant connect,resource,unlimited tablespace to rdfuser identified by rdfuser;

-- 2. Create table and model: then Insert triples (that will lead to some inferred
triples, when entailed)

conn rdfuser/<password>

CREATE TABLE emp_rdf_data(id number,triple sdo_rdf_triple_s);
ALTER TABLE emp_rdf_data add CONSTRAINT emp_rdf_data_PK PRIMARY KEY (ID);

EXEC SEM_APIS.CREATE_SEM_MODEL('emp0','emp_rdf_data','triple');

insert into emp_rdf_data values (0.1,
sdo_rdf_triple_s('emp0','<Man>','rdfs:subClassOf','<Person>'));
insert into emp_rdf_data values (0.2,
sdo_rdf_triple_s('emp0','<Woman>','rdfs:subClassOf','<Person>'));
insert into emp_rdf_data values (1,
sdo_rdf_triple_s('emp0','<John>','rdf:type','<Man>'));

-- 3. Create entailments: one with user-def rules only and the other with OWL

exec sdo_rdf_inference.create_rulebase('family_rb');

insert into mdsys.rdfr_family_rb values (
'R1',
'(?x rdf:type <Man>)',
NULL,
'(?x rdf:type <Male>)',
NULL);

insert into mdsys.rdfr_family_rb values (
'R2',
'(?x rdf:type <Woman>)',
NULL,
'(?x rdf:type <Female>)',
```

```
NULL);

begin
sem_apis.create_entailment(
index_name_in => 'emp0_u_idx',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('family_rb'),
options => 'USER_RULES=T,EPT=1');
end;
/

begin
sem_apis.create_entailment(
index_name_in => 'emp0_idx',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('owlprime'),
options => 'EPT=1');
end;
/

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 4. Activate versioning and re-try the queries

EXEC dbms_wm.enableVersioning (table_name => 'emp_rdf_data');

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;


-- 4a. [fail] try to create a post-versioning entailment for testing

begin
sem_apis.create_entailment(
index_name_in => 'emp0_idx_2',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('owl2rl'),
options => 'EPT=1');
end;
/


-- 5. Create Workspace and move into that workspace

exec dbms_wm.createworkspace('W1');
exec dbms_wm.gotoWorkspace('W1');

-- 5a. [fail] in this workspace, try to create a post-versioning entailment for
testing
```

```
begin
sem_apis.create_entailment(
index_name_in => 'emp0_idx_3',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('rdfs'),
options => 'EPT=1');
end;
/

-- 6. Workspace inherits the entailment from LIVE workspace (no physical copy yet)

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 7. Insert some triples => causes LIVE entailments to be copied to this workspace
and marked INVALID here
insert into emp_rdf_data values (2.1,
sdo_rdf_triple_s('emp0','<Mary>','rdf:type','<Woman>'));
commit;

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 8. Go back to LIVE workspace to confirm that the LIVE entailments are still VALID

disconnect
connect rdfuser/rdfuser
select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 9. Back to new workspace: make its entailments VALID

exec dbms_wm.gotoWorkspace('W1');

begin
sem_apis.create_entailment(
index_name_in => 'emp0_u_idx',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('family_rb'),
options => 'USER_RULES=T,EPT=1');
end;
/

begin
sem_apis.create_entailment(
index_name_in => 'emp0_idx',
```

```
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('owlprime'),
options => 'EPT=1');
end;
/

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 10. Go back to LIVE workspace to confirm that the entailments there are still
VALID

disconnect
connect rdfuser/rdfuser
select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 11. Insert a new row into LIVE model => causes LIVE entailments to become INVALID

insert into emp_rdf_data values (3,
sdo_rdf_triple_s('emp0','<Gary>','rdf:type','<Man>'));
commit;

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 12. Back to new workspace: confirm that its entailment is still VALID

exec dbms_wm.gotoWorkspace('W1');

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 13. Back to LIVE workspace: make the entailments VALID again

disconnect
connect rdfuser/rdfuser

begin
sem_apis.create_entailment(
index_name_in => 'emp0_u_idx',
```

```
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('family_rb'),
options => 'USER_RULES=T,EPT=1');
end;
/

begin
sem_apis.create_entailment(
index_name_in => 'emp0_idx',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('owlprime'),
options => 'EPT=1');
end;
/

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 14. Insert a new row into LIVE model => causes LIVE entailments to become
INCOMPLETE

insert into emp_rdf_data values (4,
sdo_rdf_triple_s('emp0','<Lory>','rdf:type','<Man>'));
commit;

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 15. make the entailments in LIVE workspace VALID again

begin
sem_apis.create_entailment(
index_name_in => 'emp0_u_idx',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('family_rb'),
options => 'USER_RULES=T,EPT=1');
end;
/

begin
sem_apis.create_entailment(
index_name_in => 'emp0_idx',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('owlprime'),
options => 'EPT=1');
end;
/

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
```

```
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 16. merge workspace: this should make the LIVE entailments INVALID
conn rdfuser/<password>
exec dbms_wm.mergeWorkspace('W1');

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- 17. remove workspace
exec dbms_wm.removeWorkspace('W1');

-- 18. disable versioning, refresh the INVALID entailments to make them VALID, and
then query
exec dbms_wm.disableVersioning('emp_rdf_data');

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

begin
sem_apis.create_entailment(
index_name_in => 'emp0_u_idx',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('family_rb'),
options => 'USER_RULES=T,EPT=1');
end;
/

begin
sem_apis.create_entailment(
index_name_in => 'emp0_idx',
models_in => SDO_RDF_Models('emp0'),
rulebases_in => SDO_RDF_Rulebases('owlprime'),
options => 'EPT=1');
end;
/

select * from mdsys.rdf_rules_index_info;
select count(*) from mdsys.rdfi_emp0_u_idx;
select x,y from TABLE(SEM_MATCH('(?x rdf:type ?
y)',sem_models('emp0'),sem_rulebases('family_rb'),null,null)) order by 1,2;
select count(*) from mdsys.rdfi_emp0_idx;
select x from TABLE(SEM_MATCH('(?x rdf:type
<Person>)',sem_models('emp0'),sem_rulebases('OWLPRIME'),null,null)) order by 1;

-- cleanup
exec sem_apis.drop_entailment('EMP0_IDX');
exec sem_apis.drop_entailment('EMP0_U_IDX');
```

```
exec sem_apis.drop_sem_model('EMP0');
exec sem_apis.drop_rulebase('FAMILY_RB');
drop table emp_rdf_data;
conn / as sysdba
drop user rdfuser cascade;
```

7

RDF Semantic Graph Support for Apache Jena

Note:

This feature was previously referred to as the *Jena Adapter for Oracle Database* and the *Jena Adapter*.

RDF Semantic Graph support for Apache Jena (also referred to here as support for Apache Jena) provides a Java-based interface to Oracle Spatial and Graph RDF Semantic Graph by implementing the well-known Jena Graph, Model, and DatasetGraph APIs. (Apache Jena is an open source framework. For license and copyright conditions, see `http://www.apache.org/licenses/` and `http://www.apache.org/licenses/LICENSE-2.0`.)

The DatasetGraph APIs are for managing named graph data, also referred to as **quads**. In addition, RDF Semantic Graph support for Apache Jena provides network analytical functions on top of semantic data through integrating with the Oracle Spatial and Graph Network Data Model Graph feature.

This chapter assumes that you are familiar with major concepts explained in RDF Semantic Graph Overview (page 1-1) and OWL Concepts (page 2-1). It also assumes that you are familiar with the overall capabilities and use of the Jena Java framework. For information about the Jena framework, see `http://jena.apache.org/`. If you use the network analytical function, you should also be familiar with the Network Data Model Graph feature, which is documented in *Oracle Spatial and Graph Topology Data Model and Network Data Model Graph Developer's Guide*.

Support for Apache Jena extends the semantic data management capabilities of Oracle Database RDF/OWL.

This chapter includes the following major topics:

- Setting Up the Software Environment (page 7-2)
- Setting Up the SPARQL Service (page 7-4)
- Setting Up the RDF Semantic Graph Environment (page 7-10)
- SEM_MATCH and RDF Semantic Graph Support for Apache Jena Queries Compared (page 7-11)
- Retrieving User-Friendly Java Objects from SEM_MATCH or SQL-Based Query Results (page 7-12)
- Optimized Handling of SPARQL Queries (page 7-15)
- Additions to the SPARQL Syntax to Support Other Features (page 7-17)

- Functions Supported in SPARQL Queries through RDF Semantic Graph Support for Apache Jena (page 7-24)
- SPARQL Update Support (page 7-29)
- Analytical Functions for RDF Data (page 7-31)
- Support for Server-Side APIs (page 7-38)
- Bulk Loading Using RDF Semantic Graph Support for Apache Jena (page 7-44)
- Automatic Variable Renaming (page 7-50)
- JavaScript Object Notation (JSON) Format Support (page 7-51)
- Other Recommendations and Guidelines (page 7-53)
- Example Queries Using RDF Semantic Graph Support for Apache Jena (page 7-56)
- SPARQL Gateway and Semantic Data (page 7-75)
- Deploying Joseki in Apache Tomcat or JBoss (page 7-94)

Note:

The current RDF Semantic Graph support for Apache Jena release has been tested against Apache Jena 2.11.1 and Joseki 3.4.4. Because of the nature of open source projects, you should not use this support for Apache Jena with later versions of Jena or Joseki.

7.1 Setting Up the Software Environment

To use the support for Apache Jena, you must first ensure that the system environment has the necessary software, including Oracle Database 11*g* Release 2 or later with the Spatial and Graph and Partitioning options and with RDF Semantic Graph support enabled, Apache Jena 2.11.1, and JDK 1.6 or later. You can set up the software environment by performing these actions:

1. Install Oracle Database Enterprise Edition with the Oracle Spatial and Graph and Partitioning Options.
2. If you have not yet installed Oracle Database Release 11.2.0.3 or later, install the 11.2.0.2 Patch Set for Oracle Database Server (`https://updates.oracle.com/Orion/PatchDetails/process_form?patch_num=10098816`).
3. Enable the support for RDF Semantic Graph, as explained in Enabling RDF Semantic Graph Support (page A-1).
4. Download RDF Semantic Graph support for Apache Jena from My Oracle Support at `http://support.oracle.com/`. Search the Knowledge Base for bug identifier 17241927.

   A full evaluation version of RDF Semantic Graph support for Apache Jena can be downloaded from OTN at `http://www.oracle.com/technetwork/database-options/spatialandgraph/`. Click the **Downloads** tab, and then under Licensed Software click **RDF Semantic Graph Licensed Software**.

5. Unzip the kit into a temporary directory, such as (on a Linux system) `/tmp/jena_adapter`. (If this temporary directory does not already exist, create it before the unzip operation.)

   The RDF Semantic Graph support for Apache Jena has the following top-level directories:

   ```
   |-- META-INF
     |-- examples
     |-- jar
     |-- javadoc
     |-- joseki
     |-- joseki_web_app
     |-- protege_plugin
     |-- sparqlgateway
     |-- sparqlgateway_web_app
     |-- fuseki
     |-- web
   ```

6. If JDK 1.6 or later is not already installed, install it.

7. If the JAVA_HOME environment variable does not already refer to the JDK 1.6 or later installation, define it accordingly. For example:

   ```
   setenv JAVA_HOME /usr/local/packages/jdk16/
   ```

8. If the SPARQL service to support the SPARQL protocol is not set up, set it up as explained in Setting Up the SPARQL Service (page 7-4).

After setting up the software environment, ensure that your RDF Semantic Graph environment can enable you to use the support for Apache Jena to perform queries, as explained in Setting Up the RDF Semantic Graph Environment (page 7-10).

7.1.1 If You Used a Previous Version of the Support for Apache Jena

If you used a previous version of the support for Apache Jena, note the following important changes for this version:

- `com.hp.hpl.jena.sparql.core.DataSourceImpl` is replaced by `com.hp.hpl.jena.sparql.core.DatasetImpl`.

  If you have `import com.hp.hpl.jena.sparql.core.DataSourceImpl` in your Java source code, you will need to update it to `import com.hp.hpl.jena.sparql.core.DatasetImpl`.

  If you have `DataSourceImpl.wrap` in your Java source code, you will need to update it to `DatasetImpl.wrap"`.

- `joseki-config.ttl` is moved under the `WEB-INF/classes` directory of `joseki.war`. This configuration file was formerly placed under the top-level directory of `joseki.war`. The following example shows the new placement:

  ```
  % /usr/local/packages/jdk16/bin/jar tf joseki.war
  application.xml
  index.html
  joseki-config-ttl_now_under_WEB-INF_classes
  META-INF/
  META-INF/MANIFEST.MF
  ojdbc6.jar
  StyleSheets/
  StyleSheets/joseki.css
  update.html
  ```

```
WEB-INF/
WEB-INF/lib/
WEB-INF/lib/sdordfclient.jar
WEB-INF/lib/jena-arq-2.11.1.jar
WEB-INF/lib/log4j-1.2.16.jar
WEB-INF/lib/jcl-over-slf4j-1.6.4.jar
WEB-INF/lib/httpclient-4.2.3.jar
WEB-INF/lib/commons-codec-1.6.jar
WEB-INF/lib/slf4j-log4j12-1.6.4.jar
WEB-INF/lib/servlet-api-2.5-20081211.jar
WEB-INF/lib/xercesImpl-2.11.0.jar
WEB-INF/lib/slf4j-api-1.6.4.jar
WEB-INF/lib/jena-tdb-1.0.1.jar
WEB-INF/lib/jena-core-2.11.1.jar
WEB-INF/lib/httpcore-4.2.2.jar
WEB-INF/lib/joseki-3.4.4.oracle_build_jena211.jar
WEB-INF/lib/jena-iri-1.0.1.jar
WEB-INF/lib/sdordf.jar
WEB-INF/lib/xml-apis-1.4.01.jar
WEB-INF/web.xml
WEB-INF/classes/
WEB-INF/classes/joseki-config.ttl
xml-to-html.xsl
```

7.2 Setting Up the SPARQL Service

This section explains how to set up a SPARQL web service endpoint by deploying the `joseki.war` file in WebLogic Server.

Note:

If you want to deploy Joseki in Apache Tomcat or JBoss instead of WebLogic Server, see Deploying Joseki in Apache Tomcat or JBoss (page 7-94).

---

1. Download and Install Oracle WebLogic Server 11g or later.
2. Ensure that you have Java 6 or later installed, because it is required by Joseki 3.4.4.
3. Using the WebLogic Server Administration console, create a J2EE data source named *OracleSemDS*. During the data source creation, you can specify a user and password for the database schema that contains the relevant semantic data against which SPARQL queries are to be executed.

   If you need help in creating this data source, see Creating the Required Data Source Using WebLogic Server (page 7-6).
4. Go to the `autodeploy` directory of WebLogic Server and copy files, as follows. (For information about auto-deploying applications in development domains, see: `http://docs.oracle.com/cd/E24329_01/web.1211/e24443/autodeploy.htm`)

   ```
   cd <domain_name>/autodeploy
   cp -rf  /tmp/jena_adapter/joseki_web_app/joseki.war  <domain_name>/autodeploy
   ```

   In the preceding example, <*domain_name*> is the name of a WebLogic Server domain.

Note that while you can run a WebLogic Server domain in two different modes, development and production, only development mode allows you use the auto-deployment feature.

5. Verify your deployment by using your Web browser to connect to a URL in the following format (assume that the Web application is deployed at port 7001): `http://<hostname>:7001/joseki`

   You should see a page titled *Oracle SPARQL Service Endpoint using Joseki*, and the first text box should contain an example SPARQL query.

6. Click **Submit Query**.

   You should see a page titled *Oracle SPARQL Endpoint Query Results*. There may or may not be any results, depending on the underlying semantic model against which the query is executed.

By default, the `joseki-config.ttl` file contains an `oracle:Dataset` definition using a model named `M_NAMED_GRAPHS`. The following snippet shows the configuration. The `oracle:allGraphs` predicate denotes that the SPARQL service endpoint will serve queries using all graphs stored in the `M_NAMED_GRAPHS` model.

```
<#oracle> rdf:type oracle:Dataset;
    joseki:poolSize     1 ;         ## Number of concurrent connections allowed to
this dataset.
    oracle:connection
    [ a oracle:OracleConnection ;
    ];
    oracle:allGraphs [ oracle:firstModel "M_NAMED_GRAPHS" ] .
```

The `M_NAMED_GRAPHS` model will be created automatically (if it does not already exist) upon the first SPARQL query request. You can add a few example triples and quads to test the named graph functions; for example:

```
SQL> CONNECT username/password
SQL> INSERT INTO m_named_graphs_tpl
VALUES(sdo_rdf_triple_s('m_named_graphs','<urn:s>','<urn:p>','<urn:o>'));
SQL> INSERT INTO m_named_graphs_tpl
VALUES(sdo_rdf_triple_s('m_named_graphs:<urn:G1>','<urn:g1_s>','<urn:g1_p>','<urn:g1_
o>'));
SQL> INSERT INTO m_named_graphs_tpl
VALUES(sdo_rdf_triple_s('m_named_graphs:<urn:G2>','<urn:g2_s>','<urn:g2_p>','<urn:g2_
o>'));
SQL> COMMIT;
```

After inserting the rows, go to `http://<hostname>:7001/joseki`, type the following SPARQL query, and click **Submit Query**:

```
SELECT ?g ?s ?p ?o
WHERE
  { GRAPH ?g { ?s ?p ?o} }
```

The result should be an HTML table with four columns and two sets of result bindings.

The `http://<hostname>:7001/joseki` page also contains a **JSON Output** option. If this option is selected (enabled), the SPARQL query response is converted to JSON format.

7.2.1 Creating the Required Data Source Using WebLogic Server

If you need help creating the required J2EE data source using the WebLogic Server admin console, you can follow these steps:

1. Login to: `http://<hostname>:7001/console`
2. In the Domain Structure panel, click **Services**.
3. Click **JDBC**
4. Click **Data Sources**.
5. In the Summary of JDBC Data Sources panel, click **New** under the Data Sources table.
6. In the Create a New JDBC Data Source panel, enter or select the following values.

   Name: `OracleSemDS`

   JNDI Name: `OracleSemDS`

   Database Type: `Oracle`

   Database Driver: `Oracle's Driver (Thin) Versions: 9.0.1,9.2.0,10,11`
7. Click **Next** twice.
8. In the Connection Properties panel, enter the appropriate values for the **Database Name**, **Host Name**, **Port**, **Database User Name** (schema that contains semantic data), **Password** fields.
9. Click **Next**.
10. Select (check) the target server or servers to which you want to deploy this `OracleSemDS` data source.
11. Click **Finish**.

    You should see a message that all changes have been activated and no restart is necessary.

7.2.2 Configuring the Joseki-Based SPARQL Service

By default, the SPARQL Service endpoint assumes that the queries are to be executed against a semantic model with a pre-set name. This semantic model is owned by the schema specified in the J2EE data source with a default JNDI name *OracleSemDS*. Note that you do not need to create this model explicitly using PL/SQL or Java; if the model does not exist in the network, it will be automatically created, along with the necessary application table and index.

Note:

Effective with the support for Apache Jena release in November 2011, the application table index (<*model_name*>_idx) definition is changed to accommodate named graph data (quads).

For existing models created by an older version of the support for Apache Jena, you can migrate the application table index name and definition by using the `static OracleUtils.migrateApplicationTableIndex(oracle, graph, dop)` method in the `oracle.spatial.rdf.client.jena` package. (See the Javadoc for more information.) Note that the new index definition is *critical* to the performance of DML operations against the application table.

To change the default JNDI name or to use a default semantic model, you can configure the SPARQL service by editing the `joseki-config.ttl` configuration file, which is embedded under the `WEB-INF/classes` directory in the prebuilt application `joseki_web_app/joseki.war`. (If you used a previous version of the support for Apache Jena, note that `joseki-config.ttl` is now placed under the `WEB-INF/classes` directory instead of under the top level directory of the web application.)

The supplied `joseki-config.ttl` file includes a section similar to the following for the Oracle data set:

```
#
## Datasets
#
[] ja:loadClass "oracle.spatial.rdf.client.jena.assembler.OracleAssemblerVocab" .

oracle:Dataset  rdfs:subClassOf  ja:RDFDataset .

<#oracle> rdf:type oracle:Dataset;
    joseki:poolSize     1 ;         ## Number of concurrent connections allowed to
this dataset.
    oracle:connection
    [ a oracle:OracleConnection ;
      oracle:dataSourceName "OracleSemDS"
    ];
    oracle:defaultModel [ oracle:firstModel "TEST_MODEL" ] .
```

In this section of the file, you can:

- Modify the `joseki:poolSize` value, which specifies the number of concurrent connections allowed to this Oracle data set (`<#oracle> rdf:type oracle:Dataset;`), which points to various RDF models in the database.
- Customize the name of the data source. The default name of *OracleSemDS* can be changed depending on your application requirements. The name, however, must match the data source name specified in Creating the Required Data Source Using WebLogic Server (page 7-6).
- Modify the name (or the object value of `oracle:firstModel` predicate) of the `defaultModel`, to use a different semantic model for queries. You can also specify multiple models, and one or more rulebases for this `defaultModel`.

For example, the following specifies two models (named ABOX and TBOX) and an OWLPRIME rulebase for the default model. Note that models specified using the `oracle:modelName` predicate must exist; they *will not* be created automatically.

```
<#oracle> rdf:type oracle:Dataset;
    joseki:poolSize 1 ; ## Number of concurrent connections allowed to this
dataset.
    oracle:connection
    [ a oracle:OracleConnection ;
      oracle:dataSourceName "OracleSemDS"
    ];
    oracle:defaultModel [ oracle:firstModel "ABOX";
                                    oracle:modelName "TBOX";
                                    oracle:rulebaseName "OWLPRIME" ] .
```

- Specify named graphs in the dataset. For example, you can create a named graph called `<http://G1>` based on two Oracle models and an entailment, as follows.

```
<#oracle> rdf:type oracle:Dataset;
    joseki:poolSize 1 ; ## Number of concurrent connections allowed to this
dataset.
      oracle:connection
      [ a oracle:OracleConnection ;
      ];
      oracle:namedModel [ oracle:firstModel "ABOX";
                          oracle:modelName "TBOX";
                          oracle:rulebaseName "OWLPRIME";
                          oracle:namedModelURI <http://G1> ]  .
```

  The object of `namedModel` can take the same specifications as `defaultModel`, so virtual models are supported here as well (see also the next item).

- Use a virtual model for queries by adding `oracle:useVM "TRUE"`, as shown in the following example. Note that if the specified virtual model does not exist, it *will* automatically be created on demand.

```
<#oracle> rdf:type oracle:Dataset;
    joseki:poolSize 1 ; ## Number of concurrent connections allowed to this
dataset.
    oracle:connection
    [ a oracle:OracleConnection ;
    ];
    oracle:defaultModel [ oracle:firstModel "ABOX";
                                    oracle:modelName "TBOX";
                                    oracle:rulebaseName "OWLPRIME";
                                    oracle:useVM "TRUE"
    ] .
```

  For more information, see Virtual Models Support (page 7-38).

- Specify a virtual model as the default model to answer SPARQL queries by using the predicate `oracle:virtualModelName`, as shown in the following example with a virtual model named `TRIPLE_DATA_VM_0`:

```
oracle:defaultModel [ oracle:virtualModelName "TRIPLE_DATA_VM_0" ] .
```

  If the underlying data consists of quads, you can use `oracle:virtualModelName` with `oracle:allGraphs`. The presence of `oracle:allGraphs` causes an instantiation of `DatasetGraphOracleSem` objects to answer named graph queries. An example is as follows:

```
oracle:allGraphs [ oracle:virtualModelName "QUAD_DATA_VM_0" ] .
```

Note that when a virtual model name is specified as the default graph, the endpoint can serve only query requests; SPARQL Update operations are not supported.

- Set the `queryOptions` and `inferenceMaintenance` properties to change the query behavior and inference update mode. (See the Javadoc for information about `QueryOptions` and `InferenceMaintenanceMode`.)

  By default, `QueryOptions.ALLOW_QUERY_INVALID_AND_DUP` and `InferenceMaintenanceMode.NO_UPDATE` are set, for maximum query flexibility and efficiency.

7.2.2.1 Client Identifiers

For every database connection created or used by the support for Apache Jena, a client identifier is associated with the connection. The client identifier can be helpful, especially in a Real Application Cluster (Oracle RAC) environment, for isolating RDF Semantic Graph support for Apache Jena-related activities from other database activities when you are doing performance analysis and tuning.

By default, the client identifier assigned is `JenaAdapter`. However, you can specify a different value by setting the Java VM `clientIdentifier` property using the following format:

```
-Doracle.spatial.rdf.client.jena.clientIdentifier=<identificationString>
```

To start the tracing of only RDF Semantic Graph support for Apache Jena-related activities on the database side, you can use the DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE procedure. For example:

```
SQL> EXECUTE DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE('JenaAdapter', true, true);
```

7.2.2.2 Using OLTP Compression for Application Tables and Staging Tables

By default, the support for Apache Jena creates the application tables and any staging tables (the latter used for bulk loading, as explained in Bulk Loading Using RDF Semantic Graph Support for Apache Jena (page 7-44)) using basic table compression with the following syntax:

```
CREATE TABLE .... (... column definitions ...) ... compress;
```

However, if you are licensed to use the Oracle Advanced Compression option no the database, you can set the following JVM property to turn on OLTP compression, which compresses data during all DML operations against the underlying application tables and staging tables:

```
-Doracle.spatial.rdf.client.jena.advancedCompression="compress for oltp"
```

7.2.3 Configuring the Fuseki-Based SPARQL Service

This topic briefly describes how to configure and set up the Fuseki-based SPARQL service that connects to Oracle Database.

The concepts and actions for using the Fuseki-based SPARQL service are similar to those for the Joseki-based SPARQL service, as explained in Configuring the Joseki-Based SPARQL Service (page 7-6). The information in that topic about client identifiers and OLTP compression also applies to the the Fuseki-based SPARQL service, and the ways to `defaultModel` and `allGraphs` are the same.

An example `oracle` service for the Fuseki-based service is provided in the `config-oracle.ttl` file. For detailed information, see "Fuseki: serving RDF data over HTTP" (http://jena.apache.org/documentation/serving_data/).

To start the Fuseki-based SPARQL service that connects to Oracle Database, enter the following command:

```
cd fuseki/
% ./fuseki-server  --config config-oracle.ttl
```

When Fuseki is running, you can check the status by going to a URL in the following format: `http://your-hostname:3030/`

7.2.4 Terminating Long-Running SPARQL Queries

Because some applications need to be able to terminate long-running SPARQL queries, an abort framework has been introduced with RDF Semantic Graph support for Apache Jena and the Joseki setup. Basically, for queries that may take a long time to run, you must stamp each with a unique query ID (qid) value.

For example, the following SPARQL query selects out the subject of all triples. A query ID (qid) is set so that this query can be terminated upon request.

```
PREFIX ORACLE_SEM_FS_NS:  <http://example.com/semtech#qid=8761>
SELECT ?subject WHERE {?subject ?property ?object }
```

The `qid` attribute value is of long integer type. You can choose a value for the `qid` for a particular query based on your own application needs.

To terminate a SPARQL query that has been submitted with a qid value, applications can send an abort request to a servlet in the following format and specify a matching QID value

```
http://<hostname>:7001/joseki/querymgt?abortqid=8761
```

7.2.5 N-Triples Encoding for Non-ASCII Characters

For any non-ASCII characters in the lexical representation of RDF resources, `\uHHHH` N-Triples encoding is used when the characters are inserted into the Oracle database. (For details about N-Triples encoding, see http://www.w3.org/TR/rdf-testcases/#ntrip_grammar.) Encoding of the constant resources in a SPARQL query is handled in a similar fashion.

Using `\uHHHH` N-Triples encoding enables support for international characters, such as a mix of Norwegian and Swedish characters, in the Oracle database even if a supported Unicode character set is not being used.

7.3 Setting Up the RDF Semantic Graph Environment

To use the support for Apache Jena to perform queries, you can connect as any user (with suitable privileges) and use any models in the semantic network. If your RDF Semantic Graph environment already meets the requirements, you can go directly to compiling and running Java code that uses the support for Apache Jena. If your RDF Semantic Graph environment is not yet set up to be able to use the support for Apache Jena, you can perform actions similar to the following example steps:

1. Connect as SYS with the SYSDBA role:

   ```
   sqlplus sys/<password-for-sys> as sysdba
   ```

2. Create a tablespace for the system tables. For example:

```
CREATE TABLESPACE rdf_users datafile 'rdf_users01.dbf'
    size 128M reuse autoextend on next 64M
    maxsize unlimited segment space management auto;
```

3. Create the semantic network. For example:

```
EXECUTE sem_apis.create_sem_network('RDF_USERS');
```

4. Create a database user (for connecting to the database to use the semantic network and the support for Apache Jena). For example:

```
CREATE USER rdfusr IDENTIFIED BY <password-for-udfusr>
                  DEFAULT TABLESPACE rdf_users;
```

5. Grant the necessary privileges to this database user. For example:

```
GRANT connect, resource TO rdfusr;
```

6. To use the support for Apache Jena with your own semantic data, perform the appropriate steps to store data, create a model, and create database indexes, as explained in Quick Start for Using Semantic Data (page 1-92). Then perform queries by compiling and running Java code; see Example Queries Using RDF Semantic Graph Support for Apache Jena (page 7-56) for information about example queries.

   To use the support for Apache Jena with supplied example data, see Example Queries Using RDF Semantic Graph Support for Apache Jena (page 7-56).

7.4 SEM_MATCH and RDF Semantic Graph Support for Apache Jena Queries Compared

There are two ways to query semantic data stored in Oracle Database: SEM_MATCH-based SQL statements and SPARQL queries through the support for Apache Jena. Queries using each approach are similar in appearance, but there are important behavioral differences. To ensure consistent application behavior, you must understand the differences and use care when dealing with query results coming from SEM_MATCH queries and SPARQL queries.

The following simple examples show the two approaches.

Query 1 (SEM_MATCH-based)

```
select s, p, o
    from table(sem_match('{?s ?p ?o}', sem_models('Test_Model'), ....))
```

Query 2 (SPARQL query through Support for Apache Jena)

```
select ?s ?p ?o
where {?s ?p ?o}
```

These two queries perform the same kind of functions; however, there are some important differences. Query 1 (SEM_MATCH-based):

- Reads all triples out of `Test_Model`.
- Does not differentiate among URI, bNode, plain literals, and typed literals, and it does not handle long literals.

- Does not unescape certain characters (such as `'\n'`).

Query 2 (SPARQL query executed through the support for Apache Jena) also reads all triples out of `Test_Model` (assume it executed a call to `ModelOracleSem` referring to the same underlying `Test_Model`). However, Query 2:

- Reads out additional columns (as opposed to just the `s`, `p`, and `o` columns with the SEM_MATCH table function), to differentiate URI, bNodes, plain literals, typed literals, and long literals. This is to ensure proper creation of Jena Node objects.
- Unescapes those characters that are escaped when stored in Oracle Database

Blank node handling is another difference between the two approaches:

- In a SEM_MATCH-based query, blank nodes are always treated as constants.
- In a SPARQL query, a blank node that *is not* wrapped inside < and > is treated as a variable when the query is executed through the support for Apache Jena. This matches the SPARQL standard semantics. However, a blank node that *is* wrapped inside < and > is treated as a constant when the query is executed, and the support for Apache Jena adds a proper prefix to the blank node label as required by the underlying data modeling.

The maximum length for the name of a semantic model created using the support for Apache Jena API is 22 characters.

7.5 Retrieving User-Friendly Java Objects from SEM_MATCH or SQL-Based Query Results

You can query a semantic graph using any of the following approaches:

- SPARQL (through Java methods or web service end point)
- SEM_MATCH (table function that has SPARQL queries embedded)
- SQL (by querying the MDSYS.RDFM_<*model*> view and joining with MDSYS.RDF_VALUE$ and/or other tables)

For Java developers, the results from the first approach are easy to consume. The results from the second and third approaches, however, can be difficult for Java developers because you must parse various columns to get properly typed Java objects that are mapped from typed RDF literals. RDF Semantic Graph support for Apache Jena supports several methods and helper functions to simplify the task of getting properly typed Java objects from a JDBC result set. These methods and helper functions are shown in the following examples:

- Example 7-1 (page 7-13)
- Example 7-2 (page 7-14)
- Example 7-3 (page 7-14)

These examples use a test table TGRAPH_TPL (and model TGRAPH based on it), into which a set of typed literals is added, as in the following code:

```
create table tgraph_tpl(triple sdo_rdf_triple_s);
exec sem_apis.create_sem_model('tgraph','tgraph_tpl','triple');
truncate table tgraph_tpl;

-- Add some triples
```

```
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph','<urn:s1>','<urn:p1>', '<urn:o1>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph','<urn:s2>','<urn:p2>', '"hello world"'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph','<urn:s3>','<urn:p3>', '"hello world"@en'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph','<urn:s4>','<urn:p4>', '" o1o "^^<http://
www.w3.org/2001/XMLSchema#string>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph','<urn:s4>','<urn:p4>', '"xyz"^^<http://
mytype>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph','<urn:s5>','<urn:p5>', '"123"^^<http://
www.w3.org/2001/XMLSchema#integer>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph','<urn:s5>','<urn:p5>', '"123.456"^^<http://
www.w3.org/2001/XMLSchema#double>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph','<urn:s6>','<urn:p6>', '_:bn1'));

-- Add some quads
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g1>','<urn:s1>','<urn:p1>', '<urn:o1>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g2>','<urn:s1>','<urn:p1>', '<urn:o1>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g2>','<urn:s2>','<urn:p2>', '"hello
world"'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g2>','<urn:s3>','<urn:p3>', '"hello
world"@en'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g2>','<urn:s4>','<urn:p4>', '" o1o
"^^<http://www.w3.org/2001/XMLSchema#string>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g2>','<urn:s4>','<urn:p4>',
'"xyz"^^<http://mytype>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g2>','<urn:s5>','<urn:p5>',
'"123"^^<http://www.w3.org/2001/XMLSchema#integer>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g2>','<urn:s5>','<urn:p5>',
'"123.456"^^<http://www.w3.org/2001/XMLSchema#double>'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g2>','<urn:s6>','<urn:p6>', '_:bn1'));
insert into tgraph_tpl values(sdo_rdf_triple_s('tgraph:<urn:g2>','<urn:s7>','<urn:p7>',
'"2002-10-10T12:00:00-05:00"^^<http://www.w3.org/2001/XMLSchema#dateTime>'));
```

Example 7-1 SQL-Based Graph Query

Example 7-1 (page 7-13) runs a pure SQL-based graph query and constructs Jena objects.

```
iTimeout = 0; // no time out
iDOP = 1;     // degree of parallelism
iStartColPos = 2;
queryString = "select 'hello'||rownum as extra,
o.VALUE_TYPE,o.LITERAL_TYPE,o.LANGUAGE_TYPE,o.LONG_VALUE,o.VALUE_NAME "
            + "  from mdsys.rdfm_tgraph g, mdsys.rdf_value$ o where
g.canon_end_node_id = o.value_id";

rs = oracle.executeQuery(queryString, iTimeout, iDOP, bindValues);

while (rs.next()) {
  node = OracleSemIterator.retrieveNodeFromRS(rs, iStartColPos,
OracleSemQueryPlan.CONST_FIVE_COL, translator);
  System.out.println("Result " + node.getClass().getName() + " = " + node + " " +
rs.getString(1));
}
```

Example 7-1 (page 7-13) might generate the following output:

```
Result com.hp.hpl.jena.graph.Node_URI = urn:o1 hello1
Result com.hp.hpl.jena.graph.Node_URI = urn:o1 hello2
Result com.hp.hpl.jena.graph.Node_Literal = "hello world" hello3
Result com.hp.hpl.jena.graph.Node_Literal = "hello world"@en hello4
Result com.hp.hpl.jena.graph.Node_Literal = " o1o " hello5
Result com.hp.hpl.jena.graph.Node_Literal = "xyz" hello6
```

```
Result com.hp.hpl.jena.graph.Node_Literal = "123"^^http://www.w3.org/2001/
XMLSchema#decimal hello7
Result com.hp.hpl.jena.graph.Node_Literal = "1.23456E2"^^http://www.w3.org/2001/
XMLSchema#double hello8
Result com.hp.hpl.jena.graph.Node_Blank = m8g3C75726E3A67323Egmbn1 hello9
Result com.hp.hpl.jena.graph.Node_Literal = "2002-10-10T17:00:00Z"^^http://
www.w3.org/2001/XMLSchema#dateTime hello10
Result com.hp.hpl.jena.graph.Node_Literal = "1.23456E2"^^http://www.w3.org/2001/
XMLSchema#double hello11
Result com.hp.hpl.jena.graph.Node_URI = urn:o1 hello12
Result com.hp.hpl.jena.graph.Node_Literal = "hello world" hello13
Result com.hp.hpl.jena.graph.Node_Literal = "hello world"@en hello14
Result com.hp.hpl.jena.graph.Node_Literal = " o1o " hello15
Result com.hp.hpl.jena.graph.Node_Literal = "xyz" hello16
Result com.hp.hpl.jena.graph.Node_Literal = "123"^^http://www.w3.org/2001/
XMLSchema#decimal hello17
Result com.hp.hpl.jena.graph.Node_Blank = m8mbn1 hello18
```

Example 7-2 Hybrid Query Mixing SEM_MATCH with Regular SQL Constructs

Example 7-2 (page 7-14) uses the `OracleSemIterator.retrieveNodeFromRS` API to construct a Jena object by reading the five consecutive columns (in the exact order of value type, literal type, language type, long value, and value name), and by performing the necessary unescaping and object instantiations. This example bypasses SEM_MATCH and directly joins the graph view with MDSYS.RDF_VALUE$.

```
iStartColPos = 1;
queryString = "select  g$RDFVTYP, g, count(1) as cnt "
            + "  from table(sem_match('{ GRAPH ?g { ?s ?p ?
o . } }',sem_models('tgraph'),null,null,null,null,null)) "
            + " group by g$RDFVTYP, g";

rs = oracle.executeQuery(queryString, iTimeout, iDOP, bindValues);
while (rs.next()) {
  node = OracleSemIterator.retrieveNodeFromRS(rs, iStartColPos,
OracleSemQueryPlan.CONST_TWO_COL, translator);
  System.out.println("Result " + node.getClass().getName() + " = " + node + " " +
rs.getInt(iStartColPos + 2));
}
```

Example 7-2 (page 7-14) might generate the following output:

```
Result com.hp.hpl.jena.graph.Node_URI = urn:g2 9
Result com.hp.hpl.jena.graph.Node_URI = urn:g1 1
```

In Example 7-2 (page 7-14):

- The helper function `executeQuery` in the `Oracle` class is used to run the SQL statement, and the `OracleSemIterator.retrieveNodeFromRS` API (also used in Example 7-1 (page 7-13)) is used to construct Jena objects.
- Only two columns are used in the output: value type (g$RDFVTYP) and value name (g), it is known that this g variable can never be a literal RDF resource.
- The column order is significant. For a two-column variable, the first column must be the value type and the second column must be the value name.

Example 7-3 SEM_MATCH Query

Example 7-3 (page 7-14) runs a SEM_MATCH query and constructs an iterator (instance of `OracleSemIterator`) that returns a list of Jena objects.

```
queryString = "select  g$RDFVTYP, g, s$RDFVTYP, s, p$RDFVTYP, p, o$RDFVTYP,o
$RDFLTYP,o$RDFLANG,o$RDFCLOB,o "
            + "  from table(sem_match('{ GRAPH ?g { ?s ?p ?
o . } }',sem_models('tgraph'),null,null,null,null,null))";

guide = new ArrayList<String>();
guide.add(OracleSemQueryPlan.CONST_TWO_COL);
guide.add(OracleSemQueryPlan.CONST_TWO_COL);
guide.add(OracleSemQueryPlan.CONST_TWO_COL);
guide.add(OracleSemQueryPlan.CONST_FIVE_COL);

rs = oracle.executeQuery(queryString, iTimeout, iDOP, bindValues);
osi = new OracleSemIterator(rs);
osi.setGuide(guide);
osi.setTranslator(translator);

while (osi.hasNext()) {
  result = osi.next();
  System.out.println("Result " + result.getClass().getName() + " = " + result);
}
```

Example 7-3 (page 7-14) might generate the following output:

```
Result com.hp.hpl.jena.graph.query.Domain = [urn:g1, urn:s1, urn:p1, urn:o1]
Result com.hp.hpl.jena.graph.query.Domain = [urn:g2, urn:s1, urn:p1, urn:o1]
Result com.hp.hpl.jena.graph.query.Domain = [urn:g2, urn:s2, urn:p2, "hello world"]
Result com.hp.hpl.jena.graph.query.Domain = [urn:g2, urn:s3, urn:p3, "hello
world"@en]
Result com.hp.hpl.jena.graph.query.Domain = [urn:g2, urn:s4, urn:p4, " o1o "]
Result com.hp.hpl.jena.graph.query.Domain = [urn:g2, urn:s4, urn:p4, "xyz"]
Result com.hp.hpl.jena.graph.query.Domain = [urn:g2, urn:s5, urn:p5, "123"^^http://
www.w3.org/2001/XMLSchema#decimal]
Result com.hp.hpl.jena.graph.query.Domain = [urn:g2, urn:s5, urn:p5,
"1.23456E2"^^http://www.w3.org/2001/XMLSchema#double]
Result com.hp.hpl.jena.graph.query.Domain = [urn:g2, urn:s6, urn:p6,
m8g3C75726E3A67323Egmbn1]
Result com.hp.hpl.jena.graph.query.Domain = [urn:g2, urn:s7, urn:p7,
"2002-10-10T17:00:00Z"^^http://www.w3.org/2001/XMLSchema#dateTime]
```

In Example 7-3 (page 7-14):

- `OracleSemIterator` takes in a JDBC result set. `OracleSemIterator` needs guidance on parsing all the columns that represent the bind values of SPARQL variables. A guide is simply a list of string values. Two constants have been defined to differentiate a 2-column variable (for subject or predicate position) from a 5-column variable (for object position). A translator is also required.
- Four variables are used in the output. The first three variables are not RDF literal resources, so CONST_TWO_COL is used as their guide. The last variable can be an RDF literal resource, so CONST_FIVE_COL is used as its guide.
- The column order is significant, and it must be as shown in the example.

7.6 Optimized Handling of SPARQL Queries

This section describes some performance-related features of the support for Apache Jena that can enhance SPARQL query processing. These features are performed automatically by default.

This section assumes that you are familiar with SPARQL, including the CONSTRUCT feature and property paths.

7.6.1 Compilation of SPARQL Queries to a Single SEM_MATCH Call

SPARQL queries involving DISTINCT, OPTIONAL, FILTER, UNION, ORDER BY, and LIMIT are converted to a single Oracle SEM_MATCH table function. If a query cannot be converted directly to SEM_MATCH because it uses SPARQL features not supported by SEM_MATCH (for example, CONSTRUCT), the support for Apache Jena employs a hybrid approach and tries to execute the largest portion of the query using a single SEM_MATCH function while executing the rest using the Jena ARQ query engine.

For example, the following SPARQL query is directly translated to a single SEM_MATCH table function:

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person ?name
  WHERE {
                  {?alice foaf:knows ?person . }
                UNION {
                  ?person ?p ?name. OPTIONAL { ?person ?x ?name1 }
                      }
        }
```

However, the following example query is not directly translatable to a single SEM_MATCH table function because of the CONSTRUCT keyword:

```
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT  { <http://example.org/person#Alice> vcard:FN ?obj }
  WHERE  { { ?x <http://pred/a> ?obj.}
         UNION
         { ?x <http://pred/b> ?obj.}  }
```

In this case, the support for Apache Jena converts the inner UNION query into a single SEM_MATCH table function, and then passes on the result set to the Jena ARQ query engine for further evaluation.

7.6.2 Optimized Handling of Property Paths

As defined in Jena, a property path is a possible route through an RDF graph between two graph nodes. Property paths are an extension of SPARQL and are more expressive than basic graph pattern queries, because regular expressions can be used over properties for pattern matching RDF graphs. For more information about property paths, see the documentation for the Jena ARQ query engine.

RDF Semantic Graph support for Apache Jena supports all Jena property path types through the integration with the Jena ARQ query engine, but it converts some common path types directly to native SQL hierarchical queries (not based on SEM_MATCH) to improve performance. The following types of property paths are directly converted to SQL by the support for Apache Jena when dealing with triple data:

- Predicate alternatives: (p1 | p2 | … | p*n*) where p*i* is a property URI
- Predicate sequences: (p1 / p2 / … / p*n*) where p*i* is a property URI

- Reverse paths : (^ p) where p is a predicate URI
- Complex paths: p+, p*, p{0, *n*} where p could be an alternative, sequence, reverse path, or property URI

Path expressions that cannot be captured in this grammar are not translated directly to SQL by the support for Apache Jena, and they are answered using the Jena query engine.

The following example contains a code snippet using a property path expression with path sequences:

```
String m = "PROP_PATH";

ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle, m);

GraphOracleSem graph = new GraphOracleSem(oracle, m);

// populate the RDF Graph
    graph.add(Triple.create(Node.createURI("http://a"),
    Node.createURI("http://p1"),
    Node.createURI("http://b")));

graph.add(Triple.create(Node.createURI("http://b"),
 Node.createURI("http://p2"),
 Node.createURI("http://c")));

graph.add(Triple.create(Node.createURI("http://c"),
 Node.createURI("http://p5"),
 Node.createURI("http://d")));

String query =
" SELECT ?s  " +
" WHERE {?s (<http://p1>/<http://p2>/<http://p5>)+ <http://d>.}";

QueryExecution qexec =
      QueryExecutionFactory.create(QueryFactory.create(query,
 Syntax.syntaxARQ), model);

try {
  ResultSet results = qexec.execSelect();
  ResultSetFormatter.out(System.out, results);
}
finally {
  if (qexec != null)
    qexec.close();
}

OracleUtils.dropSemanticModel(oracle, m);
model.close();
```

7.7 Additions to the SPARQL Syntax to Support Other Features

RDF Semantic Graph support for Apache Jena allows you to pass in hints and additional query options. It implements these capabilities by overloading the SPARQL namespace prefix syntax by using Oracle-specific namespaces that contain query options. The namespaces are in the form *PREFIX ORACLE_SEM*_xx_NS, where *xx* indicates the type of feature (such as HT for hint or AP for additional predicate)

7.7.1 SQL Hints

SQL hints can be passed to a SEM_MATCH query including a line in the following form:

```
PREFIX ORACLE_SEM_HT_NS: <http://oracle.com/semtech#hint>
```

Where *hint* can be any hint supported by SEM_MATCH. For example:

```
PREFIX ORACLE_SEM_HT_NS: <http://oracle.com/semtech#leading(t0,t1)>
SELECT ?book ?title ?isbn
WHERE { ?book <http://title> ?title. ?book <http://ISBN> ?isbn }
```

In this example, `t0,t1` refers to the first and second patterns in the query.

Note the slight difference in specifying hints when compared to SEM_MATCH. Due to restrictions of namespace value syntax, a comma (`,`) must be used to separate `t0` and `t1` (or other hint components) instead of a space.

For more information about using SQL hints, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22), specifically the material about the `HINT0` keyword in the `options` attribute.

7.7.2 Using Bind Variables in SPARQL Queries

In Oracle Database, using bind variables can reduce query parsing time and increase query efficiency and concurrency. Bind variable support in SPARQL queries is provided through namespace pragma specifications similar to ORACLE_SEM_FS_NS.

Consider a case where an application runs two SPARQL queries, where the second (Query 2) depends on the partial or complete results of the first (Query 1). Some approaches that do not involve bind variables include:

- Iterating through results of Query 1 and generating a set of queries. (However, this approach requires as many queries as the number of results of Query 1.)
- Constructing a SPARQL filter expression based on results of Query 1.
- Treating Query 1 as a subquery.

Another approach in this case is to use bind variables, as in the following sample scenario:

Query 1:

```
  SELECT ?x
    WHERE { ... <some complex query> ... };
```

Query 2:

```
  SELECT ?subject ?x
    WHERE {?subject <urn:related> ?x .};
```

The following example shows Query 2 with the syntax for using bind variables with the support for Apache Jena:

```
PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#no_fall_back,s2s>
PREFIX ORACLE_SEM_UEAP_NS: <http://oracle.com/semtech#x$RDFVID%20in(?,?,?)>
PREFIX ORACLE_SEM_UEPJ_NS: <http://oracle.com/semtech#x$RDFVID>
PREFIX ORACLE_SEM_UEBV_NS: <http://oracle.com/semtech#1,2,3>
```

```
SELECT ?subject ?x
WHERE {
  ?subject <urn:related>  ?x
};
```

This syntax includes using the following namespaces:

- ORACLE_SEM_UEAP_NS is like ORACLE_SEM_AP_NS, but the value portion of ORACLE_SEM_UEAP_NS is URL Encoded. Before the value portion is used, it must be URL decoded, and then it will be treated as an additional predicate to the SPARQL query.

  In this example, after URL decoding, the value portion (following the # character) of this ORACLE_SEM_UEAP_NS prefix becomes "`x$RDFVID in(?,?,?)`". The three question marks imply a binding to three values coming from Query 1.

- ORACLE_SEM_UEPJ_NS specifies the additional projections involved. In this case, because ORACLE_SEM_UEAP_NS references the x$RDFVID column, which does not appear in the SELECT clause of the query, it must be specified. Multiple projections are separated by commas.

- ORACLE_SEM_UEBV_NS specifies the list of bind values that are URL encoded first, and then concatenated and delimited by commas.

Conceptually, the preceding example query is equivalent to the following non-SPARQL syntax query, in which 1, 2, and 3 are treated as bind values:

```
SELECT ?subject ?x
  WHERE {
    ?subject <urn:related>  ?x
  }
  AND ?x$RDFVID in (1,2,3);
```

In the preceding SPARQL example of Query 2, the three integers 1, 2, and 3 come from Query 1. You can use the `oext:build-uri-for-id` function to generate such internal integer IDs for RDF resources. The following example gets the internal integer IDs from Query 1:

```
PREFIX oext: <http://oracle.com/semtech/jena-adaptor/ext/function#>
SELECT ?x  (oext:build-uri-for-id(?x) as ?xid)
WHERE { ... <some complex query> ... };
```

The values of `?xid` have the form of <rdfvid:*integer-value*>. The application can strip out the angle brackets and the "rdfvid:" strings to get the integer values and pass them to Query 2.

Consider another case, with a single query structure but potentially many different constants. For example, the following SPARQL query finds the hobby for each user who has a hobby and who logs in to an application. Obviously, different users will provide different <uri> values to this SPARQL query, because users of the application are represented using different URIs.

```
SELECT ?hobby
  WHERE { <uri> <urn:hasHobby> ?hobby };
```

One approach, which would not use bind variables, is to generate a different SPARQL query for each different <uri> value. For example, user Jane Doe might trigger the execution of the following SPARQL query:

```
SELECT ?hobby WHERE {
<http://www.example.com/Jane_Doe> <urn:hasHobby> ?hobby };
```

However, another approach is to use bind variables, as in the following example specifying user Jane Doe:

```
PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#no_fall_back,s2s>
PREFIX ORACLE_SEM_UEAP_NS: <http://oracle.com/semtech#subject$RDFVID
%20in(ORACLE_ORARDF_RES2VID(?))>
PREFIX ORACLE_SEM_UEPJ_NS: <http://oracle.com/semtech#subject$RDFVID>
PREFIX ORACLE_SEM_UEBV_NS: <http://oracle.com/semtech#http%3a%2f%2fwww.example.com
%2fJohn_Doe>
SELECT ?subject ?hobby
  WHERE {
    ?subject <urn:hasHobby>  ?hobby
  };
```

Conceptually, the preceding example query is equivalent to the following non-SPARQL syntax query, in which `http://www.example.com/Jane_Doe` is treated as a bind variable:

```
SELECT ?subject ?hobby
WHERE {
  ?subject <urn:hasHobby>  ?hobby
}
AND ?subject$RDFVID in (ORACLE_ORARDF_RES2VID('http://www.example.com/Jane_Doe'));
```

In this example, ORACLE_ORARDF_RES2VID is a function that translates URIs and literals into their internal integer ID representation. This function is created automatically when the support for Apache Jena is used to connect to an Oracle database.

7.7.3 Additional WHERE Clause Predicates

The SEM_MATCH `filter` attribute can specify additional selection criteria as a string in the form of a WHERE clause without the WHERE keyword. Additional WHERE clause predicates can be passed to a SEM_MATCH query including a line in the following form:

```
PREFIX ORACLE_SEM_AP_NS: <http://oracle.com/semtech#pred>
```

Where *pred* reflects the WHERE clause content to be appended to the query. For example:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ORACLE_SEM_AP_NS:<http://www.oracle.com/semtech#label$RDFLANG='fr'>
SELECT DISTINCT ?inst ?label
  WHERE { ?inst a <http://someCLass>. ?inst rdfs:label ?label . }
  ORDER BY (?label) LIMIT 20
```

In this example, a restriction is added to the query that the language type of the label variable must be `'fr'`.

7.7.4 Additional Query Options

Additional query options can be passed to a SEM_MATCH query including a line in the following form:

```
PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#option>
```

Where *option* reflects a query option (or multiple query options delimited by commas) to be appended to the query. For example:

```
PREFIX ORACLE_SEM_FS_NS:
<http://oracle.com/semtech#timeout=3,dop=4,INF_ONLY,ORDERED,ALLOW_DUP=T>
SELECT * WHERE {?subject ?property ?object }
```

The following query options are supported:

- `ALLOW_DUP=t` chooses a faster way to query multiple semantic models, although duplicate results may occur.
- `BEST_EFFORT_QUERY=t`, when used with the `TIMEOUT=`*n* option, returns all matches found in *n* seconds for the SPARQL query.
- `DEGREE=`*n* specifies, at the statement level, the degree of parallelism (*n*) for the query. With multi-core or multi-CPU processors, experimenting with different `DOP` values (such as 4 or 8) may improve performance.

  Contrast `DEGREE` with `DOP`, which specifies parallelism at the session level. `DEGREE` is recommended over `DOP` for use with the support for Apache Jena, because `DEGREE` involves less processing overhead.
- `DOP=`*n* specifies, at the session level, the degree of parallelism (*n*) for the query. With multi-core or multi-CPU processors, experimenting with different `DOP` values (such as 4 or 8) may improve performance.
- `INF_ONLY` causes only the inferred model to be queried.
- `JENA_EXECUTOR` disables the compilation of SPARQL queries to SEM_MATCH (or native SQL); instead, the Jena native query executor will be used.
- `JOIN=`*n* specifies how results from a SPARQL SERVICE call to a federated query can be joined with other parts of the query. For information about federated queries and the `JOIN` option, see JOIN Option and Federated Queries (page 7-22).
- `NO_FALL_BACK` causes the underlying query execution engine not to fall back on the Jena execution mechanism if a SQL exception occurs.
- `ODS=`*n* specifies, at the statement level, the level of dynamic sampling. (For an explanation of dynamic sampling, see the section about estimating statistics with dynamic sampling in *Oracle Database SQL Tuning Guide*.) Valid values for *n* are 1 through 10. For example, you could try `ODS=3` for complex queries.
- `ORDERED` is translated to a LEADING SQL hint for the query triple pattern joins, while performing the necessary RDF_VALUE$ joins last.
- `PLAIN_SQL_OPT=F` disables the native compilation of queries directly to SQL.
- `QID=`*n* specifies a query ID number; this feature can be used to cancel the query if it is not responding.
- `RESULT_CACHE` uses the Oracle RESULT_CACHE directive for the query.
- `REWRITE=F` disables ODCI_Table_Rewrite for the SEM_MATCH table function.
- `S2S` (SPARQL to pure SQL) causes the underlying SEM_MATCH-based query or queries generated based on the SPARQL query to be further converted into SQL queries ***without*** using the SEM_MATCH table function. The resulting SQL queries are executed by the Oracle cost-based optimizer, and the results are processed by the support for Apache Jena before being passed on to the client. For

more information about the S2S option, including benefits and usage information, see S2S Option Benefits and Usage Information (page 7-23).

S2S is enabled by default for all SPARQL queries. If you want to disable S2S, set the following JVM system property:

```
-Doracle.spatial.rdf.client.jena.defaultS2S=false
```

- SKIP_CLOB=T causes CLOB values not to be returned for the query.
- STRICT_DEFAULT=F allows the default graph to include triples in named graphs. (By default, STRICT_DEFAULT=T restricts the default graph to unnamed triples when no data set information is specified.)
- TIMEOUT=*n* (query timeout) specifies the number of seconds (*n*) that the query will run until it is terminated. The underlying SQL generated from a SPARQL query can return many matches and can use features like subqueries and assignments, all of which can take considerable time. The TIMEOUT and BEST_EFFORT_QUERY=t options can be used to prevent what you consider excessive processing time for the query.

7.7.4.1 JOIN Option and Federated Queries

A SPARQL federated query, as described in W3C documents, is a query "over distributed data" that entails "querying one source and using the acquired information to constrain queries of the next source." For more information, see *SPARQL 1.1 Federation Extensions* (http://www.w3.org/2009/sparql/docs/fed/service).

You can use the JOIN option (described in Additional Query Options (page 7-20)) and the SERVICE keyword in a federated query that uses the support for Apache Jena. For example, assume the following query:

```
SELECT ?s ?s1 ?o
 WHERE { ?s1 ?p1 ?s .
                    {
                     SERVICE <http://sparql.org/books> { ?s ?p ?o }
                    }
                 }
```

If the *local* query portion (?s1 ?p1 ?s,) is very selective, you can specify join=2, as shown in the following query:

```
PREFIX ORACLE_SEM_FS_NS:   <http://oracle.com/semtech#join=2>
SELECT ?s ?s1 ?o
 WHERE { ?s1 ?p1 ?s .
                    {
                     SERVICE <http://sparql.org/books> { ?s ?p ?o }
                    }
                 }
```

In this case, the local query portion (?s1 ?p1 ?s,) is executed locally against the Oracle database. Each binding of ?s from the results is then pushed into the SERVICE part (remote query portion), and a call is made to the service endpoint specified. Conceptually, this approach is somewhat like nested loop join.

If the *remote* query portion (?s ?s1 ?o) is very selective, you can specify join=3, as shown in the following query, so that the remote portion is executed first and results are used to drive the execution of local portion:

```
PREFIX ORACLE_SEM_FS_NS:   <http://oracle.com/semtech#join=3>
SELECT ?s ?s1 ?o
```

```
 WHERE { ?s1 ?p1 ?s .
                    {
                     SERVICE <http://sparql.org/books> { ?s ?p ?o }
                    }
                  }
```

In this case, a single call is made to the remote service endpoint and each binding of `?s` triggers a local query. As with `join=2`, this approach is conceptually a nested loop based join, but the difference is that the order is switched.

If neither the local query portion nor the remote query portion is very selective, then we can choose join=1, as shown in the following query:

```
PREFIX ORACLE_SEM_FS_NS:   <http://oracle.com/semtech#join=1>
SELECT ?s ?s1 ?o
 WHERE { ?s1 ?p1 ?s .
                    {
                     SERVICE <http://sparql.org/books> { ?s ?p ?o }
                    }
                }
```

In this case, the remote query portion and the local portion are executed independently, and the results are joined together by Jena. Conceptually, this approach is somewhat like a hash join.

For debugging or tracing federated queries, you can use the HTTP Analyzer in Oracle JDeveloper to see the underlying SERVICE calls.

7.7.4.2 S2S Option Benefits and Usage Information

The `S2S` option, described in Additional Query Options (page 7-20), provides the following potential benefits:

- It works well with the `RESULT_CACHE` option to improve query performance. Using the `S2S` and `RESULT_CACHE` options is especially helpful for queries that are executed frequently.
- It reduces the parsing time of the SEM_MATCH table function, which can be helpful for applications that involve many dynamically generated SPARQL queries.
- It eliminates the limit of 4000 bytes for the query body (the first parameter of the SEM_MATCH table function), which means that longer, more complex queries are supported.

The S2S option causes an internal in-memory cache to be used for translated SQL query statements. The default size of this internal cache is 1024 (that is, 1024 SQL queries); however, you can adjust the size by using the following Java VM property:

```
-Doracle.spatial.rdf.client.jena.queryCacheSize=<size>
```

7.7.5 Midtier Resource Caching

When semantic data is stored, all of the resource values are hashed into IDs, which are stored in the triples table. The mappings from value IDs to full resource values are stored in the MDSYS.RDF_VALUE$ table. At query time, for each selected variable, Oracle Database must perform a join with the RDF_VALUE$ table to retrieve the resource.

However, to reduce the number of joins, you can use the midtier cache option, which causes an in-memory cache on the middle tier to be used for storing mappings

between value IDs and resource values. To use this feature, include the following PREFIX pragma in the SPARQL query:

```
PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#midtier_cache>
```

To control the maximum size (in bytes) of the in-memory cache, use the `oracle.spatial.rdf.client.jena.cacheMaxSize` system property. The default cache maximum size is 1GB.

Midtier resource caching is most effective for queries using ORDER BY or DISTINCT (or both) constructs, or queries with multiple projection variables. Midtier cache can be combined with the other options specified in Additional Query Options (page 7-20).

If you want to pre-populate the cache with all of the resources in a model, use the `GraphOracleSem.populateCache` or `DatasetGraphOracleSem.populateCache` method. Both methods take a parameter specifying the number of threads used to build the internal midtier cache. Running either method in parallel can significantly increase the cache building performance on a machine with multiple CPUs (cores).

7.8 Functions Supported in SPARQL Queries through RDF Semantic Graph Support for Apache Jena

SPARQL queries through the support for Apache Jena can use the following kinds of functions:

- Functions in the function library of the Jena ARQ query engine
- Native Oracle Database functions for projected variables
- User-defined functions

7.8.1 Functions in the ARQ Function Library

SPARQL queries through the support for Apache Jena can use functions in the function library of the Jena ARQ query engine. These queries are executed in the middle tier.

The following examples use the `upper-case` and `namespace` functions. In these examples, the prefix `fn` is `<http://www.w3.org/2005/xpath-functions#>` and the prefix `afn` is `<http://jena.hpl.hp.com/ARQ/function#>`.

```
PREFIX  fn: <http://www.w3.org/2005/xpath-functions#>
PREFIX  afn: <http://jena.hpl.hp.com/ARQ/function#>
SELECT (fn:upper-case(?object) as ?object1)
WHERE { ?subject dc:title ?object }

PREFIX  fn: <http://www.w3.org/2005/xpath-functions#>
PREFIX  afn: <http://jena.hpl.hp.com/ARQ/function#>
SELECT ?subject (afn:namespace(?object) as ?object1)
WHERE { ?subject <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?object }
```

7.8.2 Native Oracle Database Functions for Projected Variables

SPARQL queries through the support for Apache Jena can use native Oracle Database functions for projected variables. These queries and the functions are executed inside the database. Note that the functions described in this section should not be used together with ARQ functions (described in Functions in the ARQ Function Library (page 7-24)).

This section lists the supported native functions and provides some examples. In the examples, the prefix `oext` is `<http://oracle.com/semtech/jena-adaptor/ext/function#>`.

> **Note:**
>
> In the preceding URL, note the spelling `jena-adapt`**`or`**, which is retained for compatibility with existing applications and which must be used in queries. The *adapt**er*** spelling is used in regular text, to follow Oracle documentation style guidelines.

- **`oext:upper-literal`** converts literal values (except for long literals) to uppercase. For example:

```
PREFIX  oext: <http://oracle.com/semtech/jena-adaptor/ext/function#>
SELECT (oext:upper-literal(?object) as ?object1)
WHERE { ?subject dc:title ?object }
```

- **`oext:lower-literal`** converts literal values (except for long literals) to lowercase. For example:

```
PREFIX  oext: <http://oracle.com/semtech/jena-adaptor/ext/function#>
SELECT (oext:lower-literal(?object) as ?object1)
WHERE { ?subject dc:title ?object }
```

- **`oext:build-uri-for-id`** converts the value ID of a URI, bNode, or literal into a URI form. For example:

```
PREFIX  oext: <http://oracle.com/semtech/jena-adaptor/ext/function#>
SELECT (oext:build-uri-for-id(?object) as ?object1)
WHERE { ?subject dc:title ?object }
```

  An example of the output might be: `<rdfvid:1716368199350136353>`

  One use of this function is to allow Java applications to maintain in memory a mapping of those value IDs to the lexical form of URIs, bNodes, or literals. The MDSYS.RDF_VALUE$ table provides such a mapping in Oracle Database.

  For a given variable `?var`, if only `oext:build-uri-for-id(?var)` is projected, the query performance is likely to be faster because fewer internal table join operations are needed to answer the query.

- **`oext:literal-strlen`** returns the length of literal values (except for long literals). For example:

```
PREFIX  oext: <http://oracle.com/semtech/jena-adaptor/ext/function#>
SELECT (oext:literal-strlen(?object) as ?objlen)
WHERE { ?subject dc:title ?object }
```

7.8.3 User-Defined Functions

SPARQL queries through the support for Apache Jena can use user-defined functions that are stored in the database.

In the following example, assume that you want to define a string length function (`my_strlen`) that handles long literals (CLOB) as well as short literals. On the SPARQL query side, this function can be referenced under the namespace of `ouext`, which is `http://oracle.com/semtech/jena-adaptor/ext/user-def-function#`.

```
PREFIX  ouext: <http://oracle.com/semtech/jena-adaptor/ext/user-def-function#>
SELECT ?subject ?object (ouext:my_strlen(?object) as ?obj1)
WHERE { ?subject dc:title ?object }
```

Inside the database, functions including `my_strlen`, `my_strlen_cl`, `my_strlen_la`, `my_strlen_lt`, and `my_strlen_vt` are defined to implement this capability. Conceptually, the return values of these functions are mapped as shown in Table 7-1 (page 7-26).

Table 7-1 Functions and Return Values for my_strlen Example

Function Name	Return Value
my_strlen	<VAR>
my_strlen_cl	<VAR>$RDFCLOB
my_strlen_la	<VAR>$RDFLANG
my_strlen_lt	<VAR>$RDFLTYP
my_strlen_vt	<VAR>$RDFVTYP

A set of functions (five in all) is used to implement a user-defined function that can be referenced from SPARQL, because this aligns with the internal representation of an RDF resource (in MDSYS.RDF_VALUE$). There are five major columns describing an RDF resource in terms of its value, language, literal type, long value, and value type, and these five columns can be selected out using SEM_MATCH. In this context, a user-defined function simply converts one RDF resource that is represented by five columns to another RDF resource.

These functions are defined as follows:

```
create or replace function my_strlen(rdfvtyp in varchar2,
                             rdfltyp in varchar2,
                             rdflang in varchar2,
                             rdfclob in clob,
                             value   in varchar2
                             ) return varchar2
as
  ret_val  varchar2(4000);
begin
  -- value
  if (rdfvtyp = 'LIT') then
    if (rdfclob is null) then
      return length(value);
    else
      return dbms_lob.getlength(rdfclob);
    end if;
  else
    -- Assign -1 for non-literal values so that application can
    -- easily differentiate
    return '-1';
  end if;
end;
/

create or replace function my_strlen_cl(rdfvtyp in varchar2,
                             rdfltyp in varchar2,
                             rdflang in varchar2,
```

```
                              rdfclob in clob,
                              value   in varchar2
                              ) return clob
as
begin
  return null;
end;
/

create or replace function my_strlen_la(rdfvtyp in varchar2,
                              rdfltyp in varchar2,
                              rdflang in varchar2,
                              rdfclob in clob,
                              value   in varchar2
                              ) return varchar2
as
begin
  return null;
end;
/

create or replace function my_strlen_lt(rdfvtyp in varchar2,
                              rdfltyp in varchar2,
                              rdflang in varchar2,
                              rdfclob in clob,
                              value   in varchar2
                              ) return varchar2
as
  ret_val  varchar2(4000);
begin
  -- literal type
  return 'http://www.w3.org/2001/XMLSchema#integer';
end;
/

create or replace function my_strlen_vt(rdfvtyp in varchar2,
                              rdfltyp in varchar2,
                              rdflang in varchar2,
                              rdfclob in clob,
                              value   in varchar2
                              ) return varchar2
as
  ret_val  varchar2(3);
begin
  return 'LIT';
end;
/
```

User-defined functions can also accept a parameter of VARCHAR2 type. The following five functions together define a `my_shorten_str` function that accepts an integer (in VARCHAR2 form) for the substring length and returns the substring. (The substring in this example is 12 characters, and it must not be greater than 4000 bytes.)

```
-- SPARQL query that returns the first 12 characters of literal values.
--
PREFIX  ouext: <http://oracle.com/semtech/jena-adaptor/ext/user-def-function#>
SELECT (ouext:my_shorten_str(?object, "12") as ?obj1) ?subject
WHERE { ?subject dc:title ?object }

create or replace function my_shorten_str(rdfvtyp in varchar2,
                            rdfltyp in varchar2,
```

```
                            rdflang in varchar2,
                            rdfclob in clob,
                            value   in varchar2,
                            arg     in varchar2
                            ) return varchar2
as
 ret_val  varchar2(4000);
begin
 -- value
 if (rdfvtyp = 'LIT') then
   if (rdfclob is null) then
     return substr(value, 1, to_number(arg));
   else
     return dbms_lob.substr(rdfclob, to_number(arg), 1);
   end if;
 else
   return null;
 end if;
end;
/

create or replace function my_shorten_str_cl(rdfvtyp in varchar2,
                            rdfltyp in varchar2,
                            rdflang in varchar2,
                            rdfclob in clob,
                            value   in varchar2,
                            arg     in varchar2
                            ) return clob
as
 ret_val  clob;
begin
 -- lob
 return null;
end;
/

create or replace function my_shorten_str_la(rdfvtyp in varchar2,
                            rdfltyp in varchar2,
                            rdflang in varchar2,
                            rdfclob in clob,
                            value   in varchar2,
                            arg     in varchar2
                            ) return varchar2
as
 ret_val  varchar2(4000);
begin
 -- lang
 if (rdfvtyp = 'LIT') then
   return rdflang;
 else
   return null;
 end if;
end;
/

create or replace function my_shorten_str_lt(rdfvtyp in varchar2,
                            rdfltyp in varchar2,
                            rdflang in varchar2,
                            rdfclob in clob,
                            value   in varchar2,
                            arg     in varchar2
```

```
                            ) return varchar2
as
 ret_val  varchar2(4000);
begin
 -- literal type
 ret_val := rdfltyp;
 return ret_val;
end;
/

create or replace function my_shorten_str_vt(rdfvtyp in varchar2,
                            rdfltyp in varchar2,
                            rdflang in varchar2,
                            rdfclob in clob,
                            value   in varchar2,
                            arg     in varchar2
                            ) return varchar2
as
 ret_val  varchar2(3);
begin
 return 'LIT';
end;
/
```

7.9 SPARQL Update Support

RDF Semantic Graph support for Apache Jena supports SPARQL Update (`http://www.w3.org/TR/sparql11-update/`), also referred to as SPARUL. The primary programming APIs involve the Jena class `UpdateAction` (in package `com.hp.hpl.jena.update`) and RDF Semantic Graph support for Apache Jena classes `GraphOracleSem` and `DatasetGraphOracleSem`. Example 7-4 (page 7-29) shows a SPARQL Update operation removes all triples in named graph `<http://example/graph>` from the relevant model stored in the database.

Example 7-4 Simple SPARQL Update

```
GraphOracleSem graphOracleSem = .... ;
DatasetGraphOracleSem dsgos = DatasetGraphOracleSem.createFrom(graphOracleSem);

// SPARQL Update operation
String szUpdateAction = "DROP GRAPH <http://example/graph>";

// Execute the Update against a DatasetGraph instance (can be a Jena Model as well)
UpdateAction.parseExecute(szUpdateAction, dsgos);
```

Note that Oracle Database does not keep any information about an empty named graph. This implies if you invoke CREATE GRAPH <*graph_name*> without adding any triples into this graph, then no additional rows in the application table or the underlying RDF_LINK$ table will be created. To an Oracle database, you can safely skip the CREATE GRAPH step, as is the case in Example 7-4 (page 7-29).

Example 7-5 SPARQL Update with Insert and Delete Operations

Example 7-5 (page 7-29) shows a SPARQL Update operation (from ARQ 2.8.8) involving multiple insert and delete operations.

```
PREFIX : <http://example/>
CREATE GRAPH <http://example/graph> ;
INSERT DATA { :r :p 123 } ;
INSERT DATA { :r :p 1066 } ;
DELETE DATA { :r :p 1066 } ;
```

```
INSERT DATA {
  GRAPH <http://example/graph> { :r :p 123 . :r :p 1066 }
} ;
DELETE DATA {
  GRAPH <http://example/graph>  { :r :p 123 }
}
```

After running the update operation in Example 7-5 (page 7-29) against an empty `DatasetGraphOracleSem`, running the SPARQL query `SELECT ?s ?p ?o WHERE {?s ?p ?o}` generates the following response:

```
-------------------------------------------------------------------------------------------
----------
| s                  | p                  |
o                                          |
===========================================================================================
==========
| <http://example/r> | <http://example/p> | "123"^^<http://www.w3.org/2001/
XMLSchema#decimal> |
-------------------------------------------------------------------------------------------
----------
```

Using the same data, running the SPARQL query `SELECT ?g ?s ?p ?o where {GRAPH ?g {?s ?p ?o}}` generates the following response:

```
-------------------------------------------------------------------------------------------
-----------------------------------
| g                      | s                  | p                  |
o                                            |
===========================================================================================
===================================
| <http://example/graph> | <http://example/r> | <http://example/p> | "1066"^^<http://
www.w3.org/2001/XMLSchema#decimal> |
-------------------------------------------------------------------------------------------
-----------------------------------
```

In addition to using the Java API for SPARQL Update operations, you can configure Joseki to accept SPARQL Update operations by removing the comment (##) characters at the start of the following lines in the `joseki-config.ttl` file.

```
## <#serviceUpdate>
##     rdf:type            joseki:Service ;
##     rdfs:label          "SPARQL/Update" ;
##     joseki:serviceRef   "update/service" ;
##     # dataset part
##     joseki:dataset      <#oracle>;
##     # Service part.
##     # This processor will not allow either the protocol,
##     # nor the query, to specify the dataset.
##     joseki:processor    joseki:ProcessorSPARQLUpdate
##     .
##
## <#serviceRead>
##     rdf:type            joseki:Service ;
##     rdfs:label          "SPARQL" ;
##     joseki:serviceRef   "sparql/read" ;
##     # dataset part
##     joseki:dataset      <#oracle> ;     ## Same dataset
##     # Service part.
##     # This processor will not allow either the protocol,
##     # nor the query, to specify the dataset.
```

```
##     joseki:processor    joseki:ProcessorSPARQL_FixedDS ;
##     .
```

After you edit the `joseki-config.ttl` file, you must restart the Joseki Web application. You can then try a simple update operation, as follows:

1. In your browser, go to: `http://<hostname>:7001/joseki/update.html`
2. Type or paste the following into the text box:

```
PREFIX : <http://example/>
INSERT DATA {
  GRAPH <http://example/g1> { :r :p 455 }
}
```

3. Click **Perform SPARQL Update**.

To verify that the update operation was successful, go to `http://<hostname>:7001/joseki` and enter the following query:

```
SELECT *
WHERE
  {  GRAPH <http://example/g1> {?s ?p ?o}};
```

The response should contain the following triple:

```
<http://example/r>     <http://example/p>    "455"^^<http://www.w3.org/2001/
XMLSchema#decimal>
```

A SPARQL Update can also be sent using an HTTP POST operation to the `http://<hostname>:7001/joseki/update/`service. For example, you can use `curl` (http://en.wikipedia.org/wiki/CURL) to send an HTTP POST request to perform the update operation:

```
curl --data "request=PREFIX%20%3A%20%3Chttp%3A%2F%2Fexample%2F%3E%20%0AINSERT%20DATA
%20%7B%0A%20%20GRAPH%20%3Chttp%3A%2F%2Fexample%2Fg1%3E%20%7B%20%3Ar%20%3Ap
%20888%20%7D%0A%7D%0A"  http://hostname:7001/joseki/update/service
```

In the preceding example, the URL encoded string is a simple INSERT operation into a named graph. After decoding, it reads as follows:

```
PREFIX : <http://example/>
INSERT DATA {
  GRAPH <http://example/g1> { :r :p 888 }
```

7.10 Analytical Functions for RDF Data

You can perform analytical functions on RDF data by using the `SemNetworkAnalyst` class in the `oracle.spatial.rdf.client.jena` package. This support integrates the Network Data Model Graph logic with the underlying RDF data structures. Therefore, to use analytical functions on RDF data, you must be familiar with the Network Data Model Graph feature, which is documented in *Oracle Spatial and Graph Topology Data Model and Network Data Model Graph Developer's Guide*.

The required NDM Java libraries, including `sdonm.jar` and `sdoutl.jar`, are under the directory `$ORACLE_HOME/md/jlib`. Note that `xmlparserv2.jar` (under `$ORACLE_HOME/xdk/lib`) must be included in the `classpath` definition.

Example 7-6 Performing Analytical functions on RDF Data

Example 7-6 (page 7-31) uses the `SemNetworkAnalyst` class, which internally uses the NDM `NetworkAnalyst` API

```
Oracle oracle = new Oracle(jdbcUrl, user, password);
GraphOracleSem graph = new GraphOracleSem(oracle, modelName);

Node nodeA = Node.createURI("http://A");
Node nodeB = Node.createURI("http://B");
Node nodeC = Node.createURI("http://C");
Node nodeD = Node.createURI("http://D");
Node nodeE = Node.createURI("http://E");
Node nodeF = Node.createURI("http://F");
Node nodeG = Node.createURI("http://G");
Node nodeX = Node.createURI("http://X");

// An anonymous node
Node ano = Node.createAnon(new AnonId("m1"));

Node relL = Node.createURI("http://likes");
Node relD = Node.createURI("http://dislikes");
Node relK = Node.createURI("http://knows");
Node relC = Node.createURI("http://differs");

graph.add(new Triple(nodeA, relL, nodeB));
graph.add(new Triple(nodeA, relC, nodeD));
graph.add(new Triple(nodeB, relL, nodeC));
graph.add(new Triple(nodeA, relD, nodeC));

graph.add(new Triple(nodeB, relD, ano));
graph.add(new Triple(nodeC, relL, nodeD));
graph.add(new Triple(nodeC, relK, nodeE));
graph.add(new Triple(ano,   relL, nodeD));
graph.add(new Triple(ano,   relL, nodeF));
graph.add(new Triple(ano,   relD, nodeB));

// X only likes itself
graph.add(new Triple(nodeX, relL, nodeX));

graph.commitTransaction();
HashMap<Node, Double> costMap = new HashMap<Node, Double>();
costMap.put(relL, Double.valueOf((double)0.5));
costMap.put(relD, Double.valueOf((double)1.5));
costMap.put(relC, Double.valueOf((double)5.5));

graph.setDOP(4); // this allows the underlying LINK/NODE tables
                 // and indexes to be created in parallel.

SemNetworkAnalyst sna = SemNetworkAnalyst.getInstance(
    graph,   // network data source
    true,    // directed graph
    true,    // cleanup existing NODE and LINK table
    costMap
    );

psOut.println("From nodeA to nodeC");
Node[] nodeArray = sna.shortestPathDijkstra(nodeA, nodeC);
printNodeArray(nodeArray, psOut);

psOut.println("From nodeA to nodeD");
nodeArray = sna.shortestPathDijkstra( nodeA, nodeD);
printNodeArray(nodeArray, psOut);

psOut.println("From nodeA to nodeF");
nodeArray = sna.shortestPathAStar(nodeA, nodeF);
```

```
printNodeArray(nodeArray, psOut);

psOut.println("From ano to nodeC");
nodeArray = sna.shortestPathAStar(ano, nodeC);
printNodeArray(nodeArray, psOut);

psOut.println("From ano to nodeX");
nodeArray = sna.shortestPathAStar(ano, nodeX);
printNodeArray(nodeArray, psOut);

graph.close();
oracle.dispose();
...
...

// A helper function to print out a path
public static void printNodeArray(Node[] nodeArray, PrintStream psOut)
{
  if (nodeArray == null) {
    psOut.println("Node Array is null");
    return;
  }
  if (nodeArray.length == 0) {psOut.println("Node Array is empty"); }
  int iFlag = 0;
  psOut.println("printNodeArray: full path starts");
  for (int iHops = 0; iHops < nodeArray.length; iHops++) {
    psOut.println("printNodeArray: full path item " + iHops + " = "
        + ((iFlag == 0) ? "[n] ":"[e] ") + nodeArray[iHops]);
    iFlag = 1 - iFlag;
  }
}
```

In Example 7-6 (page 7-31):

- A `GraphOracleSem` object is constructed and a few triples are added to the `GraphOracleSem` object. These triples describe several individuals and their relationships including *likes*, *dislikes*, *knows*, and *differs*.
- A cost mapping is constructed to assign a numeric cost value to different links/predicates (of the RDF graph). In this case, 0.5, 1.5, and 5.5 are assigned to predicates *likes*, *dislikes*, and *differs*, respectively. This cost mapping is optional. If the mapping is absent, then all predicates will be assigned the same cost 1. When cost mapping is specified, this mapping does not need to be complete; for predicates not included in the cost mapping, a default value of 1 is assigned.

The output of Example 7-6 (page 7-31) is as follows. In this output, the shortest paths are listed for the given start and end nodes. Note that the return value of `sna.shortestPathAStar(ano, nodeX)` is null because there is no path between these two nodes.

```
From nodeA to nodeC
printNodeArray: full path starts
printNodeArray: full path item 0 = [n] http://A          ## "n" denotes
Node
printNodeArray: full path item 1 = [e] http://likes      ## "e" denotes Edge (Link)
printNodeArray: full path item 2 = [n] http://B
printNodeArray: full path item 3 = [e] http://likes
printNodeArray: full path item 4 = [n] http://C

From nodeA to nodeD
printNodeArray: full path starts
```

```
printNodeArray: full path item 0 = [n] http://A
printNodeArray: full path item 1 = [e] http://likes
printNodeArray: full path item 2 = [n] http://B
printNodeArray: full path item 3 = [e] http://likes
printNodeArray: full path item 4 = [n] http://C
printNodeArray: full path item 5 = [e] http://likes
printNodeArray: full path item 6 = [n] http://D

From nodeA to nodeF
printNodeArray: full path starts
printNodeArray: full path item 0 = [n] http://A
printNodeArray: full path item 1 = [e] http://likes
printNodeArray: full path item 2 = [n] http://B
printNodeArray: full path item 3 = [e] http://dislikes
printNodeArray: full path item 4 = [n] m1
printNodeArray: full path item 5 = [e] http://likes
printNodeArray: full path item 6 = [n] http://F

From ano to nodeC
printNodeArray: full path starts
printNodeArray: full path item 0 = [n] m1
printNodeArray: full path item 1 = [e] http://dislikes
printNodeArray: full path item 2 = [n] http://B
printNodeArray: full path item 3 = [e] http://likes
printNodeArray: full path item 4 = [n] http://C

From ano to nodeX
Node Array is null
```

The underlying RDF graph view (SEMM_<*model_name*> or RDFM_<*model_name*>) cannot be used directly by NDM functions, and so `SemNetworkAnalyst` creates necessary tables that contain the nodes and links that are derived from a given RDF graph. These tables are not updated automatically when the RDF graph changes; rather, you can set the `cleanup` parameter in `SemNetworkAnalyst.getInstance` to `true`, to remove old node and link tables and to rebuild updated tables.

Example 7-7 Implementing NDM nearestNeighbors Function on Top of Semantic Data

Example 7-7 (page 7-34) implements the NDM `nearestNeighbors` function on top of semantic data. This gets a `NetworkAnalyst` object from the `SemNetworkAnalyst` instance, gets the node ID, creates `PointOnNet` objects, and processes `LogicalSubPath` objects.

```
%cat TestNearestNeighbor.java

import java.io.*;
import java.util.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import com.hp.hpl.jena.util.iterator.*;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.update.*;
import com.hp.hpl.jena.sparql.core.DatasetImpl;

import oracle.spatial.rdf.client.jena.*;

import oracle.spatial.rdf.client.jena.SemNetworkAnalyst;
import oracle.spatial.network.lod.LODGoalNode;
import oracle.spatial.network.lod.LODNetworkConstraint;
```

```
import oracle.spatial.network.lod.NetworkAnalyst;
import oracle.spatial.network.lod.PointOnNet;
import oracle.spatial.network.lod.LogicalSubPath;


/**
 * This class implements a nearestNeighbors function on top of semantic data
 * using public APIs provided in SemNetworkAnalyst and Oracle Spatial NDM
 */
public class TestNearestNeighbor
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];

    PrintStream psOut = System.out;

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);

    String szModelName = "test_nn";
    // First construct a TBox and load a few axioms
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle, szModelName);
    String insertString =
      " PREFIX my:  <http://my.com/> " +
      " INSERT DATA "                              +
      " { my:A   my:likes my:B .                " +
      "   my:A   my:likes my:C .                " +
      "   my:A   my:knows my:D .                " +
      "   my:A   my:dislikes my:X .             " +
      "   my:A   my:dislikes my:Y .             " +
      "   my:C   my:likes my:E .                " +
      "   my:C   my:likes my:F .                " +
      "   my:C   my:dislikes my:M .             " +
      "   my:D   my:likes my:G .                " +
      "   my:D   my:likes my:H .                " +
      "   my:F   my:likes my:M .                " +
      " }   ";
    UpdateAction.parseExecute(insertString,  model);

    GraphOracleSem g = model.getGraph();
    g.commitTransaction();
    g.setDOP(4);

    HashMap<Node, Double> costMap = new HashMap<Node, Double>();
    costMap.put(Node.createURI("http://my.com/likes"),    Double.valueOf(1.0));
    costMap.put(Node.createURI("http://my.com/dislikes"), Double.valueOf(4.0));
    costMap.put(Node.createURI("http://my.com/knows"),    Double.valueOf(2.0));

    SemNetworkAnalyst sna = SemNetworkAnalyst.getInstance(
        g,     // source RDF graph
        true,  // directed graph
        true,  // cleanup old Node/Link tables
        costMap
        );

    Node nodeStart = Node.createURI("http://my.com/A");
    long origNodeID = sna.getNodeID(nodeStart);

    long[] lIDs = {origNodeID};
```

```
    // translate from the original ID
    long nodeID = (sna.mapNodeIDs(lIDs))[0];

    NetworkAnalyst networkAnalyst = sna.getEmbeddedNetworkAnalyst();

    LogicalSubPath[] lsps = networkAnalyst.nearestNeighbors(
      new PointOnNet(nodeID),      // startPoint
      6,                           // numberOfNeighbors
      1,                           // searchLinkLevel
      1,                           // targetLinkLevel
      (LODNetworkConstraint) null, // constraint
      (LODGoalNode) null           // goalNodeFilter
      );

    if (lsps != null) {
      for (int idx = 0; idx < lsps.length; idx++) {
        LogicalSubPath lsp = lsps[idx];
        Node[] nodePath = sna.processLogicalSubPath(lsp, nodeStart);
        psOut.println("Path " + idx);
        printNodeArray(nodePath, psOut);
      }
    }

    g.close();
    sna.close();
    oracle.dispose();
  }


  public static void printNodeArray(Node[] nodeArray, PrintStream psOut)
  {
    if (nodeArray == null) {
      psOut.println("Node Array is null");
      return;
    }
    if (nodeArray.length == 0) {
      psOut.println("Node Array is empty");
    }
    int iFlag = 0;
    psOut.println("printNodeArray: full path starts");
    for (int iHops = 0; iHops < nodeArray.length; iHops++) {
      psOut.println("printNodeArray: full path item " + iHops + " = "
          + ((iFlag == 0) ? "[n] ":"[e] ") + nodeArray[iHops]);
      iFlag = 1 - iFlag;
    }
  }
}
```

The output of Example 7-7 (page 7-34) is as follows.

```
Path 0
printNodeArray: full path starts
printNodeArray: full path item 0 = [n] http://my.com/A
printNodeArray: full path item 1 = [e] http://my.com/likes
printNodeArray: full path item 2 = [n] http://my.com/C

Path 1
printNodeArray: full path starts
printNodeArray: full path item 0 = [n] http://my.com/A
printNodeArray: full path item 1 = [e] http://my.com/likes
```

```
printNodeArray: full path item 2 = [n] http://my.com/B

Path 2
printNodeArray: full path starts
printNodeArray: full path item 0 = [n] http://my.com/A
printNodeArray: full path item 1 = [e] http://my.com/knows
printNodeArray: full path item 2 = [n] http://my.com/D

Path 3
printNodeArray: full path starts
printNodeArray: full path item 0 = [n] http://my.com/A
printNodeArray: full path item 1 = [e] http://my.com/likes
printNodeArray: full path item 2 = [n] http://my.com/C
printNodeArray: full path item 3 = [e] http://my.com/likes
printNodeArray: full path item 4 = [n] http://my.com/E

Path 4
printNodeArray: full path starts
printNodeArray: full path item 0 = [n] http://my.com/A
printNodeArray: full path item 1 = [e] http://my.com/likes
printNodeArray: full path item 2 = [n] http://my.com/C
printNodeArray: full path item 3 = [e] http://my.com/likes
printNodeArray: full path item 4 = [n] http://my.com/F

Path 5
printNodeArray: full path starts
printNodeArray: full path item 0 = [n] http://my.com/A
printNodeArray: full path item 1 = [e] http://my.com/knows
printNodeArray: full path item 2 = [n] http://my.com/D
printNodeArray: full path item 3 = [e] http://my.com/likes
printNodeArray: full path item 4 = [n] http://my.com/H
```

7.10.1 Generating Contextual Information about a Path in a Graph

It is sometimes useful to see contextual information about a path in a graph, in addition to the path itself. The `buildSurroundingSubGraph` method in the `SemNetworkAnalyst` class can output a DOT file (graph description language file, extension `.gv`) into the specified `Writer` object. For each node in the path, up to ten direct neighbors are used to produce a surrounding subgraph for the path. The following example shows the usage of generating a DOT file with contextual information, specifically the output from the analytical functions used in Example 7-6 (page 7-31).

```
nodeArray = sna.shortestPathDijkstra(nodeA, nodeD);
printNodeArray(nodeArray, psOut);

FileWriter dotWriter = new FileWriter("Shortest_Path_A_to_D.gv");
sna.buildSurroundingSubGraph(nodeArray, dotWriter);
```

The generated output DOT file from the preceding example is straightforward, as shown in the following example:

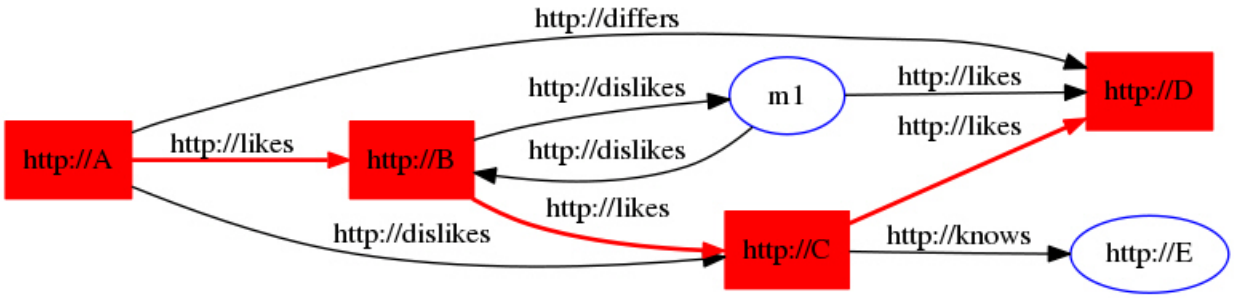
```
% cat Shortest_Path_A_to_D.gv
digraph { rankdir = LR; charset="utf-8";

"Rhttp://A" [ label="http://A" shape=rectangle,color=red,style = filled, ];
"Rhttp://B" [ label="http://B" shape=rectangle,color=red,style = filled, ];
"Rhttp://A" -> "Rhttp://B" [ label="http://likes"  color=red, style=bold, ];
"Rhttp://C" [ label="http://C" shape=rectangle,color=red,style = filled, ];
"Rhttp://A" -> "Rhttp://C" [ label="http://dislikes" ];
"Rhttp://D" [ label="http://D" shape=rectangle,color=red,style = filled, ];
```

```
"Rhttp://A" -> "Rhttp://D" [ label="http://differs" ];
"Rhttp://B" -> "Rhttp://C" [ label="http://likes"  color=red, style=bold, ];
"Rm1" [ label="m1" shape=ellipse,color=blue, ];
"Rhttp://B" -> "Rm1" [ label="http://dislikes" ];
"Rm1" -> "Rhttp://B" [ label="http://dislikes" ];
"Rhttp://C" -> "Rhttp://D" [ label="http://likes"  color=red, style=bold, ];
"Rhttp://E" [ label="http://E" shape=ellipse,color=blue, ];
"Rhttp://C" -> "Rhttp://E" [ label="http://knows" ];
"Rm1" -> "Rhttp://D" [ label="http://likes" ];
}
```

You can also use methods in the `SemNetworkAnalyst` and `GraphOracleSem` classes to produce more sophisticated visualization of the analytical function output.

You can convert the preceding DOT file into a variety of image formats. Figure 7-1 (page 7-38) is an image representing the information in the preceding DOT file.

Figure 7-1 Visual Representation of Analytical Function Output



7.11 Support for Server-Side APIs

This section describes some of the RDF Semantic Graph features that are exposed by RDF Semantic Graph support for Apache Jena. For comprehensive documentation of the API calls that support the available features, see the RDF Semantic Graph support for Apache Jena reference information (Javadoc). For additional information about the server-side features exposed by the support for Apache Jena, see the relevant chapters in this manual.

7.11.1 Virtual Models Support

Virtual models (explained in Virtual Models (page 1-13)) are specified in the `GraphOracleSem` constructor, and they are handled transparently. If a virtual model exists for the model-rulebase combination, it is used in query answering; if such a virtual model does not exist, it is created in the database.

Note:

Virtual model support through the support for Apache Jena is available only with Oracle Database Release 11.2 or later.

The following example reuses an existing virtual model.

```
String modelName = "EX";
String m1 = "EX_1";

ModelOracleSem defaultModel =
```

```
  ModelOracleSem.createOracleSemModel(oracle, modelName);

// create these models in case they don't exist
ModelOracleSem model1 = ModelOracleSem.createOracleSemModel(oracle, m1);

String vmName = "VM_" + modelName;


//create a virtual model containing EX and EX_1
oracle.executeCall(
"begin sem_apis.create_virtual_model(?,sem_models('"+ m1 + "','"+ modelName
+"'),null); end;",vmName);

String[] modelNames = {m1};
String[] rulebaseNames = {};

Attachment attachment = Attachment.createInstance(modelNames, rulebaseNames,
InferenceMaintenanceMode.NO_UPDATE, QueryOptions.ALLOW_QUERY_VALID_AND_DUP);

// vmName is passed to the constructor, so GraphOracleSem will use the virtual
// model named vmname (if the current user has read privileges on it)
GraphOracleSem graph = new GraphOracleSem(oracle, modelName, attachment, vmName);
graph.add(Triple.create(Node.createURI("urn:alice"),
                        Node.createURI("http://xmlns.com/foaf/0.1/mbox"),
                        Node.createURI("mailto:alice@example")));
ModelOracleSem model = new ModelOracleSem(graph);

String queryString =

   " SELECT ?subject ?object WHERE { ?subject ?p ?object } ";

Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;

try {
   ResultSet results = qexec.execSelect() ;
   for ( ; results.hasNext() ; ) {
      QuerySolution soln = results.nextSolution() ;
      psOut.println("soln " + soln);
   }
}
finally {
   qexec.close() ;
}

OracleUtils.dropSemanticModel(oracle, modelName);
OracleUtils.dropSemanticModel(oracle, m1);

oracle.dispose();
```

You can also use the GraphOracleSem constructor to create a virtual model, as in the following example:

```
GraphOracleSem graph = new GraphOracleSem(oracle, modelName, attachment, true);
```

In this example, the fourth parameter (`true`) specifies that a virtual model needs to be created for the specified `modelName` and `attachment`.

7.11.2 Connection Pooling Support

Oracle Database Connection Pooling is provided through the support for Apache Jena `OraclePool` class. Once this class is initialized, it can return Oracle objects out of its pool of available connections. Oracle objects are essentially database connection wrappers. After `dispose` is called on the Oracle object, the connection is returned to the pool. More information about using `OraclePool` can be found in the API reference information (Javadoc).

The following example sets up an OraclePool object with five (5) initial connections.

```
public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    // test with connection properties
    java.util.Properties prop = new java.util.Properties();
    prop.setProperty("MinLimit", "2");     // the cache size is 2 at least
    prop.setProperty("MaxLimit", "10");
    prop.setProperty("InitialLimit", "2"); // create 2 connections at startup
    prop.setProperty("InactivityTimeout", "1800");    //  seconds
    prop.setProperty("AbandonedConnectionTimeout", "900");  //  seconds
    prop.setProperty("MaxStatementsLimit", "10");
    prop.setProperty("PropertyCheckInterval", "60"); // seconds

    System.out.println("Creating OraclePool");
    OraclePool op = new OraclePool(szJdbcURL, szUser, szPasswd, prop,
               "OracleSemConnPool");
    System.out.println("Done creating OraclePool");

    // grab an Oracle and do something with it
    System.out.println("Getting an Oracle from OraclePool");
    Oracle oracle = op.getOracle();
    System.out.println("Done");
    System.out.println("Is logical connection:" +
        oracle.getConnection().isLogicalConnection());
    GraphOracleSem g = new GraphOracleSem(oracle, szModelName);
    g.add(Triple.create(Node.createURI("u:John"),
                        Node.createURI("u:parentOf"),
                        Node.createURI("u:Mary")));
    g.close();
    // return the Oracle back to the pool
    oracle.dispose();

    // grab another Oracle and do something else
    System.out.println("Getting an Oracle from OraclePool");
    oracle = op.getOracle();
    System.out.println("Done");
    System.out.println("Is logical connection:" +
        oracle.getConnection().isLogicalConnection());
    g = new GraphOracleSem(oracle, szModelName);
    g.add(Triple.create(Node.createURI("u:John"),
                        Node.createURI("u:parentOf"),
                        Node.createURI("u:Jack")));
    g.close();

    OracleUtils.dropSemanticModel(oracle, szModelName);
```

```
    // return the Oracle object back to the pool
    oracle.dispose();
}
```

7.11.3 Semantic Model PL/SQL Interfaces

Several semantic PL/SQL subprograms are available through the support for Apache Jena. Table 7-2 (page 7-41) lists the subprograms and their corresponding Java class and methods.

Table 7-2 PL/SQL Subprograms and Corresponding RDF Semantic Graph support for Apache Jena Java Class and Methods

PL/SQL Subprogram	Corresponding Java Class and Methods
SEM_APIS.DROP_SEM_MODEL (page 11-39)	OracleUtils.dropSemanticModel
SEM_APIS.MERGE_MODELS (page 11-62)	OracleUtils.mergeModels
SEM_APIS.SWAP_NAMES (page 11-69)	OracleUtils.swapNames
SEM_APIS.REMOVE_DUPLICATES (page 11-65)	OracleUtils.removeDuplicates
SEM_APIS.RENAME_MODEL (page 11-67)	OracleUtils.renameModels

For information about these PL/SQL utility subprograms, see the reference information in SEM_APIS Package Subprograms (page 11-1). For information about the corresponding Java class and methods, see the RDF Semantic Graph support for Apache Jena API Reference documentation (Javadoc).

7.11.4 Inference Options

You can add options to entailment calls by using the following methods in the `Attachment` class (in package `oracle.spatial.rdf.client.jena`):

```
public void setUseLocalInference(boolean useLocalInference)
public boolean getUseLocalInference()

public void setDefGraphForLocalInference(String defaultGraphName)
public String getDefGraphForLocalInference()

public String getInferenceOption()
public void setInferenceOption(String inferenceOption)
```

Example 7-8 Specifying Inference Options

For more information about these methods, see the Javadoc.

Example 7-8 (page 7-41) enables parallel inference (with a degree of 4) and RAW format when creating an entailment. The example also uses the `performInference` method to create the entailment (comparable to using the SEM_APIS.CREATE_ENTAILMENT (page 11-15) PL/SQL procedure).

```
import java.io.*;
import com.hp.hpl.jena.query.*;
```

```
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import com.hp.hpl.jena.util.iterator.*;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.update.*;
import com.hp.hpl.jena.sparql.core.DatasetImpl;

public class TestNewInference
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];

    PrintStream psOut = System.out;

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);

    String szTBoxName = "test_new_tbox";
    {
      // First construct a TBox and load a few axioms
      ModelOracleSem modelTBox = ModelOracleSem.createOracleSemModel(oracle,
szTBoxName);
      String insertString =
        " PREFIX my:  <http://my.com/> " +
        " PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#> " +
        " INSERT DATA "                                          +
        " { my:C1  rdfs:subClassOf my:C2 .                 " +
        "   my:C2  rdfs:subClassOf my:C3 .                 " +
        "   my:C3  rdfs:subClassOf my:C4 .                 " +
        " }   ";
      UpdateAction.parseExecute(insertString,  modelTBox);
      modelTBox.close();
    }

    String szABoxName = "test_new_abox";
    {
      // Construct an ABox and load a few quads
      ModelOracleSem modelABox = ModelOracleSem.createOracleSemModel(oracle,
szABoxName);
      DatasetGraphOracleSem dataset =
DatasetGraphOracleSem.createFrom(modelABox.getGraph());
      modelABox.close();

      String insertString =
        " PREFIX my:    <http://my.com/> " +
        " PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> " +
        " INSERT DATA                                                  " +
        " { GRAPH my:G1 { my:I1  rdf:type my:C1 .                      " +
        "                 my:I2  rdf:type my:C2 .                      " +
        "               }                                              " +
        " };                                                           " +
        " INSERT DATA                                                  " +
        " { GRAPH my:G2 { my:J1  rdf:type my:C3 .                      " +
        "               }                                              " +
        " }    ";
      UpdateAction.parseExecute(insertString,  dataset);
      dataset.close();
    }
```

```
    String[] attachedModels = new String[1];
    attachedModels[0] = szTBoxName;

    String[] attachedRBs = {"OWL2RL"};

    Attachment attachment = Attachment.createInstance(
        attachedModels, attachedRBs,
        InferenceMaintenanceMode.NO_UPDATE,
        QueryOptions.ALLOW_QUERY_INVALID);

    // We are going to run named graph based local inference
    attachment.setUseLocalInference(true);

    // Set the default graph (TBox)
    attachment.setDefGraphForLocalInference(szTBoxName);

    // Set the inference option to use parallel inference
    // with a degree of 4, and RAW format.
    attachment.setInferenceOption("DOP=4,RAW8=T");

    GraphOracleSem graph = new GraphOracleSem(
        oracle,
        szABoxName,
        attachment
        );
    DatasetGraphOracleSem dsgos = DatasetGraphOracleSem.createFrom(graph);
    graph.close();

    // Invoke create_entailment PL/SQL API
    dsgos.performInference();

    psOut.println("TestNewInference: # of inferred graph " +
        Long.toString(dsgos.getInferredGraphSize()));

    String queryString =
      " SELECT ?g ?s ?p ?o WHERE {  GRAPH ?g {?s ?p ?o } } " ;

    Query query = QueryFactory.create(queryString, Syntax.syntaxARQ);
    QueryExecution qexec = QueryExecutionFactory.create(
        query, DatasetImpl.wrap(dsgos));
    ResultSet results = qexec.execSelect();

    ResultSetFormatter.out(psOut, results);

    dsgos.close();
    oracle.dispose();
  }
}
```

The output of Example 7-8 (page 7-41) is as follows.

```
TestNewInference: # of inferred graph 9

-------------------------------------------------------------------------------
------------------------------
| g                 | s                 |
p                                           | o                 |
===============================================================================
==============================
```

```
| <http://my.com/G1> | <http://my.com/I2> | <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> | <http://my.com/C3> |
| <http://my.com/G1> | <http://my.com/I2> | <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> | <http://my.com/C2> |
| <http://my.com/G1> | <http://my.com/I2> | <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> | <http://my.com/C4> |
| <http://my.com/G1> | <http://my.com/I1> | <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> | <http://my.com/C3> |
| <http://my.com/G1> | <http://my.com/I1> | <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> | <http://my.com/C1> |
| <http://my.com/G1> | <http://my.com/I1> | <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> | <http://my.com/C2> |
| <http://my.com/G1> | <http://my.com/I1> | <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> | <http://my.com/C4> |
| <http://my.com/G2> | <http://my.com/J1> | <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> | <http://my.com/C3> |
| <http://my.com/G2> | <http://my.com/J1> | <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> | <http://my.com/C4> |
--------------------------------------------------------------------------------
------------------------------
```

For information about using OWL inferencing, see Using OWL Inferencing (page 2-5).

7.11.5 PelletInfGraph Class Support Deprecated

The support for the `PelletInfGraph` class within the support for Apache Jena is deprecated. You should instead use the more optimized Oracle/Pellet integration through the PelletDb OWL 2 reasoner for Oracle Database.

7.12 Bulk Loading Using RDF Semantic Graph Support for Apache Jena

To load thousands to hundreds of thousands of RDF/OWL data files into an Oracle database, you can use the `prepareBulk` and `completeBulk` methods in the `OracleBulkUpdateHandler` Java class to simplify the task.

The `addInBulk` method in the `OracleBulkUpdateHandler` class can load triples of a graph or model into an Oracle database in bulk loading style. If the graph or model is a Jena in-memory graph or model, the operation is limited by the size of the physical memory. The `prepareBulk` method bypasses the Jena in-memory graph or model and takes a direct input stream to an RDF data file, parses the data, and load the triples into an underlying staging table. If the staging table and an accompanying table for storing long literals do not already exist, they are created automatically.

The `prepareBulk` method can be invoked multiple times to load multiple data files into the same underlying staging table. It can also be invoked concurrently, assuming the hardware system is balanced and there are multiple CPU cores and sufficient I/O capacity.

Once all the data files are processed by the `prepareBulk` method, you can invoke `completeBulk` to load all the data into the semantic network.

Example 7-9 Loading Data into the Staging Table (prepareBulk)

Example 7-9 (page 7-44) shows how to load all data files in directory `dir_1` into the underlying staging table. Long literals are supported and will be stored in a separate table. The data files can be compressed using GZIP to save storage space, and the `prepareBulk` method can detect automatically if a data file is compressed using GZIP or not.

```
Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
GraphOracleSem graph = new GraphOracleSem(oracle, szModelName);

PrintStream psOut = System.out;

String dirname = "dir_1";
File fileDir = new File(dirname);
String[] szAllFiles = fileDir.list();

// loop through all the files in a directory
for (int idx = 0; idx < szAllFiles.length; idx++) {
  String szIndFileName = dirname + File.separator + szAllFiles[idx];
  psOut.println("process to [ID = " + idx + " ] file " + szIndFileName);
  psOut.flush();

  try {
    InputStream is = new FileInputStream(szIndFileName);
    graph.getBulkUpdateHandler().prepareBulk(
        is,                    // input stream
        "http://example.com",  // base URI
        "RDF/XML",             // data file type: can be RDF/XML, N-TRIPLE, etc.
        "SEMTS",               // tablespace
        null,                  // flags
        null,                  // listener
        null                   // staging table name.
        );
    is.close();
  }
  catch (Throwable t) {
    psOut.println("Hit exception " + t.getMessage());
  }
}

graph.close();
oracle.dispose();
```

The code in Example 7-9 (page 7-44), starting from creating a new Oracle object and ending with disposing of the Oracle object, can be executed in parallel. Assume there is a quad-core CPU and enough I/O capacity on the database hardware system; you can divide up all the data files and save them into four separate directories: `dir_1`, `dir_2`, `dir_3`, and `dir_4`. Four Java threads of processes can be started to work on those directories separately and concurrently. (For more information, see Using prepareBulk in Parallel (Multithreaded) Mode (page 7-47).)

Example 7-10 Loading Data from the Staging Table into the Semantic Network (completeBulk)

After all data files are processed, you can invoke, just once, the `completeBulk` method to load the data from staging table into the semantic network, as shown in Example 7-10 (page 7-45). Triples with long literals will be loaded also.

```
graph.getBulkUpdateHandler().completeBulk(
  null,  // flags for invoking SEM_APIS.bulk_load_from_staging_table
  null   // staging table name
);
```

The `prepareBulk` method can also take a Jena model as an input data source, in which case triples in that Jena model are loaded into the underlying staging table. For more information, see the Javadoc.

Example 7-11 Using prepareBulk with RDFa

In addition to loading triples from Jena models and data files, the `prepareBulk` method supports RDFa, as shown in Example 7-11 (page 7-46). (RDFa is explained in `http://www.w3.org/TR/xhtml-rdfa-primer/`.)

```
graph.getBulkUpdateHandler().prepareBulk(
  rdfaUrl,   // url to a web page using RDFa
  "SEMTS",   // tablespace
  null,      // flags
  null,      // listener
  null       // staging table name
);
```

To parse RDFa, the relevant `java-rdfa` libraries must be included in the classpath. No additional setup or API calls are required. (For information about java-rdfa, see `http://www.rootdev.net/maven/projects/java-rdfa/` and the other topics there under Project Information.)

Note that if the `rdfaUrl` is located outside a firewall, you may need to set the following HTTP Proxy-related Java VM properties:

```
-Dhttp.proxyPort=...
-Dhttp.proxyHost=...
```

Example 7-12 Loading Quads into a DatasetGraph

The preceding examples in this section load triple data into a single graph. Loading quad data that may span across multiple named graphs (such as data in NQUADS format) requires the use of the `DatasetGraphOracleSem` class. The `DatasetGraphOracleSem` class does not use the `BulkUpdateHandler` API, but does provide a similar `prepareBulk` and `completeBulk` interface, as shown in Example 7-12 (page 7-46).

```
Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);

// Can only create DatasetGraphOracleSem from an existing GraphOracleSem
GraphOracleSem graph = new GraphOracleSem(oracle, szModelName);
DatasetGraphOracleSem dataset = DatasetGraphOracleSem.createFrom(graph);

// Don't need graph anymore, close it to free resources
graph.close();

try {
    InputStream is = new FileInputStream(szFileName);
    // load NQUADS file into a staging table. This file can be gzipp'ed.
    dataset.prepareBulk(
        is,                // input stream
        "http://my.base/", // base URI
        "N-QUADS",         // data file type; can be "TRIG"
        "SEMTS",           // tablespace
        null,              // flags
        null,              // listener
        null,              // staging table name
        false              // truncate staging table before load
    );
    // Load quads from staging table into the dataset
    dataset.completeBulk(
        null, // flags; can be "PARSE PARALLEL_CREATE_INDEX PARALLEL=4
              //               mbv_method=shadow" on a quad core machine
        null  // staging table name
    );
```

```
}
catch (Throwable t) {
    System.out.println("Hit exception " + t.getMessage());
}
finally {
    dataset.close();
    oracle.dispose();
}
```

7.12.1 Using prepareBulk in Parallel (Multithreaded) Mode

Example 7-9 (page 7-44) provided a way to load, sequentially, a set of files under a file system directory to an Oracle Database table (staging table). Example 7-13 (page 7-47) loads, concurrently, a set of files to an Oracle table (staging table). The degree of parallelism is controlled by the input parameter `iMaxThreads`.

On a balanced hardware setup with 4 or more CPU cores, setting `iMaxThreads` to 8 (or 16) can improve significantly the speed of `prepareBulk` operation when there are many data files to be processed.

Example 7-13 Using prepareBulk with iMaxThreads

```
public void testPrepareInParallel(String jdbcUrl, String user,
                      String password, String modelName,
                      String lang,
                      String tbs,
                      String dirname,
                      int iMaxThreads,
                      PrintStream psOut)
   throws SQLException, IOException,  InterruptedException
 {
   File dir = new File(dirname);
   File[] files = dir.listFiles();

   // create a set of physical Oracle connections and graph objects
   Oracle[] oracles = new Oracle[iMaxThreads];
   GraphOracleSem[] graphs = new GraphOracleSem[iMaxThreads];
   for (int idx = 0; idx < iMaxThreads; idx++) {
     oracles[idx] = new Oracle(jdbcUrl, user, password);
     graphs[idx] = new GraphOracleSem(oracles[idx], modelName);
   }

   PrepareWorker[] workers = new PrepareWorker[iMaxThreads];
   Thread[] threads = new Thread[iMaxThreads];
   for (int idx = 0; idx < iMaxThreads; idx++) {
     workers[idx] = new PrepareWorker(
         graphs[idx],
         files,
         idx,
         iMaxThreads,
         lang,
         tbs,
         psOut
         );
     threads[idx] = new Thread(workers[idx], workers[idx].getName());
     psOut.println("testPrepareInParallel: PrepareWorker " + idx + " running");
     threads[idx].start();
   }

   psOut.println("testPrepareInParallel: all threads started");
```

```
    for (int idx = 0; idx < iMaxThreads; idx++) {
      threads[idx].join();
    }
    for (int idx = 0; idx < iMaxThreads; idx++) {
      graphs[idx].close();
      oracles[idx].dispose();
    }
  }

  static class PrepareWorker implements Runnable
  {
    GraphOracleSem graph = null;
    int idx;
    int threads;
    File[] files = null;
    String lang = null;
    String tbs  = null;
    PrintStream psOut;

    public void run()
    {
      long lStartTime = System.currentTimeMillis();
      for (int idxFile = idx; idxFile < files.length; idxFile += threads) {
        File file = files[idxFile];
        try {
          FileInputStream fis = new FileInputStream(file);
          graph.getBulkUpdateHandler().prepareBulk(
              fis,
              "http://base.com/",
              lang,
              tbs,
              null,                   // flags
              new MyListener(psOut),  // listener
              null                    // table name
              );
          fis.close();
        }
        catch (Exception e) {
          psOut.println("PrepareWorker: thread ["+getName()+"] error "+
e.getMessage());
        }
        psOut.println("PrepareWorker: thread ["+getName()+"] done to "
            + idxFile + ", file = " + file.toString()
            + " in (ms) " + (System.currentTimeMillis() - lStartTime));
      }
    }

    public PrepareWorker(GraphOracleSem graph,
                         File[] files,
                         int idx,
                         int threads,
                         String lang,
                         String tbs,
                         PrintStream psOut)
    {
      this.graph   = graph;
      this.files   = files;
      this.psOut   = psOut;
      this.idx     = idx;
      this.threads = threads;
      this.files   = files;
```

```
    this.lang    = lang;
    this.tbs     = tbs ;
  }

  public String getName()
  {
    return "PrepareWorker" + idx;
  }
}

static class MyListener implements StatusListener
{
  PrintStream m_ps = null;
  public MyListener(PrintStream ps) { m_ps = ps; }
  long lLastBatch = 0;

  public void statusChanged(long count)
  {
    if (count - lLastBatch >= 10000) {
      m_ps.println("process to " + Long.toString(count));
      lLastBatch = count;
    }
  }

  public int illegalStmtEncountered(Node graphNode, Triple triple, long count)
  {
    m_ps.println("hit illegal statement with object " +
triple.getObject().toString());
    return 0; // skip it
  }
}
```

7.12.2 Handling Illegal Syntax During Data Loading

You can skip illegal triples and quads when using `prepareBulk`. This feature is useful if the source RDF data may contain syntax errors. In Example 7-14 (page 7-49), a customized implementation of the `StatusListener` interface (defined in package `oracle.spatial.rdf.client.jena`) is passed as a parameter to prepareBulk. In this example, the `illegalStmtEncountered` method prints the object field of the illegal triple, and returns 0 so that `prepareBulk` can skip that illegal triple and move on.

Example 7-14 Skipping Triples with Illegal Syntax

```
....

Oracle oracle = new Oracle(jdbcUrl, user, password);
GraphOracleSem graph = new GraphOracleSem(oracle, modelName);
PrintStream psOut = System.err;

graph.getBulkUpdateHandler().prepareBulk(
  new FileInputStream(rdfDataFilename),
  "http://base.com/",    // base
  lang,                  // data format, can be "N-TRIPLES" "RDF/XML" ...
  tbs,                   // tablespace name
  null,                  // flags
  new MyListener(psOut), // call back to show progress and also process illegal
triples/quads
  null,                  // tableName, if null use default names
  false                  // truncate existing staging tables
  );
```

```
 graph.close();
 oracle.dispose();
 ....

 // A customized StatusListener interface implementation
  public class MyListener implements StatusListener
 {
   PrintStream m_ps = null;
   public MyListener(PrintStream ps) { m_ps = ps; }

   public void statusChanged(long count)
   {
     // m_ps.println("process to " + Long.toString(count));
   }

  public int illegalStmtEncountered(Node graphNode, Triple triple, long count)
  {
    m_ps.println("hit illegal statement with object " +
triple.getObject().toString());
    return 0; // skip it
  }
 }
```

7.13 Automatic Variable Renaming

Previously, variable names used in SPARQL queries were passed directly on to Oracle Database as a part of a SQL statement. If the variable names included a SQL or PL/SQL reserved keyword, the query failed to execute. For example, the following SPARQL query used to fail because the word date as a special meaning to the Oracle Database SQL processing engine.

```
select ?date { :event  :happenedOn ?date }
```

Currently, this query does not fail, because a "smart scan" is performed and automatic replacement is done on certain reserved variable names (or variable names that are very long) before the query is sent to Oracle database for execution. The replacement is based on a list of reserved keywords that are stored in the following file embedded in `sdordfclient.jar`:

```
oracle/spatial/rdf/client/jena/oracle_sem_reserved_keywords.lst
```

This file contains over 100 entries, and you can edit the file to add entries if necessary.

The following are examples of SPARQL queries that use SQL or PL/SQL reserved keywords as variables, and that will succeed because of automatic variable renaming:

- Query using SELECT as a variable name:

  ```
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  select ?SELECT ?z
  where
  {    ?SELECT foaf:name ?y.
       optional {?SELECT foaf:knows ?z.}
  }
  ```

- Query using ARRAY and DATE as variable names:

  ```
  PREFIX x:    <http://example.com#>
  construct {
      ?ARRAY x:date ?date .
  ```

```
    }
    where {
        ?ARRAY x:happenedOn ?date .
    }
```

7.14 JavaScript Object Notation (JSON) Format Support

JavaScript Object Notation (JSON) format is supported for SPARQL query responses. JSON data format is simple, compact, and well suited for JavaScript programs.

For example, assume the following Java code snippet, which calls the `ResultSetFormatter.outputAsJSON` method:

```
Oracle oracle = new Oracle(jdbcUrl, user, password);

GraphOracleSem graph = new GraphOracleSem(oracle, modelName);
ModelOracleSem model = new ModelOracleSem(graph);

graph.add(new Triple(
                  Node.createURI("http://ds1"),
                  Node.createURI("http://dp1"),
                  Node.createURI("http://do1")
                  )
         );

graph.add(new Triple(
                  Node.createURI("http://ds2"),
                  Node.createURI("http://dp2"),
                  Node.createURI("http://do2")
                  )
         );
graph.commitTransaction();

Query q = QueryFactory.create("select ?s ?p ?o where {?s ?p ?o}",
                              Syntax.syntaxARQ);
QueryExecution qexec = QueryExecutionFactory.create(q, model);

ResultSet results = qexec.execSelect();
ResultSetFormatter.outputAsJSON(System.out, results);
```

The JSON output is as follows:

```
{
  "head": {
    "vars": [ "s" , "p" , "o" ]
  } ,
  "results": {
    "bindings": [
      {
        "s": { "type": "uri" , "value": "http://ds1" } ,
        "p": { "type": "uri" , "value": "http://dp1" } ,
        "o": { "type": "uri" , "value": "http://do1" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://ds2" } ,
        "p": { "type": "uri" , "value": "http://dp2" } ,
        "o": { "type": "uri" , "value": "http://do2" }
      }
    ]
  }
}
```

The preceding example can be changed as follows to query a remote SPARQL endpoint instead of directly against an Oracle database. (If the remote SPARQL endpoint is outside a firewall, then the HTTP Proxy probably needs to be set.)

```
Query q = QueryFactory.create("select ?s ?p ?o where {?s ?p ?o}",
                              Syntax.syntaxARQ);
QueryExecution qe = QueryExecutionFactory.sparqlService(sparqlURL, q);

ResultSet results = qexec.execSelect();
ResultSetFormatter.outputAsJSON(System.out, results);
```

To extend the first example in this section to named graphs, the following code snippet adds two quads to the same Oracle model, executes a named graph-based SPARQL query, and serializes the query output into JSON format:

```
DatasetGraphOracleSem dsgos = DatasetGraphOracleSem.createFrom(graph);
graph.close();

dsgos.add(new Quad(Node.createURI("http://g1"),
                   Node.createURI("http://s1"),
                   Node.createURI("http://p1"),
                   Node.createURI("http://o1")
                   )
         );
dsgos.add(new Quad(Node.createURI("http://g2"),
                   Node.createURI("http://s2"),
                   Node.createURI("http://p2"),
                   Node.createURI("http://o2")
                   )
         );

Query q1 = QueryFactory.create(
  "select ?g ?s ?p ?o where { GRAPH ?g {?s ?p ?o} }");

QueryExecution qexec1 = QueryExecutionFactory.create(q1,
    DatasetImpl.wrap(dsgos));

ResultSet results1 = qexec1.execSelect();
ResultSetFormatter.outputAsJSON(System.out, results1);

dsgos.close();
oracle.dispose();
```

The JSON output is as follows:

```
{
  "head": {
    "vars": [ "g" , "s" , "p" , "o" ]
  } ,
  "results": {
    "bindings": [
      {
        "g": { "type": "uri" , "value": "http://g1" } ,
        "s": { "type": "uri" , "value": "http://s1" } ,
        "p": { "type": "uri" , "value": "http://p1" } ,
        "o": { "type": "uri" , "value": "http://o1" }
      } ,
      {
        "g": { "type": "uri" , "value": "http://g2" } ,
        "s": { "type": "uri" , "value": "http://s2" } ,
        "p": { "type": "uri" , "value": "http://p2" } ,
```

```
        "o": { "type": "uri" , "value": "http://o2" }
      }
    ]
  }
}
```

You can also get a JSON response through HTTP against a Joseki-based SPARQL endpoint, as in the following example. Normally, when executing a SPARQL query against a SPARQL Web service endpoint, the `Accept request-head` field is set to be `application/sparql-results+xml`. For JSON output format, replace the `Accept request-head` field with `application/sparql-results+json`.

```
http://hostname:7001/joseki/oracle?query=<URL_ENCODED_SPARQL_QUERY>&output=json
```

7.15 Other Recommendations and Guidelines

This section contains various recommendations and other information related to SPARQL queries.

7.15.1 BOUND or !BOUND Instead of EXISTS or NOT EXISTS

For better performance, use `BOUND` or `!BOUND` instead of `EXISTS` or `NOT EXISTS`.

7.15.2 SPARQL 1.1 SELECT Expressions

You can use SPARQL 1.1 SELECT expressions without any significant performance overhead, even if the function is not currently supported within Oracle Database. Examples include the following:

```
-- Query using MD5 and SHA1 functions
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX eg:   <http://biometrics.example/ns#>
SELECT ?name (md5(?name) as ?name_in_md5) (sha1(?email) as ?sha1)
WHERE
{
  ?x foaf:name  ?name ; eg:email ?email .
}

-- Query using CONCAT function
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ( CONCAT(?G, " ", ?S) AS ?name )
WHERE
{
  ?P foaf:givenName ?G ; foaf:surname ?S
}
```

7.15.3 Syntax Involving Bnodes (Blank Nodes)

Syntax involving bnodes can be used freely in query patterns. For example, the following bnode-related syntax is supported at the parser level, so each is equivalent to its full triple-query-pattern-based version.

```
:x :q [ :p "v" ] .

(1 ?x 3 4) :p "w" .

(1 [:p :q] ( 2 ) ) .
```

7.15.4 Limit in the SERVICE Clause

When writing a SPARQL 1.1 federated query, you can set a limit on returned rows in the subquery inside the SERVICE clause. This can effectively constrain the amount of data to be transported between the local repository and the remote SPARQL endpoint.

For example, the following query specifies `limit 100` in the subquery in the SERVICE clause:

```
PREFIX : <http://example.com/>
SELECT ?s ?o
 WHERE
     {
       ?s :name "CA"
       SERVICE <http://REMOTE_SPARQL_ENDPOINT_HERE>
          {
             select ?s  ?o
                 {?s :info ?o}
              limit 100
           }
     }
```

7.15.5 OracleGraphWrapperForOntModel Class for Better Performance

The Jena `OntModel` class lets you create, modify, and analyze an ontology stored in a Jena model. However, the `OntModel` implementation is not optimized for semantic data stored in a database. This results in suboptimal performance when using `OntModel` with an Oracle model. Therefore, the class `OracleGraphWrapperForOntModel` has been created to alleviate this performance issue.

The `OracleGraphWrapperForOntModel` class implements the Jena `Graph` interface and represents a graph backed by an Oracle RDF/OWL model that is meant for use with the Jena `OntModel` API. The `OracleGraphWrapperForOntModel` class uses two semantic stores in a hybrid approach for persisting changes and responding to queries. Both semantic stores contain the same data, but one resides in memory while the other resides in the Oracle database.

When queried through `OntModel`, the `OracleGraphWrapperForOntModel` graph runs the queries against the in-memory store to improve performance. However, the `OracleGraphWrapperForOntModel` class persists changes made through `OntModel`, such as adding or removing classes, by applying changes to both stores.

Due to its hybrid approach, an `OracleGraphWrapperForOntModel` graph requires that sufficient memory be allocated to the JVM to store a copy of the ontology in memory. In internal experiments, it was found that an ontology with approximately 3 million triples requires 6 or more GB of physical memory.

Example 7-15 Using OntModel with Ontology Stored in Oracle Database

Example 7-15 (page 7-54) shows how to use the `OntModel` APIs with an existing ontology stored in an Oracle model.

```
// Set up connection to Oracle semantic store and the Oracle model
// containing the ontology
Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
GraphOracleSem oracleGraph = new GraphOracleSem(oracle, szModelName);

// Create a new hybrid graph using the oracle graph to persist
// changes.  This method will copy all the data from the oracle graph
```

```
// into an in-memory graph, which may significantly increase JVM memory
// usage.
Graph hybridGraph = OracleGraphWrapperForOntModel.getInstance(oracleGraph);

// Build a model around the hybrid graph and wrap the model with Jena's
// OntModel
Model model = ModelFactory.createModelForGraph(hybridGraph);
OntModel ontModel = ModelFactory.createOntologyModel(ontModelSpec, model);

// Perform operations on the ontology
OntClass personClass = ontModel.createClass("<http://someuri/person>");
ontModel.createIndividual(personClass);

// Close resources (will also close oracleGraph)!
hybridGraph.close();
ontModel.close();
```

Note that any `OntModel` object created using `OracleGraphWrapperForOntModel` will not reflect changes made to the underlying Oracle model by another process, through a separate `OntModel`, or through a separate Oracle graph referencing the same underlying model. All changes to an ontology should go through a single `OntModel` object and its underlying `OracleGraphWrapperForOntModel` graph until the model or graph have been closed.

Example 7-16 Using a Custom In-Memory Graph

If the default in-memory semantic store used by `OracleGraphWrapperForOntModel` is not sufficient for an ontology and system, the class provides an interface for specifying a custom graph to use as the in-memory store. Example 7-16 (page 7-55) shows how to create an `OracleGraphWrapperForOntModel` that uses a custom in-memory graph to answer queries from `OntModel`.

```
// Set up connection to Oracle semantic store and the Oracle model
// containing the ontology
Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
GraphOracleSem oracleGraph = new GraphOracleSem(oracle, szModelName);

// Create a custom in-memory graph to use instead of the default
// Jena in-memory graph for quickly answering OntModel queries.
// Note that this graph does not *need* to be in-memory, but in-memory
// is preferred.
GraphBase queryGraph = new CustomInMemoryGraphImpl();

// Create a new hybrid graph using the oracle graph to persist
// changes and the custom in-memory graph to answer queries.
// Also set the degree of parallelism to use when copying data from
// the oracle graph to the querying graph.
int degreeOfParallelism = 4;
Graph hybridGraph = OracleGraphWrapperForOntModel.getInstance(oracleGraph,
queryGraph, degreeOfParallelism);

// Build a model and wrap the model with Jena's OntModel
Model model = ModelFactory.createModelForGraph(hybridGraph);
OntModel ontModel = ModelFactory.createOntologyModel(ontModelSpec, model);

// Perform operations on the ontology
// ...

// Close resources (will close oracleGraph and queryGraph)!
```

```
hybridGraph.close();
ontModel.close();
```

7.16 Example Queries Using RDF Semantic Graph Support for Apache Jena

This section includes example queries using the support for Apache Jena. Each example is self-contained: it typically creates a model, creates triples, performs a query that may involve inference, displays the result, and drops the model.

This section includes queries that do the following:

- Count asserted triples and asserted plus inferred triples in an example "university" ontology, both by referencing the ontology by a URL and by bulk loading the ontology from a local file
- Run several SPARQL queries using a "family" ontology, including features such as LIMIT, OFFSET, TIMEOUT, DOP (degree of parallelism), ASK, DESCRIBE, CONSTRUCT, GRAPH, ALLOW_DUP (duplicate triples with multiple models), SPARUL (inserting data)
- Use the ARQ built-in function
- Use a SELECT cast query
- Instantiate Oracle Database using OracleConnection
- Use Oracle Database connection pooling

To run a query, you must do the following:

1. Include the code in a Java source file. The examples used in this section are supplied in files in the `examples` directory of the support for Apache Jena download.
2. Compile the Java source file. For example:

   ```
   > javac -classpath ../jar/'*' Test.java
   ```

   Note:

   The `javac` and `java` commands must each be on a single command line.

3. Run the compiled file. For example:

   ```
   > java -classpath ./:../jar/'*'  Test jdbc:oracle:thin:@localhost:1521:orcl
   scott <password-for-scott> M1
   ```

7.16.1 Test.java: Query Family Relationships

Example 7-17 Query Family Relationships

Example 7-17 (page 7-56) specifies that John is the father of Mary, and it selects and displays the subject and object in each `fatherOf` relationship

```
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.query.*;
```

```
public class Test {

  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    Model model = ModelOracleSem.createOracleSemModel(
      oracle, szModelName);

    model.getGraph().add(Triple.create(
          Node.createURI("http://example.com/John"),
          Node.createURI("http://example.com/fatherOf"),
          Node.createURI("http://example.com/Mary")));
    Query query = QueryFactory.create(
        "select ?f ?k WHERE {?f <http://example.com/fatherOf> ?k .}");
    QueryExecution qexec = QueryExecutionFactory.create(query, model);
    ResultSet results = qexec.execSelect();
    ResultSetFormatter.out(System.out, results, query);
    model.close();
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-17 (page 7-56), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test.java
java -classpath ./:../jar/'*'  Test jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1

-----------------------------------------------------------
| f                         | k                           |
===========================================================
| <http://example.com/John> | <http://example.com/Mary> |
-----------------------------------------------------------
```

7.16.2 Test6.java: Load OWL Ontology and Perform OWLPrime inference

Example 7-18 (page 7-57) loads an OWL ontology and performs OWLPrime inference. Note that the OWL ontology is in RDF/XML format, and after it is loaded into Oracle it will be serialized out in N-TRIPLE form. The example also queries for the number of asserted and inferred triples.

The ontology in this example can be retrieved from `http://swat.cse.lehigh.edu/onto/univ-bench.owl`, and it describes roles, resources, and relationships in a university environment.

Example 7-18 Load OWL Ontology and Perform OWLPrime inference

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import oracle.spatial.rdf.client.jena.*;
public class Test6 {
  public static void main(String[] args) throws Exception
  {
```

```
String szJdbcURL = args[0];
String szUser    = args[1];
String szPasswd  = args[2];
String szModelName = args[3];

Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
Model model = ModelOracleSem.createOracleSemModel(oracle, szModelName);

// load UNIV ontology
InputStream in = FileManager.get().open("./univ-bench.owl" );
model.read(in, null);
OutputStream os = new FileOutputStream("./univ-bench.nt");
model.write(os, "N-TRIPLE");
os.close();

String queryString =
  " SELECT ?subject ?prop ?object WHERE { ?subject ?prop ?object } ";

Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;

try {
  int iTriplesCount = 0;
  ResultSet results = qexec.execSelect() ;
  for ( ; results.hasNext() ; ) {
    QuerySolution soln = results.nextSolution() ;
    iTriplesCount++;
  }
  System.out.println("Asserted  triples count: " + iTriplesCount);
}
finally {
  qexec.close() ;
}

Attachment attachment = Attachment.createInstance(
    new String[] {}, "OWLPRIME",
    InferenceMaintenanceMode.NO_UPDATE, QueryOptions.DEFAULT);

GraphOracleSem graph = new GraphOracleSem(oracle, szModelName, attachment);
graph.analyze();
graph.performInference();

query = QueryFactory.create(queryString) ;
qexec = QueryExecutionFactory.create(query,new ModelOracleSem(graph)) ;

try {
  int iTriplesCount = 0;
  ResultSet results = qexec.execSelect() ;
  for ( ; results.hasNext() ; ) {
    QuerySolution soln = results.nextSolution() ;
    iTriplesCount++;
  }
  System.out.println("Asserted + Infered triples count: " + iTriplesCount);
}
finally {
  qexec.close() ;
}
model.close();

OracleUtils.dropSemanticModel(oracle, szModelName);
oracle.dispose();
```

```
  }
}
```

The following are the commands to compile and run Example 7-18 (page 7-57), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test6.java
java -classpath ./:../jar/'*'  Test6 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
Asserted  triples count: 293
Asserted + Infered triples count: 340
```

Note that this output reflects an older version of the LUBM ontology. The latest version of the ontology has more triples.

7.16.3 Test7.java: Bulk Load OWL Ontology and Perform OWLPrime inference

Example 7-19 (page 7-59) loads the same OWL ontology as in Test6.java: Load OWL Ontology and Perform OWLPrime inference (page 7-57), but stored in a local file using Bulk Loader. Ontologies can also be loaded using an incremental and batch loader; these two methods are also listed in the example for completeness.

Example 7-19 Bulk Load OWL Ontology and Perform OWLPrime inference

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.rdf.model.*;
import com.hp.hpl.jena.util.*;
import oracle.spatial.rdf.client.jena.*;

public class Test7
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];
    // in memory Jena Model
    Model model = ModelFactory.createDefaultModel();
    InputStream is = FileManager.get().open("./univ-bench.owl");
    model.read(is, "", "RDF/XML");
    is.close();

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem modelDest = ModelOracleSem.createOracleSemModel(oracle,
szModelName);

    GraphOracleSem g = modelDest.getGraph();
    g.dropApplicationTableIndex();

    int method = 2; // try bulk loader
    String tbs = "SYSAUX"; // can be customized
    if (method == 0) {
      System.out.println("start incremental");
      modelDest.add(model);
      System.out.println("end size " + modelDest.size());
    }
    else if (method == 1) {
      System.out.println("start batch load");
```

```
      g.getBulkUpdateHandler().addInBatch(
          GraphUtil.findAll(model.getGraph()), tbs);
      System.out.println("end size " + modelDest.size());
    }
    else {
      System.out.println("start bulk load");
      g.getBulkUpdateHandler().addInBulk(
          GraphUtil.findAll(model.getGraph()), tbs);
      System.out.println("end size " + modelDest.size());
    }
    g.rebuildApplicationTableIndex();

    long lCount = g.getCount(Triple.ANY);
    System.out.println("Asserted  triples count: " + lCount);
    model.close();
    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-19 (page 7-59), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test7.java
java -classpath ./:../jar/'*'  Test7 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
start bulk load
end size 293
Asserted  triples count: 293
```

Note that this output reflects an older version of the LUBM ontology. The latest version of the ontology has more triples.

7.16.4 Test8.java: SPARQL OPTIONAL Query

Example 7-20 (page 7-60) shows a SPARQL OPTIONAL query. It inserts triples that postulate the following:

- John is a parent of Mary.
- John is a parent of Jack.
- Mary is a parent of Jill.

It then finds parent-child relationships, optionally including any grandchild (gkid) relationships.

Example 7-20 SPARQL OPTIONAL Query

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;

public class Test8
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
```

```
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle,
szModelName);
    GraphOracleSem g = model.getGraph();

    g.add(Triple.create(
          Node.createURI("u:John"), Node.createURI("u:parentOf"),
Node.createURI("u:Mary")));
    g.add(Triple.create(
          Node.createURI("u:John"), Node.createURI("u:parentOf"),
Node.createURI("u:Jack")));
    g.add(Triple.create(
          Node.createURI("u:Mary"), Node.createURI("u:parentOf"),
Node.createURI("u:Jill")));

    String queryString =
  " SELECT ?s ?o ?gkid " +
  " WHERE { ?s <u:parentOf> ?o . OPTIONAL {?o <u:parentOf> ?gkid }} ";

    Query query = QueryFactory.create(queryString) ;
    QueryExecution qexec = QueryExecutionFactory.create(query, model) ;

    try {
      int iMatchCount = 0;
      ResultSet results = qexec.execSelect() ;
      ResultSetFormatter.out(System.out, results, query);
    }
    finally {
      qexec.close() ;
    }
    model.close();

    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-20 (page 7-60), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test8.java
java -classpath ./:../jar/'*'  Test8 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
---------------------------------
| s        | o        | gkid     |
=================================
| <u:John> | <u:Mary> | <u:Jill> |
| <u:Mary> | <u:Jill> |          |
| <u:John> | <u:Jack> |          |
---------------------------------
```

7.16.5 Test9.java: SPARQL Query with LIMIT and OFFSET

Example 7-21 (page 7-62) shows a SPARQL query with LIMIT and OFFSET. It inserts triples that postulate the following:

- John is a parent of Mary.

- John is a parent of Jack.
- Mary is a parent of Jill.

It then finds one parent-child relationship (LIMIT 1), skipping the first two parent-child relationships encountered (OFFSET 2), and optionally includes any grandchild (gkid) relationships for the one found.

Example 7-21 SPARQL Query with LIMIT and OFFSET

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;
public class Test9
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle,
szModelName);
    GraphOracleSem g = model.getGraph();

    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                    Node.createURI("u:Mary")));
    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                    Node.createURI("u:Jack")));
    g.add(Triple.create(Node.createURI("u:Mary"),
Node.createURI("u:parentOf"),
                    Node.createURI("u:Jill")));

    String queryString =
      " SELECT ?s ?o ?gkid " +
      " WHERE { ?s <u:parentOf> ?o . OPTIONAL {?o <u:parentOf> ?gkid }} " +
      " LIMIT 1 OFFSET 2";

    Query query = QueryFactory.create(queryString) ;
    QueryExecution qexec = QueryExecutionFactory.create(query, model) ;

    int iMatchCount = 0;
    ResultSet results = qexec.execSelect() ;
    ResultSetFormatter.out(System.out, results, query);
    qexec.close() ;
    model.close();

    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-21 (page 7-62), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test9.java
java -classpath ./:../jar/'*'  Test9 jdbc:oracle:thin:@localhost:1521:orcl scott
```

```
<password-for-scott> M1
-----------------------------
| s        | o        | gkid |
=============================
| <u:John> | <u:Jack> |      |
-----------------------------
```

7.16.6 Test10.java: SPARQL Query with TIMEOUT and DOP

Example 7-22 (page 7-63) shows the SPARQL query from Test9.java: SPARQL Query with LIMIT and OFFSET (page 7-61) with additional features, including a timeout setting (TIMEOUT=1, in seconds) and parallel execution setting (DOP=4).

Example 7-22 SPARQL Query with TIMEOUT and DOP

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;

public class Test10 {
  public static void main(String[] args) throws Exception  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle, szModelName);
    GraphOracleSem g = model.getGraph();

    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                        Node.createURI("u:Mary")));
    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                        Node.createURI("u:Jack")));
    g.add(Triple.create(Node.createURI("u:Mary"), Node.createURI("u:parentOf"),
                        Node.createURI("u:Jill")));
    String queryString =
        " PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#dop=4,timeout=1> "
      + " SELECT ?s ?o ?gkid WHERE { ?s <u:parentOf> ?o . "
      + " OPTIONAL {?o <u:parentOf> ?gkid }} "
      + " LIMIT 1 OFFSET 2";

    Query query = QueryFactory.create(queryString) ;
    QueryExecution qexec = QueryExecutionFactory.create(query, model) ;

    int iMatchCount = 0;
    ResultSet results = qexec.execSelect() ;
    ResultSetFormatter.out(System.out, results, query);
    qexec.close() ;
    model.close();

    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-22 (page 7-63), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test10.java
java -classpath ./:../jar/'*'  Test10 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
-----------------------------
| s        | o        | gkid |
=============================
| <u:John> | <u:Jack> |      |
-----------------------------
```

7.16.7 Test11.java: Query Involving Named Graphs

Example 7-23 (page 7-64) shows a query involving named graphs. It involves a default graph that has information about named graph URIs and their publishers. The query finds graph names, their publishers, and within each named graph finds the mailbox value using the `foaf:mbox` predicate.

Example 7-23 Named Graph Based Query

```
import java.io.*;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.sparql.core.*;
import com.hp.hpl.jena.query.*;
import oracle.spatial.rdf.client.jena.*;

public class Test11
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    GraphOracleSem graph = new GraphOracleSem(oracle, szModelName);
    DatasetGraphOracleSem dataset = DatasetGraphOracleSem.createFrom(graph);

    // don't need the GraphOracleSem anymore, release resources
    graph.close();

    // add data to the default graph
    dataset.add(new Quad(
          Quad.defaultGraphIRI, // specifies default graph
          Node.createURI("http://example.org/bob"),
          Node.createURI("http://purl.org/dc/elements/1.1/publisher"),
          Node.createLiteral("Bob Hacker")));
    dataset.add(new Quad(
          Quad.defaultGraphIRI, // specifies default graph
          Node.createURI("http://example.org/alice"),
          Node.createURI("http://purl.org/dc/elements/1.1/publisher"),
          Node.createLiteral("alice Hacker")));

    // add data to the bob named graph
    dataset.add(new Quad(
          Node.createURI("http://example.org/bob"), // graph name
          Node.createURI("urn:bob"),
          Node.createURI("http://xmlns.com/foaf/0.1/name"),
          Node.createLiteral("Bob")));
    dataset.add(new Quad(
          Node.createURI("http://example.org/bob"), // graph name
          Node.createURI("urn:bob"),
```

```
            Node.createURI("http://xmlns.com/foaf/0.1/mbox"),
            Node.createURI("mailto:bob@example")));

        // add data to the alice named graph
        dataset.add(new Quad(
            Node.createURI("http://example.org/alice"), // graph name
            Node.createURI("urn:alice"),
            Node.createURI("http://xmlns.com/foaf/0.1/name"),
            Node.createLiteral("Alice")));
        dataset.add(new Quad(
            Node.createURI("http://example.org/alice"), // graph name
            Node.createURI("urn:alice"),
            Node.createURI("http://xmlns.com/foaf/0.1/mbox"),
            Node.createURI("mailto:alice@example")));

        DataSource ds = DatasetFactory.create(dataset);

        String queryString =
              " PREFIX foaf: <http://xmlns.com/foaf/0.1/> "
            + " PREFIX dc: <http://purl.org/dc/elements/1.1/> "
            + " SELECT ?who ?graph ?mbox "
            + " FROM NAMED <http://example.org/alice> "
            + " FROM NAMED <http://example.org/bob> "
            + " WHERE "
            + " { "
            + "    ?graph dc:publisher ?who . "
            + "    GRAPH ?graph { ?x foaf:mbox ?mbox } "
            + " } ";

        Query query = QueryFactory.create(queryString);
        QueryExecution qexec = QueryExecutionFactory.create(query, ds);

        ResultSet results = qexec.execSelect();
        ResultSetFormatter.out(System.out, results, query);

        qexec.close();
        dataset.close();

        oracle.dispose();
      }
}
```

The following are the commands to compile and run Example 7-23 (page 7-64), as well as the expected output of the `java` command.

```
javac -classpath ./:./jena-2.6.4.jar:./sdordfclient.jar:./ojdbc6.jar:./slf4j-
api-1.5.8.jar:./slf4j-log4j12-1.5.8.jar:./arq-2.8.8.jar:./xercesImpl-2.7.1.jar
Test11.java
java -classpath ./:../jar/'*'  Test11 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
-------------------------------------------------------------------------
| who            | graph                      | mbox                   |
=========================================================================
| "alice Hacker" | <http://example.org/alice> | <mailto:alice@example> |
| "Bob Hacker"   | <http://example.org/bob>   | <mailto:bob@example>   |
-------------------------------------------------------------------------
```

7.16.8 Test12.java: SPARQL ASK Query

Example 7-24 (page 7-66) shows a SPARQL ASK query. It inserts a triple that postulates that John is a parent of Mary. It then finds whether John is a parent of Mary.

Example 7-24 SPARQL ASK Query

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;
public class Test12
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle,
          szModelName);
    GraphOracleSem g = model.getGraph();

    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                        Node.createURI("u:Mary")));
    String queryString = " ASK { <u:John> <u:parentOf> <u:Mary> } ";

    Query query = QueryFactory.create(queryString) ;
    QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
    boolean b = qexec.execAsk();
    System.out.println("ask result = " + ((b)?"TRUE":"FALSE"));
    qexec.close() ;

    model.close();
    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-24 (page 7-66), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test12.java
java -classpath ./:../jar/'*'  Test12 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
ask result = TRUE
```

7.16.9 Test13.java: SPARQL DESCRIBE Query

Example 7-25 (page 7-67) shows a SPARQL DESCRIBE query. It inserts triples that postulate the following:

- John is a parent of Mary.
- John is a parent of Jack.
- Amy is a parent of Jack.

It then finds all relationships that involve any parents of Jack.

Example 7-25 SPARQL DESCRIBE Query

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;

public class Test13
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle, szModelName);
    GraphOracleSem g = model.getGraph();

    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                    Node.createURI("u:Mary")));
    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
 Node.createURI("u:Jack")));
    g.add(Triple.create(Node.createURI("u:Amy"), Node.createURI("u:parentOf"),
 Node.createURI("u:Jack")));
    String queryString = " DESCRIBE ?x WHERE {?x <u:parentOf> <u:Jack>}";

    Query query = QueryFactory.create(queryString) ;
    QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
    Model m = qexec.execDescribe();
    System.out.println("describe result = " + m.toString());

    qexec.close() ;
    model.close();
    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-25 (page 7-67), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test13.java
java -classpath ./:../jar/'*'  Test13 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
describe result = <ModelCom   {u:Amy @u:parentOf u:Jack;
     u:John @u:parentOf u:Jack; u:John @u:parentOf u:Mary} |  [u:Amy, u:parentOf,
u:Jack] [u:John, u:parentOf,
       u:Jack] [u:John, u:parentOf, u:Mary]>
```

7.16.10 Test14.java: SPARQL CONSTRUCT Query

Example 7-26 (page 7-68) shows a SPARQL CONSTRUCT query. It inserts triples that postulate the following:

- John is a parent of Mary.
- John is a parent of Jack.

- Amy is a parent of Jack.
- Each parent loves all of his or her children.

It then constructs an RDF graph with information about who loves whom.

Example 7-26 SPARQL CONSTRUCT Query

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;

public class Test14
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle, szModelName);
    GraphOracleSem g = model.getGraph();

    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
 Node.createURI("u:Mary")));
    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
 Node.createURI("u:Jack")));
    g.add(Triple.create(Node.createURI("u:Amy"), Node.createURI("u:parentOf"),
 Node.createURI("u:Jack")));
    String queryString = " CONSTRUCT { ?s <u:loves> ?o } WHERE {?s <u:parentOf> ?o}";

    Query query = QueryFactory.create(queryString) ;
    QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
    Model m = qexec.execConstruct();
    System.out.println("Construct result = " + m.toString());

    qexec.close() ;
    model.close();
    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-26 (page 7-68), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test14.java
java -classpath ./:../jar/'*'  Test14 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
Construct result = <ModelCom   {u:Amy @u:loves u:Jack;
  u:John @u:loves u:Jack; u:John @u:loves u:Mary} |  [u:Amy, u:loves, u:Jack]
[u:John, u:loves,
    u:Jack] [u:John, u:loves, u:Mary]>
```

7.16.11 Test15.java: Query Multiple Models and Specify "Allow Duplicates"

Example 7-27 (page 7-69) queries multiple models and uses the "allow duplicates" option. It inserts triples that postulate the following:

- John is a parent of Jack (in Model 1).
- Mary is a parent of Jack (in Model 2).
- Each parent loves all of his or her children.

It then finds out who loves whom. It searches both models and allows for the possibility of duplicate triples in the models (although there are no duplicates in this example).

Example 7-27 Query Multiple Models and Specify "Allow Duplicates"

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;

public class Test15
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName1 = args[3];
    String szModelName2 = args[4];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model1 = ModelOracleSem.createOracleSemModel(oracle,
szModelName1);
    model1.getGraph().add(Triple.create(Node.createURI("u:John"),
                    Node.createURI("u:parentOf"), Node.createURI("u:Jack")));
    model1.close();

    ModelOracleSem model2 = ModelOracleSem.createOracleSemModel(oracle,
szModelName2);
    model2.getGraph().add(Triple.create(Node.createURI("u:Mary"),
                    Node.createURI("u:parentOf"), Node.createURI("u:Jack")));
    model2.close();

    String[] modelNamesList = {szModelName2};
    String[] rulebasesList  = {};
    Attachment attachment = Attachment.createInstance(modelNamesList, rulebasesList,
              InferenceMaintenanceMode.NO_UPDATE,
              QueryOptions.ALLOW_QUERY_VALID_AND_DUP);

    GraphOracleSem graph = new GraphOracleSem(oracle, szModelName1, attachment);
    ModelOracleSem model = new ModelOracleSem(graph);

    String queryString = " CONSTRUCT { ?s <u:loves> ?o } WHERE {?s <u:parentOf> ?o}";
    Query query = QueryFactory.create(queryString) ;
    QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
    Model m = qexec.execConstruct();
    System.out.println("Construct result = " + m.toString());
```

```
    qexec.close() ;
    model.close();
    OracleUtils.dropSemanticModel(oracle, szModelName1);
    OracleUtils.dropSemanticModel(oracle, szModelName2);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-27 (page 7-69), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test15.java
java -classpath ./:../jar/'*'  Test15 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1 M2
Construct result = <ModelCom   {u:Mary @u:loves u:Jack; u:John @u:loves u:Jack} |
[u:Mary, u:loves, u:Jack] [u:John, u:loves, u:Jack]>
```

7.16.12 Test16.java: SPARQL Update

Example 7-28 (page 7-70) inserts two triples into a model.

Example 7-28 SPARQL Update

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import com.hp.hpl.jena.util.iterator.*;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.update.*;

public class Test16
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle, szModelName);
    GraphOracleSem g = model.getGraph();
    String insertString =
      " PREFIX dc: <http://purl.org/dc/elements/1.1/> "         +
      " INSERT DATA "                                           +
      " { <http://example/book3> dc:title    \"A new book\" ; " +
      "                          dc:creator  \"A.N.Other\" . "  +
      " }   ";

    UpdateAction.parseExecute(insertString,  model);
    ExtendedIterator ei = GraphUtil.findAll(g);
    while (ei.hasNext()) {
      System.out.println("Triple " + ei.next().toString());
    }
    model.close();
    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-28 (page 7-70), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test16.java
java -classpath ./:../jar/'*'  Test16 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
Triple http://example/book3 @dc:title "A new book"
Triple http://example/book3 @dc:creator "A.N.Other"
```

7.16.13 Test17.java: SPARQL Query with ARQ Built-In Functions

Example 7-29 (page 7-71) inserts data about two books, and it displays the book titles in all uppercase characters and the length of each title string.

Example 7-29 SPARQL Query with ARQ Built-In Functions

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import com.hp.hpl.jena.util.iterator.*;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.update.*;

public class Test17 {
  public static void main(String[] args) throws Exception  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle, szModelName);
    GraphOracleSem g = model.getGraph();
    String insertString =
      " PREFIX dc: <http://purl.org/dc/elements/1.1/> "         +
      " INSERT DATA "                                           +
      " { <http://example/book3> dc:title    \"A new book\" ; " +
      "                          dc:creator  \"A.N.Other\" . "   +
      "   <http://example/book4> dc:title    \"Semantic Web Rocks\" ; " +
      "                          dc:creator  \"TB\" . "   +
      " }   ";

    UpdateAction.parseExecute(insertString,  model);
    String queryString = "PREFIX  dc:   <http://purl.org/dc/elements/1.1/> " +
      " PREFIX  fn: <http://www.w3.org/2005/xpath-functions#> " +
      " SELECT ?subject (fn:upper-case(?object) as ?object1)  " +
      "                 (fn:string-length(?object) as ?strlen) " +
      " WHERE { ?subject dc:title ?object } "
      ;
    Query query = QueryFactory.create(queryString, Syntax.syntaxARQ);
    QueryExecution qexec = QueryExecutionFactory.create(query, model);
    ResultSet results = qexec.execSelect();
    ResultSetFormatter.out(System.out, results, query);
    model.close();
    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-29 (page 7-71), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test17.java
java -classpath ./:../jar/'*'  Test17 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
-----------------------------------------------------------
| subject                | object1              | strlen |
===========================================================
| <http://example/book3> | "A NEW BOOK"         | 10     |
| <http://example/book4> | "SEMANTIC WEB ROCKS" | 18     |
-----------------------------------------------------------
```

7.16.14 Test18.java: SELECT Cast Query

Example 7-30 (page 7-72) "converts" two Fahrenheit temperatures (18.1 and 32.0) to Celsius temperatures.

Example 7-30 SELECT Cast Query

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import com.hp.hpl.jena.util.iterator.*;
import oracle.spatial.rdf.client.jena.*;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.update.*;

public class Test18 {
  public static void main(String[] args) throws Exception  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    Oracle oracle = new Oracle(szJdbcURL, szUser, szPasswd);
    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle,
szModelName);
    GraphOracleSem g = model.getGraph();
    String insertString =
      " PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> " +
      " INSERT DATA "                                     +
      " { <u:Object1> <u:temp>    \"18.1\"^^xsd:float ; " +
      "               <u:name>    \"Foo... \" . "         +
      "   <u:Object2> <u:temp>    \"32.0\"^^xsd:float ; " +
      "               <u:name>    \"Bar... \" . "         +
      " }   ";

    UpdateAction.parseExecute(insertString,  model);
    String queryString =
      " PREFIX  fn: <http://www.w3.org/2005/xpath-functions#> " +
      " SELECT ?subject ((?temp - 32.0)*5/9 as ?celsius_temp) " +
      "WHERE { ?subject <u:temp> ?temp } "
      ;
    Query query = QueryFactory.create(queryString, Syntax.syntaxARQ);
    QueryExecution qexec = QueryExecutionFactory.create(query, model);
    ResultSet results = qexec.execSelect();
    ResultSetFormatter.out(System.out, results, query);

    model.close();
    OracleUtils.dropSemanticModel(oracle, szModelName);
```

```
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-30 (page 7-72), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test18.java
java -classpath ./:../jar/'*'  Test18 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
------------------------------------------------------------------------
| subject     | celsius_temp                                           |
========================================================================
| <u:Object1> | "-7.7222223"^^<http://www.w3.org/2001/XMLSchema#float> |
| <u:Object2> | "0.0"^^<http://www.w3.org/2001/XMLSchema#float>        |
------------------------------------------------------------------------
```

7.16.15 Test19.java: Instantiate Oracle Database Using OracleConnection

Example 7-31 (page 7-73) shows a different way to instantiate an Oracle object using a given `OracleConnection` object. (In a J2EE Web application, users can normally get an `OracleConnection` object from a J2EE data source.)

Example 7-31 Instantiate Oracle Database Using OracleConnection

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import com.hp.hpl.jena.util.iterator.*;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.update.*;
import oracle.spatial.rdf.client.jena.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.*;

public class Test19 {
  public static void main(String[] args) throws Exception {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    OracleDataSource ds = new OracleDataSource();
    ds.setURL(szJdbcURL);
    ds.setUser(szUser);
    ds.setPassword(szPasswd);
    OracleConnection conn = (OracleConnection) ds.getConnection();
    Oracle oracle = new Oracle(conn);

    ModelOracleSem model = ModelOracleSem.createOracleSemModel(oracle,
szModelName);
    GraphOracleSem g = model.getGraph();

    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                        Node.createURI("u:Mary")));
    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                        Node.createURI("u:Jack")));
    g.add(Triple.create(Node.createURI("u:Mary"), Node.createURI("u:parentOf"),
         Node.createURI("u:Jill")));
    String queryString =
```

```
      " SELECT ?s ?o  WHERE { ?s <u:parentOf> ?o .} ";
    Query query = QueryFactory.create(queryString) ;
    QueryExecution qexec = QueryExecutionFactory.create(query, model) ;

    ResultSet results = qexec.execSelect() ;
    ResultSetFormatter.out(System.out, results, query);
    qexec.close() ;
    model.close();
    OracleUtils.dropSemanticModel(oracle, szModelName);
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-31 (page 7-73), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test19.java
java -classpath ./:../jar/'*'  Test19 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
----------------------
| s        | o        |
======================
| <u:John> | <u:Mary> |
| <u:John> | <u:Jack> |
| <u:Mary> | <u:Jill> |
----------------------
```

7.16.16 Test20.java: Oracle Database Connection Pooling

Example 7-32 (page 7-74) uses Oracle Database connection pooling.

Example 7-32 Oracle Database Connection Pooling

```
import java.io.*;
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.util.FileManager;
import com.hp.hpl.jena.util.iterator.*;
import com.hp.hpl.jena.graph.*;
import com.hp.hpl.jena.update.*;
import oracle.spatial.rdf.client.jena.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.*;

public class Test20
{
  public static void main(String[] args) throws Exception
  {
    String szJdbcURL = args[0];
    String szUser    = args[1];
    String szPasswd  = args[2];
    String szModelName = args[3];

    // test with connection properties (taken from some example)
    java.util.Properties prop = new java.util.Properties();
    prop.setProperty("MinLimit", "2");     // the cache size is 2 at least
    prop.setProperty("MaxLimit", "10");
    prop.setProperty("InitialLimit", "2"); // create 2 connections at startup
    prop.setProperty("InactivityTimeout", "1800");    //  seconds
    prop.setProperty("AbandonedConnectionTimeout", "900");  //  seconds
    prop.setProperty("MaxStatementsLimit", "10");
    prop.setProperty("PropertyCheckInterval", "60"); // seconds
```

```
    System.out.println("Creating OraclePool");
    OraclePool op = new OraclePool(szJdbcURL, szUser, szPasswd, prop,
               "OracleSemConnPool");
    System.out.println("Done creating OraclePool");

    // grab an Oracle and do something with it
    System.out.println("Getting an Oracle from OraclePool");
    Oracle oracle = op.getOracle();
    System.out.println("Done");
    System.out.println("Is logical connection:" +
        oracle.getConnection().isLogicalConnection());
    GraphOracleSem g = new GraphOracleSem(oracle, szModelName);
    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                        Node.createURI("u:Mary")));
    g.close();
    // return the Oracle back to the pool
    oracle.dispose();

    // grab another Oracle and do something else
    System.out.println("Getting an Oracle from OraclePool");
    oracle = op.getOracle();
    System.out.println("Done");
    System.out.println("Is logical connection:" +
        oracle.getConnection().isLogicalConnection());
    g = new GraphOracleSem(oracle, szModelName);
    g.add(Triple.create(Node.createURI("u:John"), Node.createURI("u:parentOf"),
                        Node.createURI("u:Jack")));
    g.close();

    OracleUtils.dropSemanticModel(oracle, szModelName);

    // return the Oracle back to the pool
    oracle.dispose();
  }
}
```

The following are the commands to compile and run Example 7-32 (page 7-74), as well as the expected output of the `java` command.

```
javac -classpath ../jar/'*' Test20.java
java -classpath ./:../jar/'*'  Test20 jdbc:oracle:thin:@localhost:1521:orcl scott
<password-for-scott> M1
Creating OraclePool
Done creating OraclePool
Getting an Oracle from OraclePool
Done
Is logical connection:true
Getting an Oracle from OraclePool
Done
Is logical connection:true
```

7.17 SPARQL Gateway and Semantic Data

SPARQL Gateway is a J2EE web application that is included with the support for Apache Jena. It is designed to make semantic data (RDF/OWL/SKOS) easily available to applications that operate on relational and XML data, including Oracle Business Intelligence Enterprise Edition (OBIEE) 11*g*.

This section includes the following major topics:

- SPARQL Gateway Features and Benefits Overview (page 7-76)
- Installing and Configuring SPARQL Gateway (page 7-76)
- Using SPARQL Gateway with Semantic Data (page 7-78)
- Customizing the Default XSLT File (page 7-82)
- Using the SPARQL Gateway Java API (page 7-82)
- Using the SPARQL Gateway Graphical Web Interface (page 7-85)
- Using SPARQL Gateway as an XML Data Source to OBIEE (page 7-91)

7.17.1 SPARQL Gateway Features and Benefits Overview

SPARQL Gateway handles several challenges in exposing semantic data to a non-semantic application:

- RDF syntax, SPARQL query syntax and SPARQL protocol must be understood.
- The SPARQL query response syntax must be understood.
- A transformation must convert a SPARQL query response to something that the application can consume.

To address these challenges, SPARQL Gateway manages SPARQL queries and XSLT operations, executes SPARQL queries against any arbitrary standard-compliant SPARQL endpoints, and performs necessary XSL transformations before passing the response back to applications. Applications can then consume semantic data as if it is coming from an existing data source.

Different triple stores or quad stores often have different capabilities. For example, the SPARQL endpoint supported by Oracle Database, with RDF Semantic Graph support for Apache Jena and with Joseki, allows parallel execution, query timeout, dynamic sampling, result cache, and other features, in addition to the core function of parsing and answering a given standard-compliant SPARQL query. However, these features may not be available from another given semantic data store.

With the RDF Semantic Graph SPARQL Gateway, you get certain highly desirable capabilities, such as the ability to set a timeout on a long running query and the ability to get partial results from a complex query in a given amount of time. Waiting indefinitely for a query to finish is a challenge for end users, as is an application with a response time constraint. SPARQL Gateway provides both timeout and best effort query functions on top of a SPARQL endpoint. This effectively removes some uncertainty from consuming semantic data through SPARQL query executions. (See Specifying a Timeout Value (page 7-81) and Specifying Best Effort Query Execution (page 7-81).)

7.17.2 Installing and Configuring SPARQL Gateway

To install and configure SPARQL Gateway, follow these major steps, which are explained in their own topics:

1. Download the RDF Semantic Graph Support for Apache Jena .zip File (if Not Already Done) (page 7-77)
2. Deploy SPARQL Gateway in WebLogic Server (page 7-77)
3. Modify Proxy Settings_ if Necessary (page 7-77)

4. Configure the OracleSGDS Data Source_ if Necessary (page 7-78)
5. Add and Configure the SparqlGatewayAdminGroup Group_ if Desired (page 7-78)

7.17.2.1 Download the RDF Semantic Graph Support for Apache Jena .zip File (if Not Already Done)

If you have not already done so, download the RDF Semantic Graph support for Apache Jena file from the RDF Semantic Graph page and unzip it into a temporary directory, as explained in Setting Up the Software Environment (page 7-2).

Note that the SPARQL Gateway Java class implementations are embedded in `sdordfclient.jar` (see Using the SPARQL Gateway Java API (page 7-82)).

7.17.2.2 Deploy SPARQL Gateway in WebLogic Server

Deploy SPARQL Gateway in Oracle WebLogic Server, as follows:

1. Go to the autodeploy directory of WebLogic Server, and copy over the prebuilt `sparqlgateway.war` file as follows. (For information about auto-deploying applications in development domains, see: `http://docs.oracle.com/cd/E11035_01/wls100/deployment/autodeploy.html`)

   ```
   cp -rf  /tmp/jena_adapter/sparqlgateway_web_app/sparqlgateway.war  <domain_name>/
   autodeploy/sparqgateway.war
   ```

   In this example, <*domain_name*> is the name of a WebLogic Server domain.

   You can customize the prebuilt application in the following ways:

   - Modify the `WEB-INF/web.xml` file embedded in `sparqlgateway_web_app/sparqlgateway.war` as needed. Be sure to specify appropriate values for the `sparql_gateway_repository_filedir` and `sparql_gateway_repository_url` parameters.
   - Add XSLT files or SPARQL query files to the top-level directory of `sparqlgateway_web_app/sparqlgateway.war`, if necessary.

     The following files are provided by Oracle in that directory: `default.xslt`, `noop.xslt`, and `qb1.sparql`. The `default.xslt` file is intended mainly for transforming SPARQL query responses (XML) to a format acceptable to Oracle.

     (These files are described in Storing SPARQL Queries and XSL Transformations (page 7-79); using SPARQL Gateway with OBIEE is explained in Using SPARQL Gateway as an XML Data Source to OBIEE (page 7-91).)

2. Verify your deployment by using your Web browser to connect to a URL in the following format (assume that the Web application is deployed at port 7001):

   ```
   http://<hostname>:7001/sparqlgateway
   ```

7.17.2.3 Modify Proxy Settings, if Necessary

If your SPARQL Gateway is behind a firewall and you want SPARQL Gateway to communicate with SPARQL endpoints on the Internet as well as those inside the firewall, you probably need to use the following JVM settings:

```
-Dhttp.proxyHost=<your_proxy_host>
-Dhttp.proxyPort=<your_proxy_port>
-Dhttp.nonProxyHosts=127.0.0.1|<hostname_1_for_sparql_endpoint_inside_firewall>|
<hostname_2_for_sparql_endpoint_inside_firewall>|...|
<hostname_n_for_sparql_endpoint_inside_firewall>
```

You can specify these settings in the `startWebLogic.sh` script.

7.17.2.4 Configure the OracleSGDS Data Source, if Necessary

If an Oracle database is used for storage of and access to SPARQL queries and XSL transformations for SPARQL Gateway, then you must configure a data source named `OracleSGDS`.

To create this data source, follow the instructions in Creating the Required Data Source Using WebLogic Server (page 7-6); however, specify **`OracleSGDS`** as the data source name instead of `OracleSemDS`.

If the `OracleSGDS` data source is configured and available, SPARQL Gateway servlet will automatically create all the necessary tables and indexes upon initialization.

7.17.2.5 Add and Configure the SparqlGatewayAdminGroup Group, if Desired

The following JSP files in SPARQL Gateway can help you to view, edit, and update SPARQL queries and XSL transformations that are stored in an Oracle database:

```
http://<host>:7001/sparqlgateway/admin/sparql.jsp
http://<host>:7001/sparqlgateway/admin/xslt.jsp
```

These files are protected by HTTP Basic Authentication. In `WEB-INF/weblogic.xml`, a principal named `SparqlGatewayAdminGroup` is defined.

To be able to log in to either of these JSP pages, you must use the WebLogic Server to add a group named `SparqlGatewayAdminGroup`, and create a new user or assign an existing user to this group.

7.17.3 Using SPARQL Gateway with Semantic Data

The primary interface for an application to interact with SPARQL Gateway is through a URL with the following format:

```
http://host:port/sparqlgateway/sg?<SPARQL_ENDPOINT>&<SPARQL_QUERY>&<XSLT>
```

In the preceding format:

- *<SPARQL_ENDPOINT>* specifies the `ee` parameter, which contains a URL encoded form of a SPARQL endpoint.

  For example, `ee=http%3A%2F%2Fsparql.org%2Fbooks` is the URL encoded string for SPARQL endpoint `http://sparql.org/books`. It means that SPARQL queries are to be executed against endpoint `http://sparql.org/books`.

- *<SPARQL_QUERY>* specifies either the SPARQL query, or the location of the SPARQL query.

  If it is feasible for an application to accept a very long URL, you can encode the whole SPARQL query and set `eq=<encoded_SPARQL_query>` in the URL If it is not feasible for an application to accept a very long URL, you can store the SPARQL queries and make them available to SPARQL Gateway using one of the

approaches described in Storing SPARQL Queries and XSL Transformations (page 7-79).

- <*XSLT*> specifies either the XSL transformation, or the location of the XSL transformation.

  If it is feasible for an application to accept a very long URL, you can encode the whole XSL transformation and set `ex=<encoded_XSLT>` in the URL If it is not feasible for an application to accept a very long URL, you can store the XSL transformations and make them available to SPARQL Gateway using one of the approaches described in Storing SPARQL Queries and XSL Transformations (page 7-79).

Related topics:

- Storing SPARQL Queries and XSL Transformations (page 7-79)
- Specifying a Timeout Value (page 7-81)
- Specifying Best Effort Query Execution (page 7-81)
- Specifying a Content Type Other Than text/xml (page 7-82)

7.17.3.1 Storing SPARQL Queries and XSL Transformations

If it is not feasible for an application to accept a very long URL, you can specify the location of the SPARQL query and the XSL transformation in the <*SPARQL_QUERY*> and <*XSLT*> portions of the URL format described in Using SPARQL Gateway with Semantic Data (page 7-78), using any of the following approaches:

- Store the SPARQL queries and XSL transformations in the SPARQL Gateway Web application itself.

  To do this, unpack the `sparqlgateway.war` file, and store the SPARQL queries and XSL transformations in the top-level directory; then pack the `sparqlgateway.war` file and redeploy it.

  The `sparqlgateway.war` file includes the following example files: `qb1.sparql` (SPARQL query) and `default.xslt` (XSL transformation).

  Tip:

  Use the file extension `.sparql` for SPARQL query files, and the file extension `.xslt` for XSL transformation files.

  The syntax for specifying these files (using the provided example file names) is **`wq`**`=qb1`.sparql for a SPARQL query file and **`wx`**`=default.xslt` for an XSL transformation file.

  If you want to customize the default XSL transformations, see the examples in Customizing the Default XSLT File (page 7-82).

  If you specify `wx=noop.xslt`, XSL transformation is not performed and the SPARQL response is returned "as is" to the client.

- Store the SPARQL queries and XSL transformations in a file system directory, and make sure that the directory is accessible for the deployed SPARQL Gateway Web application.

  By default, the directory is set to `/tmp`, as shown in the following `<init-param>` setting:

```
<init-param>
   <param-name>sparql_gateway_repository_filedir</param-name>
   <param-value>/tmp/</param-value>
</init-param>
```

It is recommended that you customize this directory before deploying the SPARQL Gateway. To change the directory setting, edit the text in between the `<param-value>` and `</param-value>` tags.

The following example specifies a SPARQL query file and an XSL transformation file that are in the directory specified in the `<init-param>` element for `sparql_gateway_repository_filedir`:

```
fq=qb1.sparql
fx=myxslt1.xslt
```

- Make the SPARQL queries and XSL transformations accessible from a website.

  By default, the website directory is set to `http://127.0.0.1/queries/`, as shown in the following `<init-param>` setting:

  ```
  <init-param>
     <param-name>sparql_gateway_repository_url</param-name>
     <param-value>http://127.0.0.1/queries/</param-value>
  </init-param>
  ```

  Customize this directory before deploying the SPARQL Gateway. To change the website setting, edit the text in between the `<param-value>` and `</param-value>` tags.

  The following example specifies a SPARQL query file and an XSL transformation file that are in the URL specified in the `<init-param>` element for `sparql_gateway_repository_url`.

  ```
  uq=qb1.sparql
  ux=myxslt1.xslt
  ```

  Internally, SPARQL Gateway computes the appropriate complete URL, fetches the content, starts query execution, and applies the XSL transformation to the query response XML.

- Store the SPARQL queries and XSL transformations in an Oracle database.

  This approach requires that the J2EE data source `OracleSGDS` be defined. After SPARQL Gateway retrieves a database connection from the OracleSGDS data source, a SPARQL query is read from the database table ORACLE_ORARDF_SG_QUERY using the integer ID provided.

  The syntax for fetching a SPARQL query from an Oracle database is **`dq=`**`<integer-id>`, and the syntax for fetching an XSL transformation from an Oracle database is **`dx=`**`<integer-id>`.

  Upon servlet initialization, the following tables are created automatically if they do not already exist (you do not need to create them manually):

  – ORACLE_ORARDF_SG_QUERY with a primary key of QID (integer type)

  – ORACLE_ORARDF_SG_XSLT with a primary key of XID (integer type)

7.17.3.2 Specifying a Timeout Value

When you submit a potentially long-running query using the URL format described in Using SPARQL Gateway with Semantic Data (page 7-78), you can limit the execution time by specifying a timeout value in milliseconds. For example, the following shows the URL format and a timeout specification that the SPARQL query execution started from SPARQL Gateway is to be ended after 1000 milliseconds (1 second):

```
http://host:port/sparqlgateway/sg?<SPARQL_ENDPOINT>&<SPARQL_QUERY>&<XSLT>&t=1000
```

If a query does not finish when timeout occurs, then an empty SPARQL response is constructed by SPARQL Gateway.

Note that even if SPARQL Gateway times out a query execution at the HTTP connection level, the query may still be running on the server side. The actual behavior will be vendor-dependent.

7.17.3.3 Specifying Best Effort Query Execution

Note:

You can specify best effort query execution only if you also specify a timeout value (described in Specifying a Timeout Value (page 7-81)).

When you submit a potentially long-running query using the URL format described in Using SPARQL Gateway with Semantic Data (page 7-78), if you specify a timeout value, you can also specify a "best effort" limitation on the query. For example, the following shows the URL format with a timeout specification of 1000 milliseconds (1 second) and a best effort specification (`&b=t`):

```
http://host:port/sparqlgateway/sg?<SPARQL_ENDPOINT>&<SPARQL_QUERY>&<XSLT>&t=1000&b=t
```

The web.xml file includes two parameter settings that affect the behavior of the best effort option: `sparql_gateway_besteffort_maxrounds` and `sparql_gateway_besteffort_maxthreads`. The following show the default definitions:

```
<init-param>
  <param-name>sparql_gateway_besteffort_maxrounds</param-name>
  <param-value>10</param-value>
</init-param>

<init-param>
  <param-name>sparql_gateway_besteffort_maxthreads</param-name>
  <param-value>3</param-value>
</init-param>
```

When a SPARQL SELECT query is executed in best effort style, a series of queries will be executed with an increasing LIMIT value setting in the SPARQL query body. (The core idea is based on the observation that a SPARQL query runs faster with a smaller LIMIT setting.) SPARQL Gateway starts query execution with a "LIMIT 1" setting. Ideally, this query can finish before the timeout is due. Assume that is the case, the next query will have its LIMIT setting is increased, and subsequent queries have higher limits. The maximum number of query executions is controlled by the `sparql_gateway_besteffort_maxrounds` parameter.

If it is possible to run the series of queries in parallel, the `sparql_gateway_besteffort_maxthreads` parameter controls the degree of parallelism.

7.17.3.4 Specifying a Content Type Other Than text/xml

By default, SPARQL Gateway assumes that XSL transformations generate XML, and so the default content type set for HTTP response is `text/xml`. However, if your application requires a response format other than XML, you can specify the format in an additional URL parameter (with syntax `&rt=`), using the following format:

```
http://host:port/sparqlgateway/sg?
<SPARQL_ENDPOINT>&<SPARQL_QUERY>&<XSLT>&rt=<content_type>
```

Note that <*content_type*> must be URL encoded.

7.17.4 Customizing the Default XSLT File

You can customize the default XSL transformation file (the one referenced using `wx=default.xslt`). This section presents some examples of customizations.

The following example implements this namespace prefix replacement logic: if a variable binding returns a URI that starts with `http://purl.org/goodrelations/v1#`, that portion is replaced by `gr:`; and if a variable binding returns a URI that starts with `http://www.w3.org/2000/01/rdf-schema#`, that portion is replaced by `rdfs:`.

```
<xsl:when test="starts-with(text(),'http://purl.org/goodrelations/v1#')">
   <xsl:value-of select="concat('gr:',substring-after(text(),'http://purl.org/
goodrelations/v1#'))"/>
</xsl:when>
...
<xsl:when test="starts-with(text(),'http://www.w3.org/2000/01/rdf-schema#')">
   <xsl:value-of select="concat('rdfs:',substring-after(text(),'http://www.w3.org/
2000/01/rdf-schema#'))"/>
</xsl:when>
```

The following example implements logic to trim a leading `http://localhost/` or a leading `http://127.0.0.1/`.

```
<xsl:when test="starts-with(text(),'http://localhost/')">
  <xsl:value-of select="substring-after(text(),'http://localhost/')"/>
</xsl:when>
<xsl:when test="starts-with(text(),'http://127.0.0.1/')">
  <xsl:value-of select="substring-after(text(),'http://127.0.0.1/')"/>
</xsl:when>
```

7.17.5 Using the SPARQL Gateway Java API

In addition to a Web interface, the SPARQL Gateway administration service provides a convenient Java application programming interface (API) for managing SPARQL queries and their associated XSL transformations. The Java API is included in the RDF Semantic Graph support for Apache Jena library, `sdordfclient.jar`.

Java API reference information is available in the `javadoc_sparqlgateway.zip` file that is included in the SPARQL Gateway .zip file (described in Download the RDF Semantic Graph Support for Apache Jena .zip File (if Not Already Done) (page 7-77)).

The main entry point for this API is the `oracle.spatial.rdf.client.jena.SGDBHandler` class (SPARQL Gateway

Database Handler), which provides the following static methods for managing queries and transformations:

- `deleteSparqlQuery(Connection, int)`
- `deleteXslt(Connection, int)`
- `insertSparqlQuery(Connection, int, String, String, boolean)`
- `insertXslt(Connection, int, String, String, boolean)`
- `getSparqlQuery(Connection, int, StringBuilder, StringBuilder)`
- `getXslt(Connection, int, StringBuilder, StringBuilder)`

These methods manipulate and retrieve entries in the SPARQL Gateway associated tables that are stored in an Oracle Database instance. To use these methods, the necessary associated tables must already exist. If the tables do not exist, deploy the SPARQL Gateway on a Web server and access a URL in the following format:

`http://<host>:<port>/sparqlgateway/sg?`

where <*host*> is the host name of the Web server and <*port*> is the listening port of the Web server. Accessing this URL will automatically create the necessary tables if they do not already exist.

Any changes made through the Java API affect the SPARQL Gateway Web service in the same way as changes made through the administration Web interface. This provides the flexibility to manage queries and transformations using the interface you find most convenient.

Note that the insert methods provided by the Java API will not replace existing queries or transformations stored in the tables. Attempting to replace an existing query or transformation will fail. To replace a query or transformation, you must remove the existing entry in the table using one of the delete methods, and then insert the new query or transformation using one of the insert methods.

The following examples demonstrate how to perform common management tasks using the Java API. The examples assume a connection has already been established to the underlying Oracle Database instance backing the SPARQL Gateway.

Example 7-33 Storing a SPARQL Query and an XSL Transformation

Example 7-33 (page 7-83) adds a query and an XSL transformation to the database backing the SPARQL Gateway. After the query and transformation are added, other programs can use the query and transformation through the gateway by specifying the appropriate query ID (`qid`) and XSL transformation ID (`xid`) in the request URL.

Note that Although Example 7-33 (page 7-83) inserts both a query and transformation, the query and transformation are not necessarily related and do not need to be used together when accessing SPARQL Gateway. Any query in the database can be used with any transformation in the database when submitting a request to SPARQL Gateway.

```
String query = "PREFIX ... SELECT ..."; // full SPARQL query text
String xslt  = "<?xml ...> ...";        // full XSLT transformation text

String queryDesc = "Conference attendee information"; // description of SPARQL query
String xsltDesc = "BIEE table widget transformation"; // description of XSLT
transformation
```

```
int queryId = queryIdCounter++; // assign a unique ID to this query
int xsltId  = xsltIdCounter++;  // assign a unique ID to this transformation

// Inserting a query or transformation will fail if the table already contains
// an entry with the same ID.  Setting this boolean to true will ignore these
// exceptions (but the table will remain unchanged). Here we specify that we
// want an exception thrown if we encounter a duplicate ID.
boolean ignoreDupException = false;

// add the query
try {
  // Delete query if one already exists with this ID (this will not throw an
  // error if no such entry exists)
  SGDBHandler.deleteSparqlQuery( connection, queryId );
  SGDBHandler.insertSparqlQuery( connection, queryId, query, queryDesc,
ignoreDupException );
} catch( SQLException sqle ) {
  // Handle exception
} catch( QueryException qe ) {
  // Handle query syntax exception
}

// add the XSLT
try {
  // Delete xslt if one already exists with this ID (this will not throw an
  // error if no such entry exists)
  SGDBHandler.deleteXslt( connection, xsltId );
  SGDBHandler.insertXslt( connection, xsltId, xslt, xsltDesc, ignoreDupException );
}  catch( SQLException sqle ) {
  // Handle database exception
} catch( TransformerConfigurationException tce ) {
  // Handle XSLT syntax exception
}
```

Example 7-34 Modifying a Query

Example 7-34 (page 7-84) retrieves an existing query from the database, modifies it, then stores the updated version of the query back in the database. These steps simulate editing a query and saving the changes. (Note that if the query does not exist, an exception is thrown.)

```
StringBuilder query;
StringBuilder description;

// Populate these with the query text and description from the database
query = new StringBuilder( );
description = new StringBuilder( );

// Get the query from the database
try {
  SGDBHandler.getSparqlQuery( connection, queryId, query, description );
} catch( SQLException sqle ) {
  // Handle exception
  // NOTE: exception is thrown if query with specified ID does not exist
}

// The query and description should be populated now

// Modify the query
String updatedQuery = query.toString( ).replaceAll("invite", "attendee");
```

```
// Insert the query back into the database
boolean ignoreDup = false;
try {
  // First must delete the old query
  SGDBHandler.deleteSparqlQuery( connection, queryId );
  // Now we can add
  SGDBHandler.insertSparqlQuery( connection, queryId, updatedQuery,
description.toString( ), ignoreDup );
} catch( SQLException sqle ) {
  // Handle exception
} catch( QueryException qe ) {
  // Handle query syntax exception
}
```

Example 7-35 Retrieving and Printing an XSL Transformation

Example 7-35 (page 7-85) retrieves an existing XSL transformation and prints it to standard output. (Note that if the transformation does not exist, an exception is thrown.)

```
StringBuilder xslt;
StringBuilder description;

// Populate these with the XSLT text and description from the database
xslt = new StringBuilder( );
description = new StringBuilder( );

try {
  SGDBHandler.getXslt( connection, xsltId, xslt, description );
} catch( SQLException sqle ) {
  // Handle exception
  // NOTE: exception is thrown if transformation with specified ID does not exist
}

// Print it to standard output
System.out.printf( "XSLT description: %s\n", description.toString( ) );
System.out.printf( "XSLT body:\n%s\n", xslt.toString( ) );
```

7.17.6 Using the SPARQL Gateway Graphical Web Interface

SPARQL Gateway provides several browser-based interfaces to help you test queries, navigate semantic data, and manage SPQARQL query and XSLT files:

- Main Page (index.html) (page 7-85)
- Navigation and Browsing Page (browse.jsp) (page 7-87)
- XSLT Management Page (xslt.jsp) (page 7-89)
- SPARQL Management Page (sparql.jsp) (page 7-90)

7.17.6.1 Main Page (index.html)

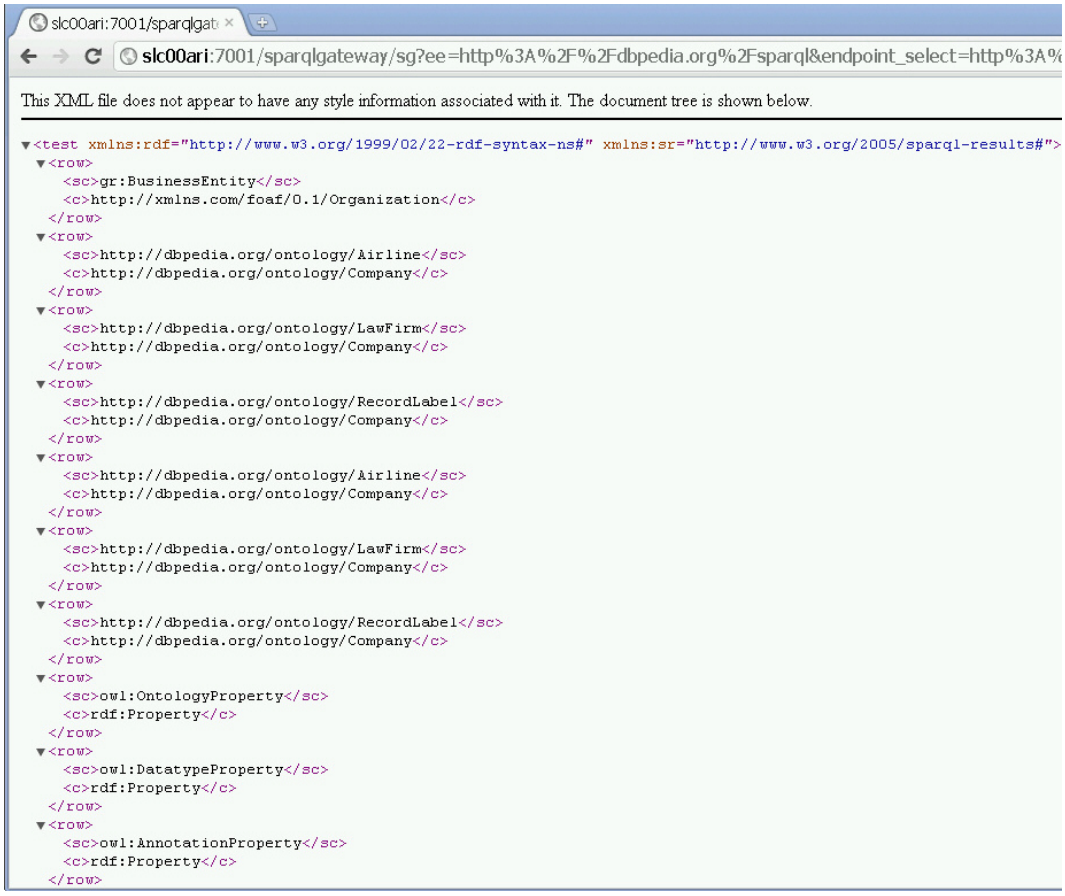
`http://<host>:<port>/sparqlgateway/index.html` provides a simple interface for executing SPARQL queries and then applying the transformations in the default.xslt file to the response. Figure 7-2 (page 7-86) shows this interface for executing a query.

Figure 7-2 Graphical Interface Main Page (index.html)

Enter or select a **SPARQL Endpoint**, specify the **SPARQL SELECT Query Body,** and press **Submit Query**.

For example, if you specify `http://dbpedia.org/sparql` as the SPARQL endpoint and use the SPARQL query body from Figure 7-2 (page 7-86), the response will be similar to Figure 7-3 (page 7-87). Note that the default transformations (in `default.xslt`) have been applied to the XML output in this figure.

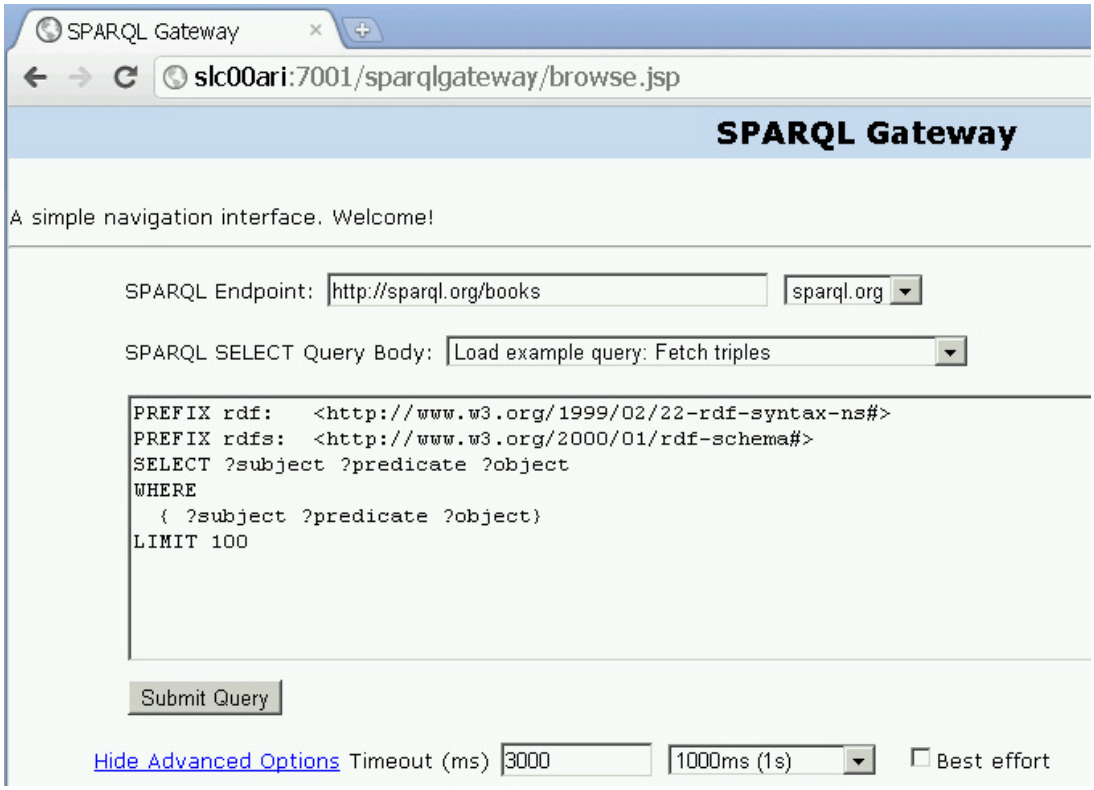
Figure 7-3 SPARQL Query Main Page Response



7.17.6.2 Navigation and Browsing Page (browse.jsp)

`http://<host>:<port>/sparqlgateway/browse.jsp` provides navigation and browsing capabilities for semantic data. It works against any standard compliant SPARQL endpoint. Figure 7-4 (page 7-88) shows this interface for executing a query.

Figure 7-4 Graphical Interface Navigation and Browsing Page (browse.jsp)



Enter or select a **SPARQL Endpoint**, specify the **SPARQL SELECT Query Body,** optionally specify a **Timeout (ms)** value in milliseconds and the **Best Effort** option, and press **Submit Query**.

The SPARQL response is parsed and then presented in table form, as shown in Figure 7-5 (page 7-88).

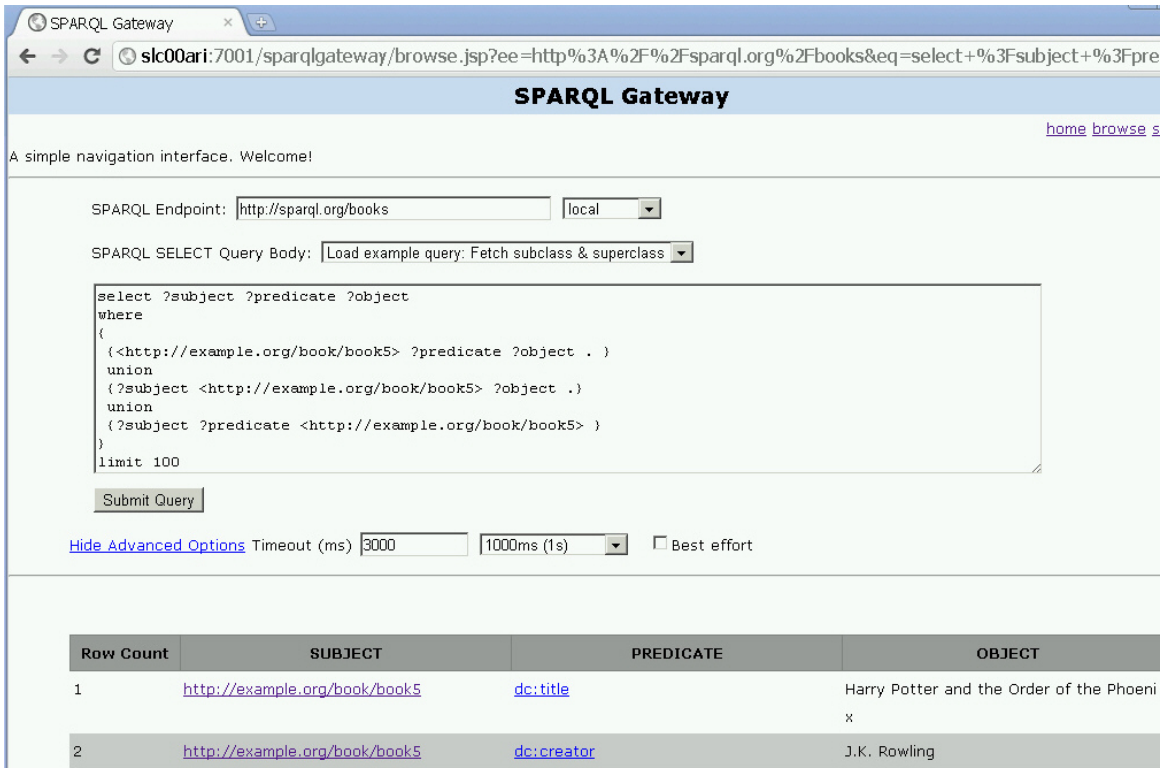
Figure 7-5 Browsing and Navigation Page: Response

Row Count	SUBJECT	PREDICATE	OBJECT
1	http://example.org/book/book5	dc:title	Harry Potter and the Order of the Phoeni x
2	http://example.org/book/book5	http://purl.org/dc/elements/1.1/title	J.K. Rowling
3	_:b0	http://www.w3.org/2001/vcard-rdf/3.0#FN	J.K. Rowling
4	_:b0	http://www.w3.org/2001/vcard-rdf/3.0#N	_:b1

In Figure 7-5 (page 7-88), note that URIs are clickable to allow navigation, and that when users move the cursor over a URI, tool tips are shown for the URIs which have been shortened for readability (as in `http://purl.org.dc/elements/1.1/title` being displayed as the tool tip for `dc:title` in the figure).

If you click the URI `http://example.org/book/book5` in the output shown in Figure 7-5 (page 7-88), a new SPARQL query is automatically generated and executed. This generated SPARQL query has three query patterns that use this particular URI as subject, predicate, and object, as shown in Figure 7-6 (page 7-89). Such a query can give you a good idea about how this URI is used and how it is related to other resources in the data set.

Figure 7-6 Query and Response from Clicking URI Link



When there are many matches of a query, the results are organized in pages and you can click on any page. The page size by default is 50 results. To display more (or fewer) than 50 rows per page in a response with the Browsing and Navigation Page (`browse.jsp`), you can specify the `&resultsPerPage` parameter in the URL. For example, to allow 100 rows per page, include the following in the URL:

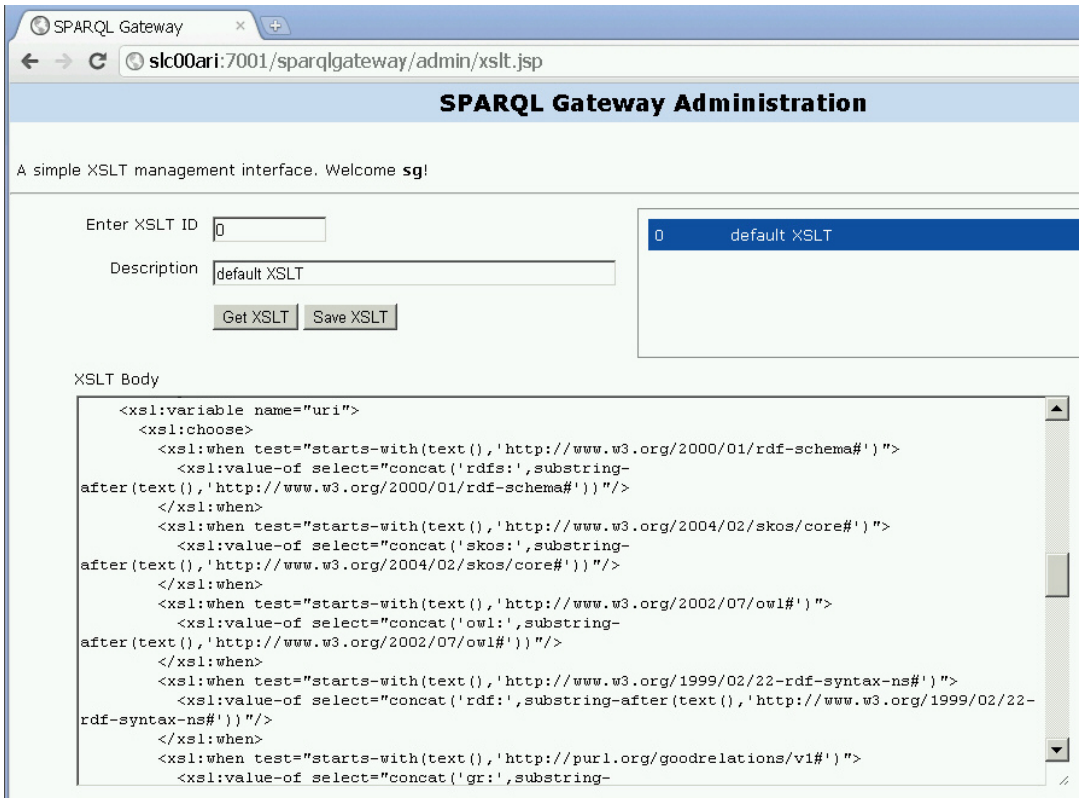
```
&resultsPerPage=100
```

7.17.6.3 XSLT Management Page (xslt.jsp)

`http://<host>:<port>/sparqlgateway/admin/xslt.jsp` provides a simple XSLT management interface. You can enter an XSLT ID (integer) and click **Get XSLT** to retrieve both the Description and XSLT Body. You can modify the XSLT Body text and then save the changes by clicking **Save XSLT**. Note that there is a previewer to help you navigate among available XSLT definitions.

Figure 7-7 (page 7-90) shows the XSLT Management Page.

Figure 7-7 XSLT Management Page

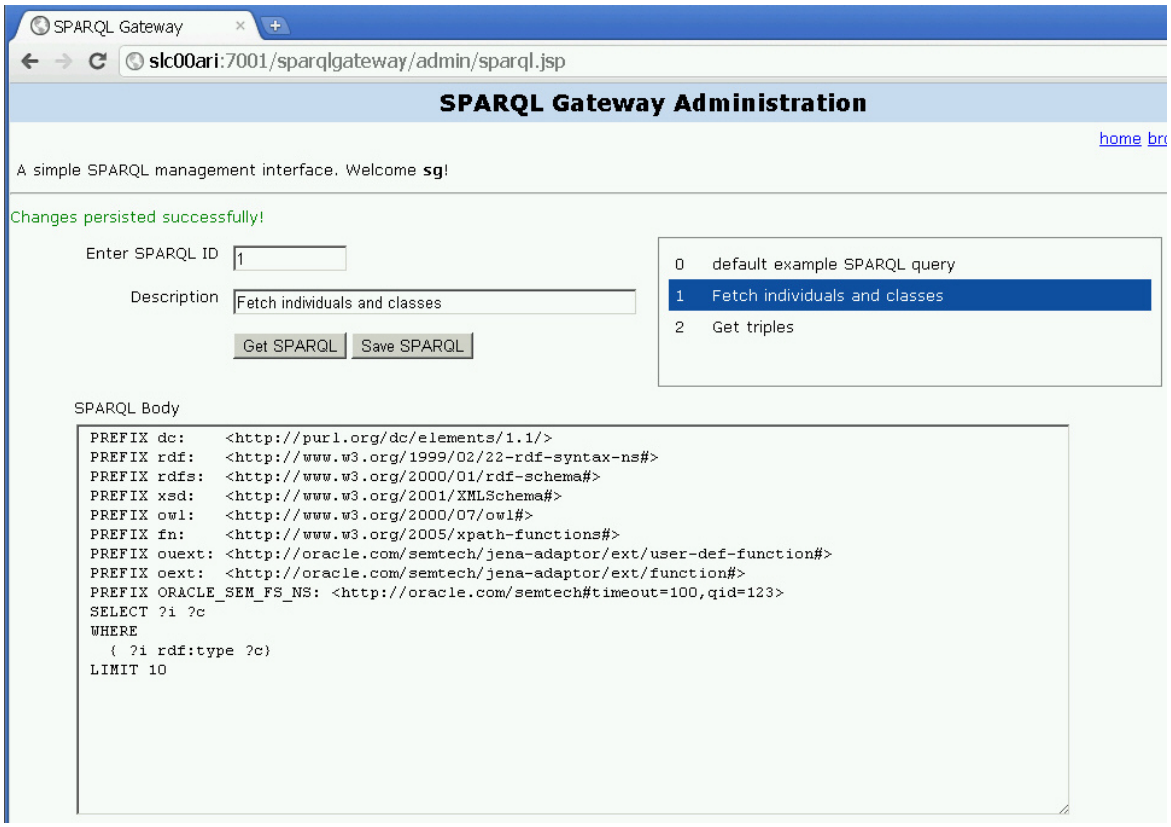


7.17.6.4 SPARQL Management Page (sparql.jsp)

`http://<host>:<port>/sparqlgateway/admin/xslt.jsp` provides a simple SPARQL management interface. You can enter a SPARQL ID (integer) and click **Get SPARQL** to retrieve both the Description and SPARQL Body. You can modify the SPARQL Body text and then save the changes by clicking **Save SPARQL**. Note that there is a previewer to help you navigate among available SPARQL queries.

Figure 7-8 (page 7-91) shows the SPARQL Management Page.

Figure 7-8 SPARQL Management Page

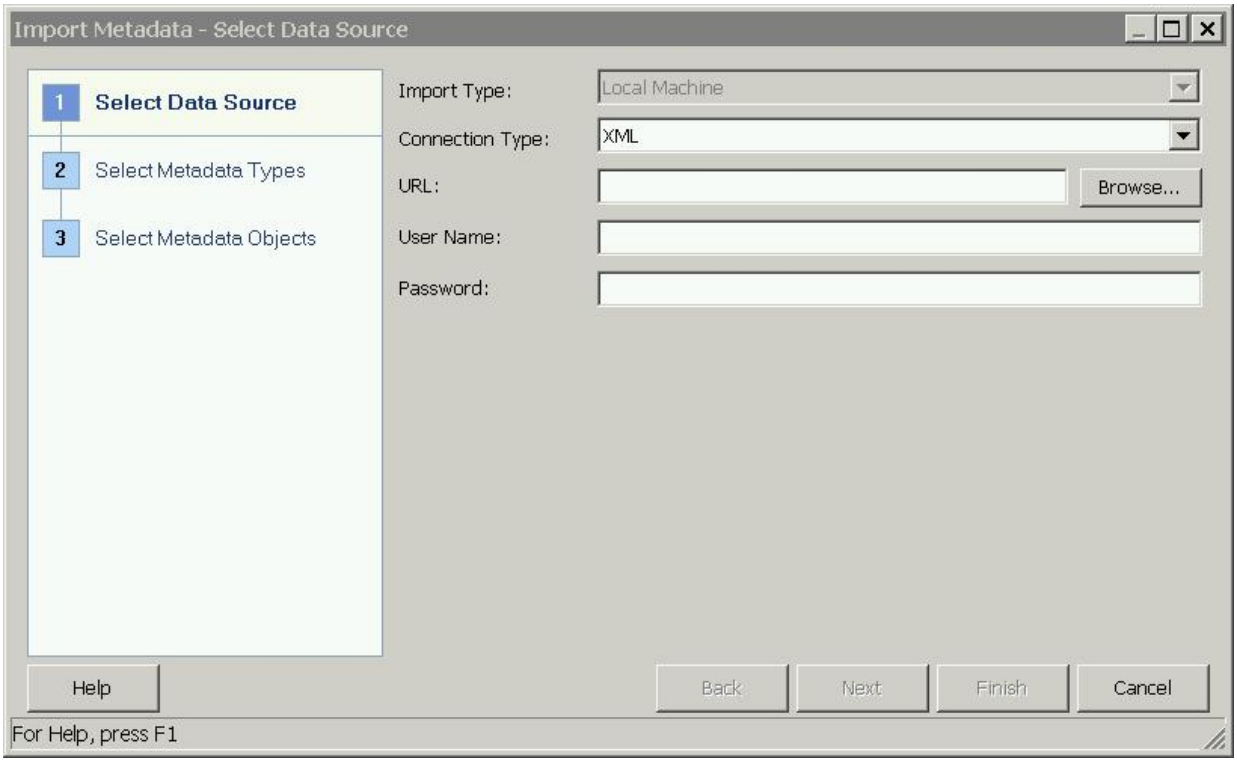


7.17.7 Using SPARQL Gateway as an XML Data Source to OBIEE

This section explains how to create an XML Data source for Oracle Business Intelligence Enterprise Edition (OBIEE), by integrating OBIEE with RDF using SPARQL Gateway as a bridge. (The specific steps and illustrations reflect the Oracle BI Administration Tool Version 11.1.1.3.0.100806.0408.000.)

1. Start the Oracle BI Administration Tool.
2. Click **File**, then **Import Metadata**. The first page of the Import Metadata wizard is displayed, as shown in Figure 7-9 (page 7-92).

Figure 7-9 Import Metadata - Select Data Source



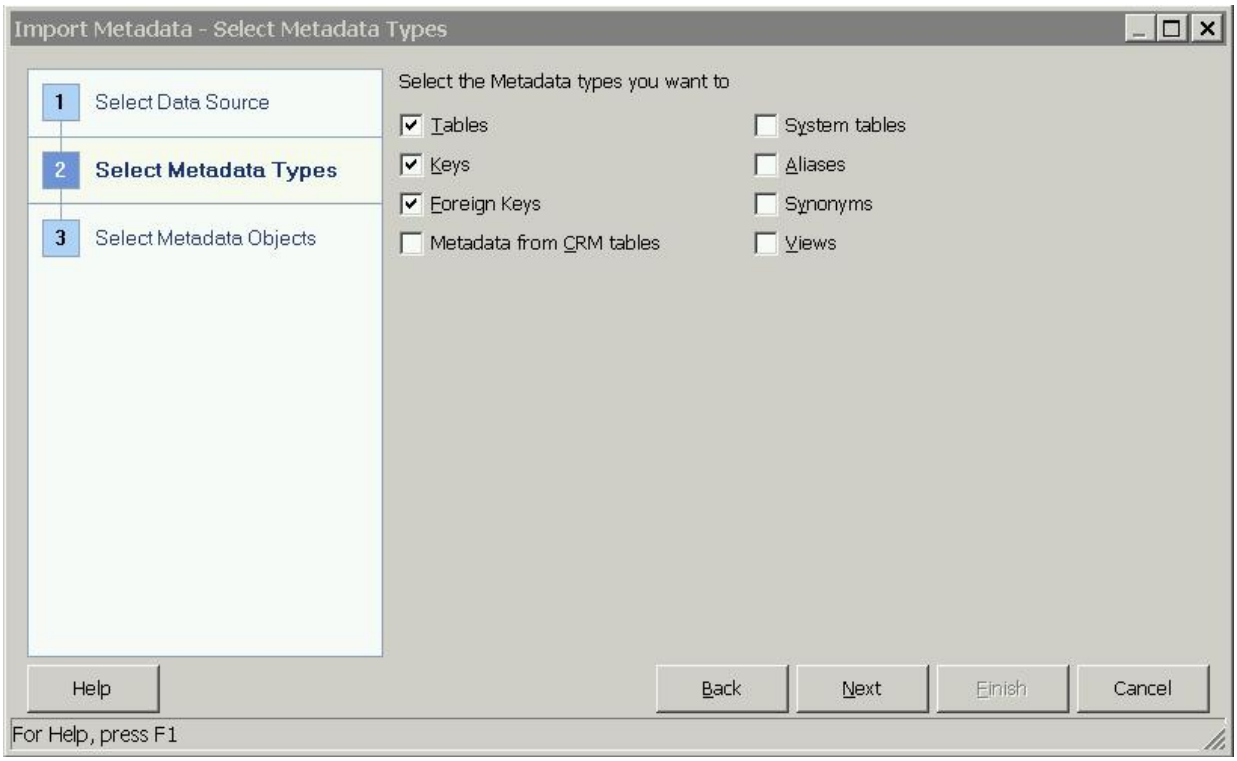
Connection Type: Select `XML`.

URL: URL for an application to interact with SPARQL Gateway, as explained in Using SPARQL Gateway with Semantic Data (page 7-78). You can also include the timeout and best effort options.

Ignore the **User Name** and **Password** fields.

3. Click **Next**. The second page of the Import Metadata wizard is displayed, as shown in Figure 7-10 (page 7-93).

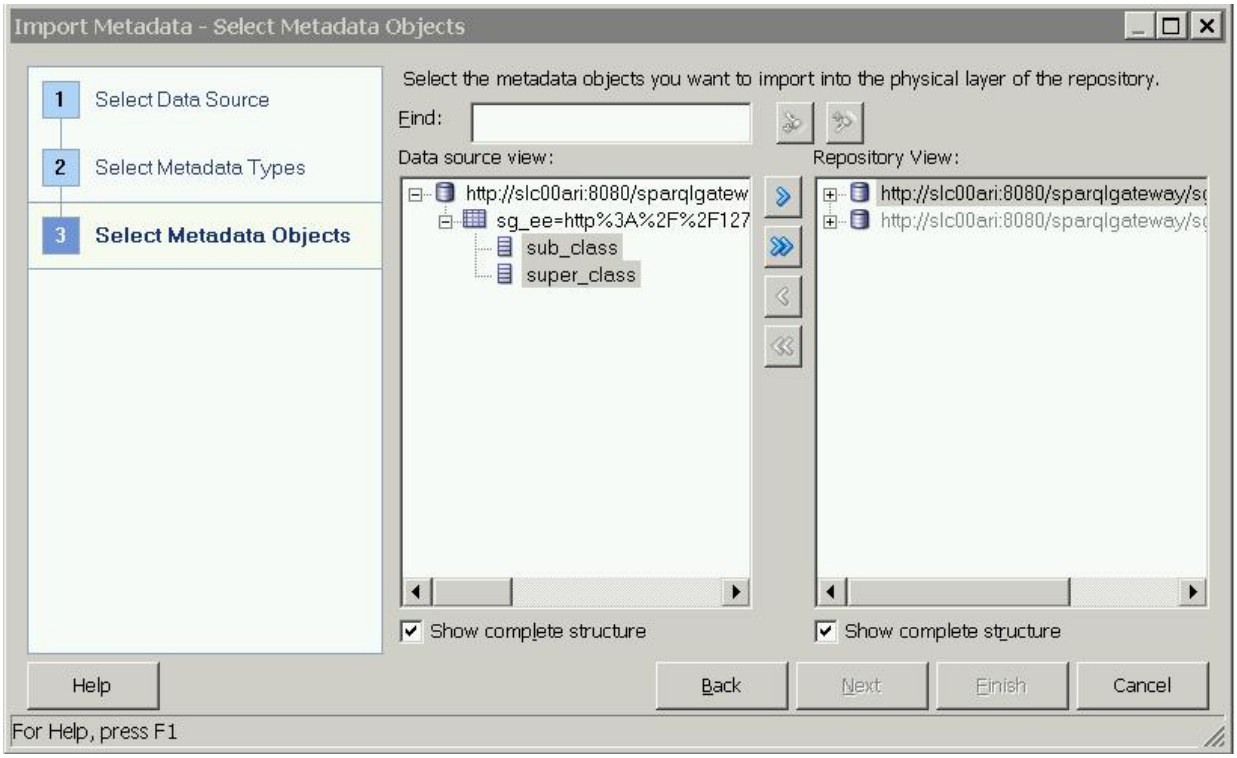
Figure 7-10 Import Metadata - Select Metadata Types



Select the desired metadata types to be imported. Be sure that **Tables** is included in the selected types.

4. Click **Next**. The third page of the Import Metadata wizard is displayed, as shown in Figure 7-11 (page 7-94).

Figure 7-11 Import Metadata - Select Metadata Objects



In the **Data Source View**, expand the node that has the table icon, select the column names (mapped from projected variables defined in the SPARQL SELECT statement), and click the right-arrow (>) button to move the selected columns to the **Repository View**.

5. Click **Finish**.
6. Complete the remaining steps for the usual BI Business Model work and Mapping and Presentation definition work, which are not specific to SPARQL Gateway or RDF data.

7.18 Deploying Joseki in Apache Tomcat or JBoss

If you choose not to deploy Joseki in Oracle WebLogic Server as described in Setting Up the SPARQL Service (page 7-4), you can deploy it in Apache Tomcat or JBoss as described in the following topics:

- Deploying Joseki in Apache Tomcat 6.0.29 or 7.0.42 (page 7-94)
- Deploying Joseki in JBoss 7.1.1 (page 7-96)

7.18.1 Deploying Joseki in Apache Tomcat 6.0.29 or 7.0.42

To deploy Joseki in Apache Tomcat 6.0.29 or 7.0.42, follow these steps.

1. Download and install Apache Tomcat 6.0.29 or 7.0.42.

   The directory root for Apache Tomcat installation will be referred to in these instructions as $CATALINA_HOME.

2. Set up environment variables using the following as examples:

```
setenv JAVA_HOME< your_path_here>/jdk16/
setenv PATH <your_path_here>/jdk16/bin/:$PATH
setenv CATALINA_HOME <your_path_here>/apache-tomcat-6.0.29 OR
setenv CATALINA_HOME <your_path_here>/apache-tomcat-7.0.42
```

3. Copy `ojdbc6.jar` into `${CATALINA_HOME}/lib`.

4. Modify `tomcat-users.xml` so that it includes the following:

```
<tomcat-users>
<role rolename="tomcat"/>
<role rolename="role1"/>
<user username="tomcat" password="<tomcat-password>" roles="tomcat,manager"/>
<user username="both" password="<tomcat-password>" roles="tomcat,role1"/>
<user username="role1" password="<tomcat-password>" roles="role1"/>
</tomcat-users>
```

   However, if you are using Apache Tomcat version 7.0.42, replace roles="tomcat,manager" with roles="tomcat,**manager-gui**" in the preceding code.

5. Start Tomcat:

```
$CATALINA_HOME/bin/startup.sh
```

   If this file does not have executable permission, enter the following command and then again attempt to start Tomcat:

```
chmod u+x $CATALINA_HOME/bin/startup.sh
```

6. In a browser go to: `http://hostname:8080/manager/html`

   If authentication is required, enter `tomcat` for the user and then enter the Tomcat password.

7. Deploy Joseki in Tomcat.

   a. In the Tomcat Manager page (`http://hostname:8080/manager/html`), select `Deploy directory or WAR file located on server`.

   b. For **Context Path**, enter: **/joseki**

   c. For **WAR or Directory URL**, enter the path to `joseki.war`, for example: `/tmp/jena_adapter/joseki_web_app/joseki.war`

   d. Click **Deploy**.

   A directory named `joseki` has been created under `apache-tomcat-6.0.29/webapps` or `apache-tomcat-7.0.42/webapps`. This directory will be referred to as JOSEKI_HOME.

8. Verify that Joseki is installed properly by going to:

```
http://<hostname>:8080/joseki
```

9. Create a J2EE data source named *OracleSemDS*.

   During the data source creation, you can specify a user and password for the database schema that contains the relevant semantic data against which SPARQL queries are to be executed. To create a data source in Tomcat, make the following changes:

   - In the global `$CATALINA_HOME/conf/server.xml` file, add the following resource in GlobalNamingResources (customize where needed):

```
<Resource name="OracleSemDS" auth="Container"
type="oracle.jdbc.pool.OracleDataSource"
driverClassName="oracle.jdbc.OracleDriver"
factory="oracle.jdbc.pool.OracleDataSourceFactory"
url="jdbc:oracle:thin:@hostname:port:sid" user="username"
password="password" maxActive="30" maxIdle="10" maxWait="-1"/>
```

- In the global `$CATALINA_HOME/conf/context.xml` file, add the following link:

```
<ResourceLink global="OracleSemDS" name="OracleSemDS"
type="oracle.jdbc.pool.OracleDataSource"/>
```

For more information about setting up data sources, see the Tomcat documentation.

10. Shut down and restart Tomcat:

```
$CATALINA_HOME/bin/shutdown.sh
$CATALINA_HOME/bin/startup.sh
```

11. Verify your deployment by going to (assuming that the web application is deployed at port 8080):

```
http://<hostname>:8080/joseki/querymgt?abortqid=0
```

You should see an XML response, which indicates a successful deployment of Joseki with the RDF Semantic Graph support for Apache Jena query management servlet.

7.18.2 Deploying Joseki in JBoss 7.1.1

To deploy Joseki in JBoss 7.1.1, follow these steps. (The steps have also been tested, with modifications as needed, against Red Hat JBoss Enterprise Application Platform 6.1.0.)

1. Download and install JBoss Application Server 7.1.1.

   These instructions assume that `jboss-as-7.1.1.Final/` is the top-level directory of the JBoss installation.

2. Install the JDBC driver:

```
create directory jboss-as-7.1.1.Final/modules/oracle/jdbc/main/
```

3. Copy `ojdbc6.jar` into this directory

4. Create `module.xml` in this directory with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
 <module xmlns="urn:jboss:module:1.0" name="oracle.jdbc">
     <resources>
        <resource-root path="ojdbc6.jar"/>
    </resources>
    <dependencies>
        <module name="javax.api"/>
         <module name="javax.transaction.api"/>
     </dependencies>
</module>
```

5. Modify `jboss-as-7.1.1.Final/standalone/configuration/standalone.xml` by adding the following line:

```
<driver name="OracleJDBCDriver" module="oracle.jdbc"/>
```

The modified `standalone.xml` file should include the following:

```
...
       <drivers>
         <driver name="OracleJDBCDriver" module="oracle.jdbc"/>
         <driver name="h2" module="com.h2database.h2">
              <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-
class>
          </driver>
       </drivers>
...
```

6. Create the necessary data source.

   a. Log into the JBoss AS Administration Console:

      `http://<hostname>:9990/console/App.html#server-overview`

   b. Click **Datasource**.

   c. Click **Profile**.

   d. Click **Add**, and enter the following:

      Name: `OracleSemDS`

      JNDI Name: `java:jboss/datasources/OracleSemDS`

   e. Select **OracleJDBCDriver**.

   f. Click **Next**.

      The following information is displayed:

      ```
      Connection URL: jdbc:oracle:thin:@hostname:port:sid     NOTE: customize
      Username:    scott                                       NOTE: customize
      Password:    tiger                                       NOTE: customize
      Security Domain: (Leave empty)
      ```

   g. Customize the information as needed and leave Security Domain blank, and click **Done**.

7. Highlight this new data source, click **Enable**, then click **Confirm**.

8. Deploy the `joseki.war` file using the JBoss Administration Console.

   a. Go to the following page:

      `http://<hostname>:9990/console/App.html#deployments`

   b. Click **Deployments**.

   c. Click **Manage Deployments**.

   d. Click **Add** and specify the `joseki.war` file.

8

RDF Semantic Graph Support for OpenRDF Sesame

Note:

This feature was previously referred to as the *Sesame Adapter for Oracle Database* and the *Sesame Adapter*.

RDF Semantic Graph support for OpenRDF Sesame (also referred to here as support for Sesame) integrates the popular Sesame Java APIs with Oracle Spatial and Graph RDF Semantic Graph support. Sesame is "a powerful Java framework for processing and handling RDF data. This includes creating, parsing, storing, inferencing and querying over such data." (See http://rdf4j.org.)

This chapter assumes that you are familiar with major concepts explained in RDF Semantic Graph Overview (page 1-1) and OWL Concepts (page 2-1). It also assumes that you are familiar with the overall capabilities and use of the Sesame Java framework; for information, click the Documentation link at http://rdf4j.org.

RDF Semantic Graph support for OpenRDF Sesame extends the semantic data management capabilities of Oracle Database RDF/OWL.

This chapter includes the following major topics:

- RDF Semantic Graph Support for OpenRDF Sesame Overview (page 8-2)
- Setup and Configuration for RDF Semantic Graph Support for OpenRDF Sesame (page 8-3)
- SEM_MATCH and RDF Semantic Graph Support for OpenRDF Sesame Queries Compared (page 8-15)
- Optimized Handling of SPARQL Queries (page 8-16)
- Recommendations for Best Performance (page 8-16)
- Additions to the SPARQL Syntax to Support Other Features (page 8-17)
- Support for Server-Side APIs (page 8-19)
- Oracle-Specific Extensions to Sesame APIs (page 8-22)
- Using the Sesame Console with RDF Semantic Graph Support for OpenRDF Sesame (page 8-24)
- Example Queries Using RDF Semantic Graph support for OpenRDF Sesame (page 8-27)

8.1 RDF Semantic Graph Support for OpenRDF Sesame Overview

Like the RDF Semantic Graph support for Apache Jena (described in RDF Semantic Graph Support for Apache Jena (page 7-1)), the support for Sesame provides a Java-based interface to Oracle semantic data through an open source framework and related tools. However, RDF Semantic Graph support for OpenRDF Sesame adds support for **context** to the data:

- Jena statements deal with triples (subject, predicate, object).
- Sesame statements deal with triples and contexts; a Sesame statement is a quad consisting of a triple and a context. For example, with hospital patient data, the context might contain the patient's URI.

RDF Semantic Graph support for OpenRDF Sesame provides a Java API for interacting with semantic data stored in Oracle Database. It also provides integration with the following Sesame tools:

- Sesame Server, which provides an HTTP SPARQL endpoint.
- Sesame Console, a command-line tool for creating and managing repositories. The available commands include the following:

```
connect  Connects to a (local or remote) set of repositories
disconnect  Disconnects from the current set of repositories
create  Creates a new repository
drop  Drops a repository
open  Opens a repository to work on, takes a repository ID as argument
close  Closes the current repository
show  Displays an overview of various resources
load  Loads a data file into a repository, takes a file path or URL as argument
verify  Verifies the syntax of an RDF data file, takes a file path or URL as
argument
clear  Removes data from a repository
serql  Evaluates the SeRQL query, takes a query as argument
sparql  Evaluates the SPARQL query, takes a query as argument
set  Allows various console parameters to be set
exit, quit  Exits the console
```

- OpenRDF Workbench, a graphical user interface to the Sesame Server.

The features provided by the support for Sesame include:

- Loading (bulk and incremental), exporting, and removing statements, with and without context
- Querying of data, with and without context
- Inferencing, where inferred data is considered during query execution (if `includeInferred=true`), and where inferred triples have a null context
- Managing namespaces

RDF Semantic Graph support for OpenRDF Sesame implements various interfaces of the Sesame Storage and Inference Layer (SAIL) API. For example, the class `OracleSailConnection` is an Oracle implementation of the Sesame `SailConnection` interface, and the class `OracleSailStore` is an Oracle implementation of the Sesame `Sail` interface.

A typical usage flow for the support for Sesame might include the excerpt shown in Example 8-1 (page 8-3).

Example 8-1 Sample Usage flow for RDF Semantic Graph Support for OpenRDF Sesame

```
String query =
" PREFIX foaf: <http://xmlns.com/foaf/0.1/> "
" PREFIX dc: <http://purl.org/dc/elements/1.1/> "
" select ?s ?p ?o ?name WHERE {?s ?p ?o . OPTIONAL {?o foaf:name ?name .} } ";

SailRepository sr = new SailRepository(
   new OracleSailStore(oracle, modelName,…attachmentForInference));
RepositoryConnection repConn = sr.getConnection();
// Data loading can happen here.
repConn.add(alice, friendOf, bob, context1);

TupleQuery tq = repConn.prepareTupleQuery(QueryLanguage.SPARQL, query);
TupleQueryResult tqr = tq.evaluate();
while(tqr.hasNext()) {
    psOut.print((tqr.next().toString()));
};
tqr.close()
repConn.close();
sr.shutDown();
```

8.2 Setup and Configuration for RDF Semantic Graph Support for OpenRDF Sesame

Setting up and configuring the support for Sesame involves the following actions:

1. Setting Up the Software Environment (page 8-3)
2. Setting Up the SPARQL Service (page 8-4)
3. Setting Up the RDF Semantic Graph Environment (page 8-14)

8.2.1 Setting Up the Software Environment

To use the support for Sesame, you must first ensure that the system environment has the necessary software, including Oracle Database 11*g* Release 2 or later with the Spatial and Graph and Partitioning options and with RDF Semantic Graph support enabled, Sesame version 2.3.1, RDF Semantic Graph support for OpenRDF Sesame, and JDK 6. You can set up the software environment by performing these actions:

1. Install Oracle Database Enterprise Edition with the Oracle Spatial and Graph and Partitioning Options.
2. Enable the support for RDF Semantic Graph, as explained in Enabling RDF Semantic Graph Support (page A-1).
3. Download the Sesame 2.3.1 `onejar` library and include it in your classpath, as explained in the "Setting up to use the Sesame libraries" chapter of the *User Guide*.
4. Install the Sesame 2.3.1 Server, as explained in the "Server software installation" chapter of the *User Guide*.
5. Download RDF Semantic Graph support for OpenRDF Sesame from the RDF Semantic Graph page (`http://www.oracle.com/technetwork/database/`

options/semantic-tech/and click **Downloads**), and unzip it into a temporary directory, such as (on a Linux system) `/tmp/sesame_adapter`. (If this temporary directory does not already exist, create it before the unzip operation.)

RDF Semantic Graph support for OpenRDF Sesame directories and files have the following structure:

```
 examples/
   examples/Example1.java
   examples/Example2.java
   .
   .
   .
lib/
   lib/sdordfsesame.jar
web/
   web/web.xml
javadoc/
   javadoc/javadoc.zip
```

6. Copy `ojdbc6.jar` into <*Sesame_DIR*>`/lib` (Linux) or <*Sesame_DIR*>`\lib` (Windows). (`ojdbc6.jar` is in `$ORACLE_HOME/jdbc/lib` or `%ORACLE_HOME%\jdbc\lib`.)
7. Copy `sdordf.jar` into <*Sesame_DIR*>`/lib` (Linux) or <*Sesame_DIR*>`\lib` (Windows). (`sdordf.jar` is in `$ORACLE_HOME/md/jlib` or `%ORACLE_HOME%\md\jlib`.)
8. If JDK 6 is not already installed, install it.
9. If the JAVA_HOME environment variable does not already refer to the JDK 6 installation, define it accordingly. For example:

   ```
   setenv JAVA_HOME /usr/local/packages/jdk16/
   ```

10. If the SPARQL service to support the SPARQL protocol is not set up, set it up as explained in Setting Up the SPARQL Service (page 8-4).

After setting up the software environment, ensure that your RDF Semantic Graph environment can enable you to use the support for Sesame to perform queries, as explained in Setting Up the RDF Semantic Graph Environment (page 8-14).

8.2.2 Setting Up the SPARQL Service

Setting up a SPARQL endpoint using the support for Sesame involves downloading the Sesame Server, with which client applications can communicate over HTTP protocol. The Sesame Server (and OpenRDF Workbench) is bundled with the Sesame 2.3.1 release package. This section explains how to set up a SPARQL service using a servlet deployed in WebLogic Server. The number and complexity of the steps reflect the fact that Oracle is not permitted to bundle all the dependent third-party libraries in a `.war` or `.ear` file.

1. Download and Install Oracle WebLogic Server 11g Release 1 (10.3.1).
2. Ensure that you have Java 6 installed, because it is required by Sesame.
3. Download the Sesame 2.3.1 release package (in .zip format) from `http://sourceforge.net/projects/sesame/files/Sesame%202/`.
4. Unpack the .zip file into a temporary directory. For example:

```
mkdir /tmp/sesame
cp openrdf-sesame-2.3.1-sdk.zip /tmp/sesame/
cd /tmp/sesame
unzip openrdf-sesame-2.3.1-sdk.zip
```

5. Ensure that you have downloaded and unzipped RDF Semantic Graph support for OpenRDF Sesame, as explained in Setting Up the Software Environment (page 8-3).

6. Create a directory named `openrdf-sesame.war` at the same level as the `Sesame_adapter` directory, and go to it. For example:

```
/tmp/openrdf-sesame.war
cd /tmp/openrdf-sesame.war
```

7. Unzip necessary files into the directory created in the preceding step:

```
cd /tmp/openrdf-sesame.war
unzip /tmp/sesame/openrdf-sesame-2.3.1/war/openrdf-sesame.war
```

8. Copy necessary files, as follows. (Assume that ORACLE_HOME points to the home directory of an Oracle Database.)

```
cp /tmp/Sesame_adapter/web/web.xml /tmp/openrdf-sesame.war/WEB-INF/web.xml
cp /tmp/Sesame_adapter/lib/*.jar   /tmp/openrdf-sesame.war/WEB-INF/lib
cp $ORACLE_HOME/md/jlib/sdordf.jar /tmp/openrdf-sesame.war/WEB-INF/lib
cp $ORACLE_HOME/jdbc/lib/ojdbc6.jar /tmp/openrdf-sesame.war/WEB-INF/lib
```

9. Using the WebLogic Server Administration console, create a J2EE data source named *OracleSemDS*. During the data source creation, you can specify a user and password for the database schema that contains the relevant semantic data against which SPARQL queries are to be executed.

   If you need help in creating this data source, see Creating the Required Data Source Using WebLogic Server (page 8-13).

10. Go to the `autodeploy` directory of WebLogic Server and copy files, as follows. (For information about auto-deploying applications in development domains, see: `http://docs.oracle.com/cd/E11035_01/wls100/deployment/autodeploy.html`)

```
cd <domain_name>/autodeploy
cp -rf  /tmp/openrdf-sesame.war  <domain_name>/autodeploy
```

    In the preceding example, <*domain_name*> is the name of a WebLogic Server domain.

    Note that while you can run a WebLogic Server domain in two different modes, development and production, only development mode allows you use the auto-deployment feature.

11. Check the files and the directory structure, as in the following example:

```
autodeploy/% ls -1R ./openrdf-sesame.war/

openrdf-sesame.war/:
favicon.ico
favicon.png
images
META-INF
styles
WEB-INF
```

```
openrdf-sesame.war/images:
DEBUG.png
ERROR.png
INFO.png
logo.png
productgroup.png
transparent.png
WARN.png

openrdf-sesame.war/META-INF:
MANIFEST.MF
maven

openrdf-sesame.war/META-INF/maven:
info.aduna.appbase
org.openrdf.sesame

openrdf-sesame.war/META-INF/maven/info.aduna.appbase:
aduna-appbase-webapp-system

openrdf-sesame.war/META-INF/maven/info.aduna.appbase/aduna-appbase-webapp-system:
pom.properties
pom.xml

openrdf-sesame.war/META-INF/maven/org.openrdf.sesame:
sesame-http-server

openrdf-sesame.war/META-INF/maven/org.openrdf.sesame/sesame-http-server:
pom.properties
pom.xml

openrdf-sesame.war/styles:
basic
default
msie-png-alpha.css
msie-png-alpha.png
w3-html40-recommended.css

openrdf-sesame.war/styles/basic:
all.css

openrdf-sesame.war/styles/default:
images
msie-minheight.css
print.css
screen.css

openrdf-sesame.war/styles/default/images:
bg_body.png
bg_header.png
bg_html.png
hr_menu.png

openrdf-sesame.war/WEB-INF:
aduna-webapp-servlet.xml
aduna-webapp-system-servlet.xml
classes
includes
lib
openrdf-http-server-servlet.xml
```

```
sample-web.xml
taglibs
tags
urlrewrite.xml
views
web.xml

openrdf-sesame.war/WEB-INF/classes:
logback.xml

openrdf-sesame.war/WEB-INF/includes:
components
stylesheets.html.jspf
taglibs.jspf

openrdf-sesame.war/WEB-INF/includes/components:
bodyStart.html.jspf
bodyStop.html.jspf
ContentHeader.html.jspf
Footer.html.jspf
Header.html.jspf
head.html.jspf
htmlStart.html.jspf
htmlStop.html.jspf
logfilterform.html.jspf
logpaginationfooter.html.jspf
logpaginationheader.html.jspf
Message.html.jspf
Navigation.html.jspf
page.html.jspf
tabs.html.jspf

openrdf-sesame.war/WEB-INF/lib:
activation-1.1.jar
aduna-appbase-core-3.9.0.jar
aduna-appbase-logging-api-3.9.0.jar
aduna-appbase-logging-file-3.9.0.jar
aduna-appbase-webapp-base-core-3.9.0.jar
aduna-appbase-webapp-system-core-3.9.0.jar
aduna-commons-collections-2.7.0.jar
aduna-commons-concurrent-2.6.0.jar
aduna-commons-i18n-1.3.0.jar
aduna-commons-io-2.8.0.jar
aduna-commons-iteration-2.7.0.jar
aduna-commons-lang-2.8.0.jar
aduna-commons-net-2.6.0.jar
aduna-commons-platform-info-2.8.0.jar
aduna-commons-text-2.6.0.jar
aduna-commons-webapp-core-2.8.0.jar
aduna-commons-xml-2.6.0.jar
aopalliance-1.0.jar
asm-1.5.3.jar
cglib-2.1_3.jar
commons-codec-1.3.jar
commons-dbcp-1.2.2.jar
commons-httpclient-3.1.jar
commons-pool-1.3.jar
jcl-over-slf4j-1.5.10.jar
jstl-1.1.2.jar
logback-classic-0.9.18.jar
logback-core-0.9.18.jar
```

```
ojdbc6.jar
sdordf.jar
sdordfsesame.jar
sesame-http-client-2.3.1.jar
sesame-http-protocol-2.3.1.jar
sesame-http-server-2.3.1.jar
sesame-http-server-spring-2.3.1.jar
sesame-model-2.3.1.jar
sesame-query-2.3.1.jar
sesame-queryalgebra-evaluation-2.3.1.jar
sesame-queryalgebra-model-2.3.1.jar
sesame-queryparser-api-2.3.1.jar
sesame-queryparser-serql-2.3.1.jar
sesame-queryparser-sparql-2.3.1.jar
sesame-queryresultio-api-2.3.1.jar
sesame-queryresultio-binary-2.3.1.jar
sesame-queryresultio-sparqljson-2.3.1.jar
sesame-queryresultio-sparqlxml-2.3.1.jar
sesame-queryresultio-text-2.3.1.jar
sesame-repository-api-2.3.1.jar
sesame-repository-contextaware-2.3.1.jar
sesame-repository-dataset-2.3.1.jar
sesame-repository-event-2.3.1.jar
sesame-repository-http-2.3.1.jar
sesame-repository-manager-2.3.1.jar
sesame-repository-sail-2.3.1.jar
sesame-rio-api-2.3.1.jar
sesame-rio-n3-2.3.1.jar
sesame-rio-ntriples-2.3.1.jar
sesame-rio-rdfxml-2.3.1.jar
sesame-rio-trig-2.3.1.jar
sesame-rio-trix-2.3.1.jar
sesame-rio-turtle-2.3.1.jar
sesame-runtime-2.3.1.jar
sesame-sail-api-2.3.1.jar
sesame-sail-inferencer-2.3.1.jar
sesame-sail-memory-2.3.1.jar
sesame-sail-nativerdf-2.3.1.jar
sesame-sail-rdbms-2.3.1.jar
slf4j-api-1.5.10.jar
spring-aop-2.5.6.jar
spring-beans-2.5.6.jar
spring-context-2.5.6.jar
spring-context-support-2.5.6.jar
spring-core-2.5.6.jar
spring-web-2.5.6.jar
spring-webmvc-2.5.6.jar
standard-1.1.2.jar
urlrewritefilter-3.0.4.jar

openrdf-sesame.war/WEB-INF/taglibs:
navigation.tld

openrdf-sesame.war/WEB-INF/tags:
errors.tag

openrdf-sesame.war/WEB-INF/views:
home
overview.jsp
system
```

```
openrdf-sesame.war/WEB-INF/views/home:
overview.jsp

openrdf-sesame.war/WEB-INF/views/system:
info
logging
overview.jsp

openrdf-sesame.war/WEB-INF/views/system/info:
debug.jsp
overview.jsp

openrdf-sesame.war/WEB-INF/views/system/logging:
overview.jsp
```

12. Start or restart WebLogic Server.

13. Verify your deployment by using your Web browser to connect to a URL in the following format (assume that the Web application is deployed at port 7001): `http://<hostname>:7001/openrdf-sesame/querymgt?abortqid=0`

    You should see an XML response, which indicates a successful deployment of the Sesame Server with the support for Sesame query management servlet.

You can also install the OpenRDF Workbench, which is a graphical user interface (GUI) for communicating with the Sesame Server. To deploy the OpenRDF Workbench into the Oracle WebLogic Server, you must have performed at least the first 7 steps in the preceding instructions for setting up the SPARQL endpoint (explained earlier in this section); then, follow these steps:

1. Go to the `autodeploy` directory of WebLogic Server and copy files, as follows. (For information about auto-deploying applications in development domains, see: http://docs.oracle.com/cd/E11035_01/wls100/deployment/autodeploy.html)

   ```
   cd <domain_name>/autodeploy
   cp -rf  /tmp/openrdf-workbench.war  <domain_name>/autodeploy
   ```

   In the preceding example, *<domain_name>* is the name of a WebLogic Server domain.

2. Check the files and the directory structure, as in the following example:

   ```
   autodeploy/% ls -1R ./openrdf-workbench.war/
    openrdf-workbench.war/:

   favicon.ico
   favicon.png
   images
   locale
   META-INF
   scripts
   styles
   transformations
   WEB-INF

   openrdf-workbench.war/images:
   affirmative.png
   edit.png
   logo.png
   negative.png
   ```

```
productgroup.png
server_network.png
server.png
view.png

openrdf-workbench.war/locale:
messages.xsl

openrdf-workbench.war/META-INF:
MANIFEST.MF
maven

openrdf-workbench.war/META-INF/maven:
org.openrdf.sesame

openrdf-workbench.war/META-INF/maven/org.openrdf.sesame:
sesame-http-workbench

openrdf-workbench.war/META-INF/maven/org.openrdf.sesame/sesame-http-workbench:
pom.properties
pom.xml

openrdf-workbench.war/scripts:
cookies.html

openrdf-workbench.war/styles:
basic
default
msie-png-alpha.css
msie-png-alpha.png
w3-html40-recommended.css

openrdf-workbench.war/styles/basic:
all.css

openrdf-workbench.war/styles/default:
images
msie-minheight.css
print.css
screen.css

openrdf-workbench.war/styles/default/images:
bg_body.png
bg_header.png
bg_html.png
hr_menu.png

openrdf-workbench.war/transformations:
add.xsl
boolean.xsl
clear.xsl
contexts.xsl
create-memory-rdfs-dt.xsl
create-memory-rdfs.xsl
create-memory.xsl
create-mysql.xsl
create-native-rdfs-dt.xsl
create-native-rdfs.xsl
create-native.xsl
create-pgsql.xsl
create-remote.xsl
```

```
create.xsl
delete.xsl
explore.xsl
export.xsl
graph.xsl
information.xsl
list.xsl
namespaces.xsl
query.xsl
remove.xsl
repositories.xsl
server.xsl
summary.xsl
table.xsl
template.xsl
tuple.xsl
types.xsl
url-encode.xsl

openrdf-workbench.war/WEB-INF:
classes
lib
web.xml

openrdf-workbench.war/WEB-INF/classes:
org

openrdf-workbench.war/WEB-INF/classes/org:
openrdf

openrdf-workbench.war/WEB-INF/classes/org/openrdf:
workbench

openrdf-workbench.war/WEB-INF/classes/org/openrdf/workbench:
base
commands
exceptions
proxy
RepositoryServlet.class
util

openrdf-workbench.war/WEB-INF/classes/org/openrdf/workbench/base:
BaseRepositoryServlet.class
BaseServlet.class
TransformationServlet.class
TupleServlet.class

openrdf-workbench.war/WEB-INF/classes/org/openrdf/workbench/commands:
AddServlet.class
ClearServlet.class
ContextsServlet.class
CreateServlet.class
DeleteServlet.class
ExploreServlet.class
ExportServlet.class
InformationServlet.class
InfoServlet.class
NamespacesServlet.class
QueryServlet.class
RemoveServlet.class
RepositoriesServlet.class
```

```
SummaryServlet.class
TypesServlet.class

openrdf-workbench.war/WEB-INF/classes/org/openrdf/workbench/exceptions:
BadRequestException.class
MissingInitParameterException.class

openrdf-workbench.war/WEB-INF/classes/org/openrdf/workbench/proxy:
CookieCacheControlFilter$CacheAwareRequest.class
CookieCacheControlFilter$CacheAwareResponse.class
CookieCacheControlFilter.class
ProxyRepositoryServlet.class
RedirectFilter.class
WorkbenchGateway.class
WorkbenchServlet.class

openrdf-workbench.war/WEB-INF/classes/org/openrdf/workbench/util:
BasicServletConfig.class
ConfigTemplate.class
DynamicHttpRequest.class
TupleResultBuilder.class
WorkbenchRequest.class

openrdf-workbench.war/WEB-INF/lib:
aduna-appbase-core-3.9.0.jar
aduna-appbase-logging-api-3.9.0.jar
aduna-appbase-logging-file-3.9.0.jar
aduna-commons-collections-2.7.0.jar
aduna-commons-concurrent-2.6.0.jar
aduna-commons-i18n-1.3.0.jar
aduna-commons-io-2.8.0.jar
aduna-commons-iteration-2.7.0.jar
aduna-commons-lang-2.8.0.jar
aduna-commons-net-2.6.0.jar
aduna-commons-platform-info-2.8.0.jar
aduna-commons-text-2.6.0.jar
aduna-commons-xml-2.6.0.jar
commons-cli-1.1.jar
commons-codec-1.3.jar
commons-dbcp-1.2.2.jar
commons-fileupload-1.2.1.jar
commons-httpclient-3.1.jar
commons-io-1.3.2.jar
commons-pool-1.3.jar
jcl-over-slf4j-1.5.10.jar
sesame-console-2.3.1.jar
sesame-http-client-2.3.1.jar
sesame-http-protocol-2.3.1.jar
sesame-model-2.3.1.jar
sesame-query-2.3.1.jar
sesame-queryalgebra-evaluation-2.3.1.jar
sesame-queryalgebra-model-2.3.1.jar
sesame-queryparser-api-2.3.1.jar
sesame-queryparser-serql-2.3.1.jar
sesame-queryparser-sparql-2.3.1.jar
sesame-queryresultio-api-2.3.1.jar
sesame-queryresultio-binary-2.3.1.jar
sesame-queryresultio-sparqljson-2.3.1.jar
sesame-queryresultio-sparqlxml-2.3.1.jar
sesame-queryresultio-text-2.3.1.jar
sesame-repository-api-2.3.1.jar
```

```
sesame-repository-contextaware-2.3.1.jar
sesame-repository-dataset-2.3.1.jar
sesame-repository-event-2.3.1.jar
sesame-repository-http-2.3.1.jar
sesame-repository-manager-2.3.1.jar
sesame-repository-sail-2.3.1.jar
sesame-rio-api-2.3.1.jar
sesame-rio-n3-2.3.1.jar
sesame-rio-ntriples-2.3.1.jar
sesame-rio-rdfxml-2.3.1.jar
sesame-rio-trig-2.3.1.jar
sesame-rio-trix-2.3.1.jar
sesame-rio-turtle-2.3.1.jar
sesame-runtime-2.3.1.jar
sesame-sail-api-2.3.1.jar
sesame-sail-inferencer-2.3.1.jar
sesame-sail-memory-2.3.1.jar
sesame-sail-nativerdf-2.3.1.jar
sesame-sail-rdbms-2.3.1.jar
slf4j-api-1.5.10.jar
slf4j-jdk14-1.5.10.jar
```

3. Start or restart WebLogic Server.

4. Verify your deployment by using your Web browser to connect to a URL in the following format (assume that the Web application is deployed at port 7001): `http://<hostname>:7001/openrdf-workbench`

   You should see a web page with the OpenRDF Workbench logo. Click the **Repositories** link on the left to see a list of repositories.

8.2.2.1 Creating the Required Data Source Using WebLogic Server

If you need help creating the required J2EE data source using the WebLogic Server admin console, you can follow these steps:

1. Login to: `http://<hostname>:7001/console`

2. In the Domain Structure panel, click **Services**.

3. Click **JDBC**

4. Click **Data Sources**.

5. In the Summary of JDBC Data Sources panel, click **New** under the Data Sources table.

6. In the Create a New JDBC Data Source panel, enter or select the following values.

   Name: `OracleSemDS`

   JNDI Name: `OracleSemDS`

   Database Type: `Oracle`

   Database Driver: `Oracle's Driver (Thin) Versions: 9.0.1,9.2.0,10,11`

7. Click **Next** twice.

8. In the Connection Properties panel, enter the appropriate values for the **Database Name**, **Host Name**, **Port**, **Database User Name** (schema that contains semantic data), **Password** fields.
9. Click **Next**.
10. Select (check) the target server or servers to which you want to deploy this `OracleSemDS` data source.
11. Click **Finish**.

    You should see a message that all changes have been activated and no restart is necessary.

8.2.2.2 Configuring the SPARQL Service

You can use the `OracleSailStore` API and other relevant APIs to create semantic models, load data (incrementally or in bulk), and perform logical inference.

After the semantic models are in the Oracle database, you can use the Sesame Console (described in Using the Sesame Console with RDF Semantic Graph Support for OpenRDF Sesame (page 8-24)) to connect to the Sesame Server and create an Oracle semantic model-backed repository. For example, you might create a repository with the repository ID `myOracleRepos` that uses a semantic model named `LUBM` and the `OWLPRIME` rulebase.

This newly created repository provides a new service endpoint on the Sesame Server.

8.2.2.3 Terminating Long-Running SPARQL Queries

Because some applications need to be able to terminate long-running SPARQL queries, an abort framework has been introduced with RDF Semantic Graph support for OpenRDF Sesame and the Sesame Server setup. Basically, for queries that may take a long time to run, you must stamp each with a unique query ID (qid) value.

For example, the following SPARQL query selects out the subject of all triples. A query ID (qid) is set so that this query can be terminated upon request.

```
PREFIX ORACLE_SEM_FS_NS:  <http://example.com/semtech#qid=8761>
SELECT ?subject WHERE {?subject ?property ?object }
```

The `qid` attribute value is of long integer type. You can choose a value for the `qid` for a particular query based on your own application needs.

To terminate a SPARQL query that has been submitted with a qid value, applications can send an abort request to a servlet in the following format and specify a matching QID value

```
http://<hostname>:7001/openrdf-sesame/querymgt?abortqid=8761
```

8.2.3 Setting Up the RDF Semantic Graph Environment

To use the support for Sesame to perform queries, you can connect as any user (with suitable privileges) and use any models in the semantic network. If your RDF Semantic Graph environment already meets the requirements, you can go directly to compiling and running Java code that uses the support for Sesame. If your RDF Semantic Graph environment is not yet set up to be able to use the support for Sesame, you can perform actions similar to the following example steps:

1. Connect as SYS with the SYSDBA role:

```
sqlplus sys/<password-for-sys> as sysdba
```

2. Create a tablespace for the system tables. For example:

```
CREATE TABLESPACE rdf_users datafile 'rdf_users01.dbf'
    size 128M reuse autoextend on next 64M
    maxsize unlimited segment space management auto;
```

3. Create the semantic network. For example:

```
EXECUTE sem_apis.create_sem_network('RDF_USERS');
```

4. Create a database user (for connecting to the database to use the semantic network and the support for Sesame). For example:

```
CREATE USER rdfusr IDENTIFIED BY <password-for-udfusr>
                  DEFAULT TABLESPACE rdf_users;
```

5. Grant the necessary privileges to this database user. For example:

```
GRANT connect, resource TO rdfusr;
```

6. To use the support for Sesame with your own semantic data, perform the appropriate steps to store data, create a model, and create database indexes, as explained in Quick Start for Using Semantic Data (page 1-92). Then perform queries by compiling and running Java code; see Example Queries Using RDF Semantic Graph support for OpenRDF Sesame (page 8-27) for information about example queries.

   To use the support for Sesame with supplied example data, see Example Queries Using RDF Semantic Graph support for OpenRDF Sesame (page 8-27).

8.3 SEM_MATCH and RDF Semantic Graph Support for OpenRDF Sesame Queries Compared

There are two ways to query semantic data stored in Oracle Database: SEM_MATCH-based SQL statements and SPARQL queries through RDF Semantic Graph support for OpenRDF Sesame. Queries using each approach are similar in appearance, but there are important behavioral differences. To ensure consistent application behavior, you must understand the differences and use care when dealing with query results coming from SEM_MATCH queries and SPARQL queries.

The following simple examples show the two approaches.

Query 1 (SEM_MATCH-based)

```
select s, p, o
    from table(sem_match('{?s ?p ?o}', sem_models('Test_Model'), ....))
```

Query 2 (SPARQL query through RDF Semantic Graph support for OpenRDF Sesame)

```
select ?s ?p ?o
where {?s ?p ?o}
```

These two queries perform the same kind of functions; however, there are some important differences. Query 1 (SEM_MATCH-based):

- Reads all triples out of `Test_Model`.

- Does not differentiate among URI, bNode, plain literals, and typed literals, and it does not handle long literals.
- Does not unescape certain characters (such as `'\n'`).

Query 2 (SPARQL query executed through RDF Semantic Graph support for OpenRDF Sesame) also reads all triples out of `Test_Model` (assume it executed a call to `ModelOracleSem` referring to the same underlying `Test_Model`). However, Query 2:

- Reads out additional columns (as opposed to just the `s`, `p`, and `o` columns with the SEM_MATCH table function), to differentiate URI, bNodes, plain literals, typed literals, and long literals. This is to ensure proper creation of Sesame Node objects.
- Unescapes those characters that are escaped when stored in Oracle Database

Blank node handling is another difference between the two approaches:

- In a SEM_MATCH-based query, blank nodes are always treated as constants.
- In a SPARQL query, a blank node that *is not* wrapped inside < and > is treated as a variable when the query is executed through the support for Sesame. This matches the SPARQL standard semantics. However, a blank node that *is* wrapped inside < and > is treated as a constant when the query is executed, and the support for Sesame adds a proper prefix to the blank node label as required by the underlying data modeling.

The maximum length for the name of a semantic model created using the RDF Semantic Graph support for OpenRDF Sesame API is 22 characters.

8.4 Optimized Handling of SPARQL Queries

This section describes some performance-related features of RDF Semantic Graph support for OpenRDF Sesame that can enhance SPARQL query processing. These features are performed automatically by default.

8.4.1 Compilation of SPARQL Queries to a Single SEM_MATCH Call

SPARQL queries involving DISTINCT, OPTIONAL, GRAPH, FILTER, UNION, ORDER BY, and LIMIT are converted to a single SQL query based on the SEM_MATCH table function. For SPARQL ASK, DESCRIBE, and CONSTRUCT queries, the support for Sesame employs additional query transformation logic and mid-tier processing.

If the SQL query based on SEM_MATCH fails to execute, the support for Sesame employs the Sesame evaluation strategy to execute the query.

8.5 Recommendations for Best Performance

The following are recommendations for enhancing the performance of queries using RDF Semantic Graph support for OpenRDF Sesame:

- Put only relevant triple patterns in the GRAPH clause of a SPARQL query, to reduce the number of additional joins with the application table. (For an explanation, see Statement Storage and Implications for RDF Semantic Graph Support for OpenRDF Sesame Queries (page 8-17).)

- For the triple patterns inside GRAPH clause of a named graph query, do not use variables with null bindings. A null binding will not match any row in the join with the relevant application table, and therefore this binding will be dropped.
- Because of the preceding item, do not put an OPTIONAL clause in a GRAPH clause, or a GRAPH clause in an OPTIONAL clause.
- Do not use blank nodes for the CONTEXT column in the application table (see Statement Storage and Implications for RDF Semantic Graph Support for OpenRDF Sesame Queries (page 8-17)), because blank nodes in named graphs from two different semantic models will be treated as the same resource if they have the same label. This is not the case for blank nodes in triples (they are stored separately if coming from different models).

8.5.1 Statement Storage and Implications for RDF Semantic Graph Support for OpenRDF Sesame Queries

Because the MDSYS.SEMM_*model-name* view (described in Table 1-2 (page 1-4) in Metadata for Models (page 1-3)) does not have columns for the graph name or context for each model, these two pieces of information are not stored inside that Oracle semantic model. Rather, this information is kept in the CONTEXT column in the user's application table.

When an `OracleSailStore` object is created for the first time using a model name that does not exist in the semantic network, a new application table is created automatically, and it includes these columns:

- TRIPLE, of type SDO_RDF_TRIPLE_S
- CONTEXT, of type VARCHAR2

A semantic model with the specified model name is then created. To enhance the performance of named graph queries and statement-oriented operations, a default index is created on the application table, and this index covers the CONTEXT column (leading column) as well as the three numeric ID attributes for triple's subject, predicate, and object field of the TRIPLE column.

When a SPARQL query runs against a semantic model, if the query does not use a named graph (that is, does not contain a GRAPH clause), the generated SQL statement will not use the model's application table. However, if the query has a GRAPH clause, the generated SQL statement will join a SEM_MATCH-based subquery with the model's application table to scope the results down to relevant contexts or named graphs. The number of joins with the application table depends on how many triple patterns are in the GRAPH clause, and therefore it is best to put only relevant triple patterns in the GRAPH clause, to reduce the number of additional joins with the application table.

8.6 Additions to the SPARQL Syntax to Support Other Features

RDF Semantic Graph support for OpenRDF Sesame allows you to pass in hints and additional query options. It implements these capabilities by overloading the SPARQL namespace prefix syntax by using Oracle-specific namespaces that contain query options. The namespaces are in the form `PREFIX ORACLE_SEM_xx_NS`, where *xx* indicates the type of feature (such as `HT` for hint or `AP` for additional predicate)

8.6.1 SQL Hints

SQL hints can be passed to a SEM_MATCH query including a line in the following form:

```
PREFIX ORACLE_SEM_HT_NS: <http://oracle.com/semtech#hint>
```

Where *hint* can be any hint supported by SEM_MATCH. For example:

```
PREFIX ORACLE_SEM_HT_NS: <http://oracle.com/semtech#leading(t0,t1)>
SELECT ?book ?title ?isbn
WHERE { ?book <http://title> ?title. ?book <http://ISBN> ?isbn }
```

In this example, `t0,t1` refers to the first and second patterns in the query.

Note the slight difference in specifying hints when compared to SEM_MATCH. Due to restrictions of namespace value syntax, a comma (`,`) must be used to separate `t0` and `t1` (or other hint components) instead of a space.

For more information about using SQL hints, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22), specifically the material about the `HINT0` keyword in the `options` attribute.

8.6.2 Additional WHERE Clause Predicates

The SEM_MATCH `filter` attribute can specify additional selection criteria as a string in the form of a WHERE clause without the WHERE keyword. Additional WHERE clause predicates can be passed to a SEM_MATCH query including a line in the following form:

```
PREFIX ORACLE_SEM_AP_NS: <http://oracle.com/semtech#pred>
```

Where *pred* reflects the WHERE clause content to be appended to the query. For example:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ORACLE_SEM_AP_NS:<http://www.oracle.com/semtech#label$RDFLANG='fr'>
SELECT DISTINCT ?inst ?label
  WHERE { ?inst a <http://someCLass>. ?inst rdfs:label ?label . }
  ORDER BY (?label) LIMIT 20
```

In this example, a restriction is added to the query that the language type of the label variable must be `'fr'`.

8.6.3 Additional Query Options

Additional query options can be passed to a SEM_MATCH query including a line in the following form:

```
PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#option>
```

Where *option* reflects a query option (or multiple query options delimited by commas) to be appended to the query. For example:

```
PREFIX ORACLE_SEM_FS_NS:
<http://oracle.com/semtech#timeout=3,dop=4,INF_ONLY,ORDERED,ALLOW_DUP=T>
SELECT * WHERE {?subject ?property ?object }
```

The following query options are supported:

- `ALLOW_DUP=t` chooses a faster way to query multiple semantic models, although duplicate results may occur.
- `DOP=`*n* specifies the degree of parallelism (*n*) for the query. With multi-core or multi-CPU processors, experimenting with different `DOP` values (such as 4 or 8) may improve performance.
- `INF_ONLY` causes only the inferred model to be queried. (For this option, you should also specify the `includeInferred=true` option for the query.)
- `ORDERED` is translated to a LEADING SQL hint for the query triple pattern joins, while performing the necessary RDF_VALUE$ joins last.
- `QID=`*n* specifies a query ID number; this feature can be used to cancel the query if it is not responding.
- `REWRITE=F` disables ODCI_Table_Rewrite for the SEM_MATCH table function.
- `SKIP_CLOB=T` causes CLOB values not to be returned for the query.
- `TIMEOUT=`*n* (query timeout) specifies the number of seconds (*n*) that the query will run until it is terminated.
- `USE_BIND_VAR(`*n*`)` specifies that a bind variable be used during query parsing for the *n*th constant in triple patterns. (Do not apply this feature to constants inside an OPTIONAL clause.)

8.7 Support for Server-Side APIs

This section describes some of the RDF Semantic Graph features that are exposed by RDF Semantic Graph support for OpenRDF Sesame. For comprehensive documentation of the API calls that support the available features, see the RDF Semantic Graph support for OpenRDF Sesame reference information (Javadoc). For additional information about the server-side features exposed by the support for Sesame, see the relevant chapters in this manual.

8.7.1 Virtual Models Support

Virtual models (explained in Virtual Models (page 1-13)) are specified in the `OracleSailStore` constructor, and they are handled transparently. If a virtual model exists for the model-rulebase combination, it is used in query answering; if such a virtual model does not exist, it is created in the database.

Note:

Virtual model support through RDF Semantic Graph support for OpenRDF Sesame is available only with Oracle Database Release 11.2 or later.

The following example reuses an existing virtual model.

```
public void vm(String jdbcUrl,
                        String user,
                        String password,
                        String model,
                        PrintStream psOut)
{
String m1 = "TEST_1";
```

```
String m2 = "TEST_2";
OraclePool op = new OraclePool(
    OraclePool.getOracleDataSource(jdbcUrl, user, password));

// create the attachment semantic model in case it doesn't exist
// (constructor handles it)
OracleSailStore store = new OracleSailStore(op, m1);
store.shutDown();

store= new OracleSailStore(op, m2);
store.shutDown();

String vmName = "VM_" + m1;

// create a virtual model with name vmName that will be used in the
// OracleSailStore
Oracle oracle = op.getOracle();
oracle.executeCall(
"begin sem_apis.create_virtual_model(?,sem_models(?,?), null); end;",
 vmName, m1, m2);
op.returnOracle(oracle);

 String[] modelNames = {m2};
 String[] rulebaseNames = {};

 Attachment attachment = Attachment.createInstance(modelNames,
 rulebaseNames);

    store = new OracleSailStore(op, m1, attachment, vmName);

OracleSailConnection conn = store.getConnection();
conn.clear();
conn.addStatement(
        store.getValueFactory().createURI("http://a1"),
        store.getValueFactory().createURI("http://p1"),
        store.getValueFactory().createURI("http://b1"));

CloseableIteration res = conn.getStatements(null, null, null, false);
while (res.hasNext()) {
  Statement stmt = (Statement) res.next();
  System.out.println(stmt);
}
conn.close();
store.shutDown();
}
```

You can also use the `OracleSailStore` constructor to create a virtual model, as in the following example:

```
OracleSailStore store = new OracleSailStore(oracle, modelName, attachment, true);
```

In this example, the fourth parameter (`true`) specifies that a virtual model needs to be created for the specified `modelName` and `attachment`.

8.7.2 Connection Pooling Support

Oracle Database Connection Pooling is provided through the RDF Semantic Graph support for OpenRDF Sesame OraclePool class. Usually, OraclePool is initialized with a `DataSource`, using the `OraclePool (DataSource ods)` constructor. In this case, `OraclePool` acts as an extended wrapper for the `DataSource`, while using the

connection pooling capabilities of the data source. When you create an `OracleSailStore` object, it is sufficient to specify the `OraclePool` object in the `store` constructor; the database connections will then be managed automatically by the support for Sesame.

If you need to retrieve Oracle connection objects (which are essentially database connection wrappers) explicitly, you can invoke the `OraclePool.getOracle` method. After finishing with the connection, you can invoke the `OraclePool.returnOracle` method to return the object to the connection pool.

`OraclePool` can also be initialized with an `OracleConnection` object. In this case, only a single database connection is used when referring to this `OraclePool`; and because multiple Sesame `SailConnection` objects will use the same database connection, you must be aware that "dirty" reads across different `SailConnection` objects are possible.

More information about using `OraclePool` can be found in the RDF Semantic Graph support for OpenRDF Sesame API reference information (Javadoc).

The following example sets up an `OracleSailStore` object using the `OraclePool` class, with two initial connections.

```
public void op(String jdbcUrl,
                String user,
                String password,
                String model,
                PrintStream psOut)
  throws Exception
  {
    // test with connection properties
    java.util.Properties prop = new java.util.Properties();
    prop.setProperty("MinLimit", "2");     // the cache size is 2 at least
    prop.setProperty("MaxLimit", "10");
    prop.setProperty("InitialLimit", "2"); // create 2 connections at startup
    prop.setProperty("InactivityTimeout", "1800");    //  seconds
    prop.setProperty("AbandonedConnectionTimeout", "900");  //  seconds
    prop.setProperty("MaxStatementsLimit", "10");
    prop.setProperty("PropertyCheckInterval", "60"); // seconds

    System.out.println("Creating Data Source");
    OracleDataSource ods =
OraclePool.getOracleDataSource(jdbcUrl, user, password,
        prop, "OracleSemConnPool");
    System.out.println("Done creating Data Source");

    OraclePool op = new OraclePool(ods);

    // create an OracleSailStore with the OraclePool
    OracleSailStore store = new OracleSailStore(op, model);
    store.shutDown();
    op.close();
    ods.close();
  }
```

8.7.3 Semantic Model PL/SQL Interfaces

Several semantic PL/SQL subprograms are available through the support for Sesame. Table 8-1 (page 8-22) lists the subprograms and their corresponding Java class and methods.

Table 8-1 PL/SQL Subprograms and Corresponding Support for Sesame Java Class and Methods

PL/SQL Subprogram	Corresponding Java Class and Methods
SEM_APIS.DROP_SEM_MODEL (page 11-39)	OracleUtils.dropSemanticModel
SEM_APIS.MERGE_MODELS (page 11-62)	OracleUtils.mergeModels
SEM_APIS.SWAP_NAMES (page 11-69)	OracleUtils.swapNames
SEM_APIS.REMOVE_DUPLICATES (page 11-65)	OracleUtils.removeDuplicates
SEM_APIS.RENAME_MODEL (page 11-67)	OracleUtils.renameModels

For information about these PL/SQL utility subprograms, see the reference information in SEM_APIS Package Subprograms (page 11-1). For information about the corresponding Java class and methods, see the RDF Semantic Graph support for OpenRDF Sesame API Reference documentation (Javadoc).

8.7.4 Inference Options

You can add options to entailment calls by using the `setInferenceOption(String options)` procedure with the `OracleSailConnection` object. The following example enables incremental inference and specifies a parallelism value of 4 when creating an entailment.

```
conn.setInferenceOption("INC=T,DOP=4");
```

For information about inference options, see Using OWL Inferencing (page 2-5).

8.8 Oracle-Specific Extensions to Sesame APIs

This section describes the extensions that RDF Semantic Graph support for OpenRDF Sesame provides to the Sesame `SailConnection` and Sail APIs.

8.8.1 Statement Uniqueness

In Sesame 2.3.0, statement uniqueness is enforced at the context level: the same triple cannot occur twice in a single context. In the support for Sesame, such uniqueness is enforced through a unique index on the context, predicate, subject, and object. However, in the support for Sesame, statement uniqueness is not enabled by default. When constructing an `OracleSailStore` object, the default value for statement uniqueness is `DONTCARE`, meaning that if such a unique index does not already exist, it will not be created.

You can explicitly enable and disable the requirement for statement uniqueness either through the `OracleSailStore` constructor, or through the `enableUniquenessConstraint` and `disableUniquenessConstraint` methods.

8.8.2 Indexes and Interoperability with the RDF Semantic Graph Support for Apache Jena

Indexes on the application table are critical for good performance when performing DML operations (especially deletes) and named graph queries against an `OracleSailStore` object. When creating a new `OracleSailStore` object, a default `CPSO` index is created (where `C`,`P`,`S`,`O` stands for the context, predicate, subject, and object columns).

There are some storage differences between Oracle semantic models created using the RDF Semantic Graph support for Sesame and Apache Jena. For example, the default application table index is different in the support for Apache Jena. To access data in an RDF Semantic Graph support for Apache Jena model from the RDF Semantic Graph support for OpenRDF Sesame (and vice versa), you can use the `OracleUtils.migrateFromJena` and `OracleUtils.migrateToJena` methods. These methods preserve the semantic data while making changes to the underlying storage structures. In particular, all triples along with their context information are preserved when migrating from RDF Semantic Graph support for OpenRDF Sesame to RDF Semantic Graph support for Apache Jena and back.

Example 8-2 (page 8-23) illustrates migration between models created using the RDF Semantic Graph support for OpenRDF Sesame and Apache Jena:

Example 8-2 Migration Between RDF Semantic Graph Support for Apache Jena and Sesame Models

```
public void migrate(String jdbcUrl,
                        String user,
                        String password,
                        String model,
                        PrintStream psOut)
  throws Exception
  {
    OraclePool oraclePool= new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));

    OracleSailStore store = new OracleSailStore(oraclePool, model);

    OracleSailConnection conn = store.getConnection();
    conn.clear();

    // create a statement and add it to the store
    URI subject =
store.getValueFactory().createURI("http://example.org/foo");
    URI object = store.getValueFactory().createURI("http://example.org/bar");
    URI ngURI = store.getValueFactory().createURI("http://example.org/bar");

    conn.addStatement(subject, RDF.TYPE, object, ngURI);
    conn.commit();

    // convert it to a jena model
    OracleUtils.migrateToJena(oraclePool, model);

    Oracle newOra = oraclePool.getOracle();
    oracle.spatial.rdf.client.jena.Oracle ora =
      new oracle.spatial.rdf.client.jena.Oracle(newOra.getConnection());

    GraphOracleSem graph = new GraphOracleSem(ora, model);
```

```
    // check size of jena graph
    System.out.print("GraphOracleSem size:" + graph.getSize());

    store.shutDown();

    // now convert it back to an Oracle sailstore.
    OracleUtils.migrateFromJena(oraclePool, model);
    conn = store.getConnection();

    // check overall store size
    System.out.println("Size of store :" +conn.size());
    // check context size, should be 1
    System.out.println("Size of context " + ngURI + ":" + conn.size(ngURI));
    store.shutDown();
    ora.dispose();
    oraclePool.returnOracle(newOra);
  }
```

8.8.3 Inference

You can perform inference using the `OracleSailConnection.performInference` method. This method builds an entailment for the `OracleSailStore` object, given the models and rulebases that were specified in the attachment when constructing the store.

For information about creating entailments, see Entailments (Rules Indexes) (page 1-11).

8.8.4 Performing Analysis Operations

It is strongly recommended that you analyze the application table, semantic model, and inferred graph (if it exists) before performing inference and after loading a significant amount of semantic data into the database. Performing the analysis operations causes statistics to be gathered, which will help the Oracle optimizer select efficient execution plans when answering queries.

To gather relevant statistics, you can use the following methods in the `OracleSailConnection` and `OracleSailStore` classes:

- `OracleSailConnection.analyze`
- `OracleSailConnection.analyzeApplicationTable`
- `OracleSailConnection.analyzeInferredGraph`

For information about these methods, including their parameters, see the RDF Semantic Graph support for OpenRDF Sesame reference information (Javadoc).

8.9 Using the Sesame Console with RDF Semantic Graph Support for OpenRDF Sesame

The Sesame Console is, as OpenRDF.org explains, "a command-line application for interacting with Sesame. For now, the best way to create and manage repositories in a SYSTEM repository is to use the Sesame Console. " For basic usage and reference information about the Sesame Console, see the documentation at `http://docs.rdf4j.org/`.

This section describes how to use the Sesame Console with RDF Semantic Graph support for OpenRDF Sesame.

To create a new SAIL repository (`OracleSailStore` object) using the Sesame Console command-line tool, use the following command:

```
create oracle.
```

When you are prompted for parameters, you can specify the following:

Repository ID
ID string associated with the repository. Example: `myOracleRepos`

Repository Title
Title string associated with the repository. Example: `My Oracle Repository`

Model Name
Name of the Oracle semantic model that corresponds to this repository. All DML operations are performed against this model.

Additional Models
Comma-separated list of any additional semantic models to be included in this repository. Any additional models are used only for inference and query operations.

Rulebases
Comma-separated list of rulebases to be considered when building the entailment closure for this repository. The default is no rulebases, which means that no inference will be performed.

Use Virtual Model (TRUE|FALSE) [TRUE]
Specifies whether to use a virtual model when answering queries. The default is `TRUE`, which means to use a virtual model.

Virtual Model Name
If `Use Virtual Model` is `TRUE`, specify the name of the virtual model to be used to answer queries, if the database user that owns the repository has read permission on the specified virtual model.

Inference Maintenance Mode (NO_UPDATE|UPDATE_WHEN_COMMIT) [NO_UPDATE]
Specifies whether to update the entailment closure on each commit operation (`UPDATE_WHEN_COMMIT`) or only when the `OracleSailConnection.performInference` method is invoked (`NO_UPDATE`). The default is `NO_UPDATE`.

Allow Duplicates in Query (TRUE|FALSE) [TRUE]
Specifies whether, when querying a repository that has additional models, the query response can include duplicates across the different models. The default is `TRUE`, allows duplicates to be included, and which speeds query performance.

Allow Query with Non-Valid Inference status (INVALID|VALID|INCOMPLETE) [INVALID]
Specifies whether to allow queries when the entailment is not updated, that is, when the inference status is not valid. The default value is `INVALID`, which means that the entailment status can be `INVALID` when querying.

To allow queries only when entailments have a valid status, specify `VALID`. To allow queries when entailments have either a valid or incomplete status, specify `INCOMPLETE`.

Application Table Indexes [CPSO]
Specifies the custom index to be created on the application table for this repository. You can specify the columns to be included (and their order) by using a string of up to 4 of the following characters: `C` (*context*), `P` (*property*), `S` (*subject*), `O` (*object*). (You can use only those characters and cannot repeat any characters.) The default is `CPSO`, which creates an index on all four columns with context as the leading column.

Index Option: Degree of Parallelism [1]
Specifies the DOP (degree of parallelism) value for the application table indexes. With multi-core or multi-CPU processors, experimenting with different `DOP` values (such as 4 or 8) may improve performance. The default is 1 (no parallelism).

Index Option: Prefix Length to Compress [2]
Specifies whether to use key compression on the application table indexes. The default is 2, which specifies compression on the first two columns. A value of 0 (zero) means that no compression will be used.

Enforce Uniqueness (FALSE|TRUE) []
Specifies whether to enforce statement uniqueness across multiple contexts in the repository. Note that statement uniqueness is enforced by a unique index on the application table. The default is [] (that is, no value specified), which means that if the model exists, the uniqueness setting will be inherited from that model; and if the model does not already exist, a non-unique index will be created.

If you specify FALSE and if a unique index exists, that index is dropped and it is re-created as a non-unique index.

Data Source Name [OracleSemDS]
Name of the JDBC data source that provides the database connection setting for accessing the Oracle semantic models and application tables for this repository. The default is `OracleSemDS`.

Example 8-3 (page 8-26) shows a sample scenario of using the Sesame Console.

Example 8-3 Using the Sesame Console

```
> connect http://localhost:8080/openrdf-sesame.
Disconnecting from default data directory
Connected to http://localhost:8080/openrdf-sesame
> create oracle.
Please specify values for the following variables:
Repository ID: model2Repos
Repository title: Repository for Model 2
Model Name: model2
Additional Models (comma separated):
Rulebases (comma separated): OWLPRIME    -- OWLPRIME inference will be performed
when this model is created
Use Virtual Model (TRUE|FALSE) [TRUE]:  TRUE
Virtual Model Name:     -- a new virtual model will be created if nonexistent
Inference Maintenance Mode (UPDATE_WHEN_COMMIT|NO_UPDATE) [UPDATE_WHEN_COMMIT]:
NO_UPDATE
Allow duplicates in query (TRUE|FALSE) [TRUE]: TRUE
Allow query with non-valid inference status (NONE|INCOMPLETE|INVALID) [NONE]: NONE -
inference must be up to date before query
Application Table Index [CPSO]: CPSO      -- default, recommended index
Index option: Degree of parallelism [1]: 1
Index option: Leading columns to compress [2]: 2
Enforce Uniqueness (|FALSE|TRUE) []: FALSE
DataSource Name [OracleSemDS]: OracleSemDS
```

```
Repository created
> open model2Repos.
Opened repository 'model2Repos'
model2Repos> show c.
--no contexts found--
model2Repos>
```

8.10 Example Queries Using RDF Semantic Graph support for OpenRDF Sesame

This section includes example queries using RDF Semantic Graph support for OpenRDF Sesame. Each example is self-contained: it typically starts an `OraclePool` object, creates an `OracleSailStore` object, adds and perhaps removes statements, performs a query that may involve inference, displays the result, shuts down the `OracleSailStore` object, and closes the `OraclePool` object.

For these examples, the following libraries must be included in the CLASSPATH definition:

```
sesame-console-2.3.1.jar
sesame-http-client-2.3.1.jar
sesame-http-protocol-2.3.1.jar
sesame-http-server-spring-2.3.1.jar
sesame-model-2.3.1.jar
sesame-query-2.3.1.jar
sesame-queryalgebra-evaluation-2.3.1.jar
sesame-queryalgebra-model-2.3.1.jar
sesame-queryparser-api-2.3.1.jar
sesame-queryparser-serql-2.3.1.jar
sesame-queryparser-sparql-2.3.1.jar
sesame-queryresultio-api-2.3.1.jar
sesame-queryresultio-binary-2.3.1.jar
sesame-queryresultio-sparqljson-2.3.1.jar
sesame-queryresultio-sparqlxml-2.3.1.jar
sesame-queryresultio-text-2.3.1.jar
sesame-repository-api-2.3.1.jar
sesame-repository-contextaware-2.3.1.jar
sesame-repository-dataset-2.3.1.jar
sesame-repository-event-2.3.1.jar
sesame-repository-http-2.3.1.jar
sesame-repository-manager-2.3.1.jar
sesame-repository-sail-2.3.1.jar
sesame-rio-api-2.3.1.jar
sesame-rio-n3-2.3.1.jar
sesame-rio-ntriples-2.3.1.jar
sesame-rio-rdfxml-2.3.1.jar
sesame-rio-trig-2.3.1.jar
sesame-rio-trix-2.3.1.jar
sesame-rio-turtle-2.3.1.jar
sesame-runtime-2.3.1.jar
sesame-sail-api-2.3.1.jar
sesame-sail-inferencer-2.3.1.jar
sesame-sail-memory-2.3.1.jar
sesame-sail-nativerdf-2.3.1.jar
sesame-sail-rdbms-2.3.1.jar
sdordf.jar
sdordfsesame.jar
ojdbc6.jar
```

To simplify the examples, an environment variable named `CP` was defined, as follows:

```
setenv CP ./:sesame-console-2.3.1.jar:sesame-http-client-2.3.1.jar:sesame-http-
protocol-2.3.1.jar:sesame-http-server-spring-2.3.1.jar:sesame-model-2.3.1.jar:sesame-
query-2.3.1.jar:sesame-queryalgebra-evaluation-2.3.1.jar:sesame-queryalgebra-
model-2.3.1.jar:sesame-queryparser-api-2.3.1.jar:sesame-queryparser-
serql-2.3.1.jar:sesame-queryparser-sparql-2.3.1.jar:sesame-queryresultio-
api-2.3.1.jar:sesame-queryresultio-binary-2.3.1.jar:sesame-queryresultio-
sparqljson-2.3.1.jar:sesame-queryresultio-sparqlxml-2.3.1.jar:sesame-queryresultio-
text-2.3.1.jar:sesame-repository-api-2.3.1.jar:sesame-repository-
contextaware-2.3.1.jar:sesame-repository-dataset-2.3.1.jar:sesame-repository-
event-2.3.1.jar:sesame-repository-http-2.3.1.jar:sesame-repository-
manager-2.3.1.jar:sesame-repository-sail-2.3.1.jar:sesame-rio-api-2.3.1.jar:sesame-
rio-n3-2.3.1.jar:sesame-rio-ntriples-2.3.1.jar:sesame-rio-rdfxml-2.3.1.jar:sesame-
rio-trig-2.3.1.jar:sesame-rio-trix-2.3.1.jar:sesame-rio-turtle-2.3.1.jar:sesame-
runtime-2.3.1.jar:sesame-sail-api-2.3.1.jar:sesame-sail-inferencer-2.3.1.jar:sesame-
sail-memory-2.3.1.jar:sesame-sail-nativerdf-2.3.1.jar:sesame-sail-
rdbms-2.3.1.jar:sdordf.jar:sdordfsesame.jar:ojdbc6.jar
```

Note:

Enter the `setenv` command, as well as each `javac` and `java` command, on a single command line.

To run a query, you must do the following:

1. Include the code in a Java source file. The examples used in this section are supplied in files in the `examples` directory of the RDF Semantic Graph support for OpenRDF Sesame download.
2. Compile the Java source file. For example:

   ```
   > javac -classpath $CP Example1.java
   ```

3. Run the compiled file. For example:

   ```
   > java -classpath $CP Test jdbc:oracle:thin:@localhost:1521:orcl scott <password-
   for-scott> M1
   ```

8.10.1 Example1.java: Basic Operations

Example 8-4 (page 8-28) shows the `Example1.java` file, which performs some basic operations.

Example 8-4 Basic Operations

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;

import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;

public class Example1
{
```

```
  public static void main(String[] args) throws SQLException, SailException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);

    OracleSailConnection conn = store.getConnection();

    ValueFactory vf = store.getValueFactory();
    URI p = vf.createURI("http://p");
    URI cls = vf.createURI("http://cls");
    URI a = vf.createURI("http://a");
    URI b = vf.createURI("http://b");
    URI ng1 = vf.createURI("http://ng1");

    conn.clear();
    conn.addStatement(p, RDFS.DOMAIN, cls);
    conn.addStatement(a, p, b, ng1);

    psOut.println("size of context " + ng1 + ":" + conn.size(ng1));

    // returns OracleStatements
    CloseableIteration<? extends Statement, SailException> it =
      conn.getStatements(null, null, null, false);

    while (it.hasNext()) {
      Statement stmt = it.next();
      psOut.println("getStatements: stmt: " + stmt.toString());
    }

    conn.removeStatements(null, null, null, ng1);

    psOut.println("size of context " + ng1 + ":" + conn.size(ng1));

    conn.removeAll();

    psOut.println("size of store: " + conn.size());

    conn.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
/usr/local/packages/jdk16/bin/javac -classpath $CP Example1.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example1
jdbc:oracle:thin:@localhost:1521:ORCL scott <password-for-scott> TestModel
```

The expected output of the `java` command might appear as follows:

```
size of context http://ng1:1
getStatements: stmt:  0: (http://a, http://p, http://b) [http://ng1]
getStatements: stmt:  0: (http://p, http://www.w3.org/2000/01/rdf-schema#domain,
http://cls) [null]
size of context http://ng1:0
size of store: 0
```

8.10.2 Example2.java: Add a Data File (in TRIG format)

Example 8-5 (page 8-30) shows the `Example2.java` file, which adds a data file in TRIG format.

Example 8-5 Add a Data File (in TRIG format)

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;

public class Example2
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];
    String trigFile = args[4];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);
    SailRepository sr = new SailRepository(store);
    RepositoryConnection repConn = sr.getConnection();

    repConn.setAutoCommit(false);
    repConn.add(new File(trigFile), "http://my.com/", RDFFormat.TRIG);
    repConn.commit();

    psOut.println("size " + Long.toString(repConn.size()));

    repConn.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example2.java
```

For running this example, assume that a sample TRIG data file named `test.trig` was created as:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix swp: <http://www.w3.org/2004/03/trix/swp-1/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.org/> .
@prefix : <http://example.org/> .

# default graph
{
<http://example.org/bob>    dc:publisher  "Bob Hacker" .
<http://example.org/alice>  dc:publisher  "Alice Hacker" .
}

:bob {
_:a foaf:mbox <mailto:bob@oldcorp.example.org> .
}

:alice {
_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@work.example.org> .
}

:jack {
_:a foaf:name "Jack" .
_:a foaf:mbox <mailto:jack@oracle.example.org> .
}
```

To run this example using the `test.trig` data file, enter the following command:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example2
jdbc:oracle:thin:@localhost:1521:ORCL scott  <password>  TestModel ./test.trig
```

The expected output of the `java` command might appear as follows:

```
size 7
```

8.10.3 Example3.java: Simple Query

Example 8-6 (page 8-31) shows the `Example3.java` file, which performs a simple query.

Example 8-6 Simple Query

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
```

```
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example3
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);
    SailRepository myRepository = new SailRepository(store);

    ValueFactory f = myRepository.getValueFactory();
    RepositoryConnection conn = myRepository.getConnection();

    // create some resources and literals to make statements out of
    URI alice = f.createURI("http://example.org/people/alice");
    URI name = f.createURI("http://example.org/ontology/name");
    URI person = f.createURI("http://example.org/ontology/Person");
    Literal alicesName = f.createLiteral("Alice");

    conn.clear(); // to start from scratch
    conn.add(alice, RDF.TYPE, person);
    conn.add(alice, name, alicesName);
    conn.commit();

    store.analyze();
    store.analyzeApplicationTable();

    try {
      //run a query against the repository
      String queryString = " SELECT * WHERE {?x ?p ?y} LIMIT 1 ";
      TupleQuery tupleQuery = conn.prepareTupleQuery(QueryLanguage.SPARQL,
queryString);
      TupleQueryResult result = tupleQuery.evaluate();
      try {
        while (result.hasNext()) {
          BindingSet bindingSet = result.next();
          psOut.println("value of x: " + bindingSet.getValue("x"));
        }
      }
```

```
      finally {
        result.close();
      }
    }
    finally {
      conn.clear();
      if (conn != null && conn.isOpen()) {
        conn.close();
      }

      myRepository.shutDown();
      op.close();
    }
  }
}
```

To compile this example, enter the following command:

```
/usr/local/packages/jdk16/bin/javac -classpath $CP Example3.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example3
jdbc:oracle:thin:@localhost:1521:ORCL scott <password-for-scott> TestModel
```

The expected output of the `java` command might appear as follows:

```
value of x: http://example.org/people/alice
```

8.10.4 Example4.java: Simple Bulk Load

Example 8-7 (page 8-33) shows the `Example4.java` file, which performs a simple bulk load operation.

Example 8-7 Simple Bulk Load

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import org.openrdf.model.Resource;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;
```

```
public class Example4
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];
    String filename = args[4]; // N-TRIPLES file

    OraclePool op = new OraclePool(OraclePool.getOracleDataSource(jdbcUrl, user,
password));

    psOut.println("testBulkLoad: start");

    OracleSailStore store = new OracleSailStore(op, model);
    OracleSailConnection osc = store.getConnection();
    FileInputStream fis = new FileInputStream(filename);

    long loadBegin = System.currentTimeMillis();
    osc.getBulkUpdateHandler().addInBulk(
        fis,
        "http://abc",               // baseURI
        RDFFormat.NTRIPLES,         // dataFormat
        null,                       // tablespaceName
        null,                       // flags
        null,                       // StatusListener
        (Resource[]) null           // Resource... for contexts
        );

    long loadEnd = System.currentTimeMillis();
    psOut.println("testBulkLoad: " + (loadEnd - loadBegin) + "ms.\n");

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example4.java
```

For running this example, assume that a sample ntriples data file named `test.ntriples` was created as:

```
<http://example.org/bob>    <http://purl.org/dc/elements/1.1/publisher>  "Bob
Hacker" .
<http://example.org/alice>  <http://purl.org/dc/elements/1.1/publisher>  "Alice
Hacker" .
```

To run this example using the `test.ntriples` file, enter the following command:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example4
jdbc:oracle:thin:@localhost:1521:ORCL scott  <password>  TestModel ./test.ntriples
```

The expected output of the `java` command might appear as follows:

```
testBulkLoad: start
testBulkLoad: 756ms
```

8.10.5 Example5.java: Bulk Load RDF/XML and Application Table Index Maintenance

Example 8-8 (page 8-35) shows the `Example5.java` file, which disables indexes on the application table, performs a bulk load operation from an RDF file in XML format, and re-enables the indexes.

Example 8-8 Bulk Load RDF/XML and Application Table Index Maintenance

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import org.openrdf.model.Resource;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example5
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];
    String filename = args[4]; // RDF/XML

    OraclePool op = new OraclePool(OraclePool.getOracleDataSource(jdbcUrl, user,
password));

    psOut.println("testBulkLoad: start");

    OracleSailStore store = new OracleSailStore(op, model);
    OracleSailConnection osc = store.getConnection();

    // disable indexes on application table
    store.disableAllAppTabIndexes();
```

```
    // NOTE: can be a gzipped file!
    FileInputStream fis = new FileInputStream(filename);

    long loadBegin = System.currentTimeMillis();
    osc.getBulkUpdateHandler().addInBulk(
        fis,
        "http://abc",               // baseURI
        RDFFormat.RDFXML,           // dataFormat
        null,                       // tablespaceName
        null,                       // flags
        null,                       // StatusListener
        (Resource[]) null           // Resource... for contexts
        );

    long loadEnd = System.currentTimeMillis();
    psOut.println("testBulkLoad: " + (loadEnd - loadBegin) + "ms.\n");

    // enable indexes on application table
    // Note: one can also specify to rebuild indexes in parallel.
    store.enableAllAppTabIndexes();

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example5.java
```

For running this example, assume that a sample file named `test.rdfxml` was created as:

```
<?xml version="1.0"?>
<!DOCTYPE owl [
     <!ENTITY owl  "http://www.w3.org/2002/07/owl#" >
     <!ENTITY xsd  "http://www.w3.org/2001/XMLSchema#" >
   ]>

<rdf:RDF
  xmlns     = "http://a/b#" xml:base  = "http://a/b#" xmlns:my  = "http://a/b#"
  xmlns:owl = "http://www.w3.org/2002/07/owl#"
  xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs= "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema#">
  <owl:Class rdf:ID="Color">
    <owl:oneOf rdf:parseType="Collection">
      <owl:Thing rdf:ID="Red"/>
      <owl:Thing rdf:ID="Blue"/>
    </owl:oneOf>
  </owl:Class>
</rdf:RDF>
```

To run this example using the `test.rdfxml` file, enter the following command:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example5
jdbc:oracle:thin:@localhost:1521:ORCL scott  <password>  EX5 ./test.rdfxml
```

The expected output of the `java` command might appear as follows:

```
testBulkLoad: start
testBulkLoad: 825ms
```

8.10.6 Example6.java: Bulk Load With StatusListener to Handle Loading with Bad Data

Example 8-9 (page 8-37) shows the `Example6.java` file, which performs the bulk load and index maintenance operations shown in Example 8-7 (page 8-33), but also registers and implements a `StatusListener` class to check for and report any data errors.

Example 8-9 Bulk Load With StatusListener to Handle Loading with Bad Data

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import org.openrdf.model.Resource;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.model.Statement;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example6
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];
    String filename = args[4]; // RDF/XML

    OraclePool op = new OraclePool(OraclePool.getOracleDataSource(jdbcUrl, user,
password));

    psOut.println("testBulkLoad: start");

    OracleSailStore store = new OracleSailStore(op, model);
    OracleSailConnection osc = store.getConnection();
```

```
    // disable indexes on application table
    store.disableAllAppTabIndexes();

    // NOTE: can be a gzipped file!
    FileInputStream fis = new FileInputStream(filename);

    long loadBegin = System.currentTimeMillis();
    osc.getBulkUpdateHandler().addInBulk(
        fis,
        "http://abc",               // baseURI
        RDFFormat.NTRIPLES,         // dataFormat
        null,                       // tablespaceName
        null,                       // flags
        new MyListener(psOut),      // register a StatusListener
        (Resource[]) null           // Resource... for contexts
        );

    long loadEnd = System.currentTimeMillis();
    psOut.println("testBulkLoad: " + (loadEnd - loadBegin) + "ms.\n");

    // enable indexes on application table
    // Note: one can also specify to rebuild indexes in parallel.
    store.enableAllAppTabIndexes();

    osc.close();
    store.shutDown();
    op.close();
  }

  static class MyListener implements StatusListener {
    PrintStream m_ps = null;
    public MyListener(PrintStream ps) { m_ps = ps; }

    public void statusChanged(long count)
    {
      m_ps.println("process to " + Long.toString(count));
    }

    public int illegalStmtEncountered(Statement statement, long count)
    {
      m_ps.println("hit illegal statement with object " +
statement.getObject().toString());
      return 0; // skip it
    }
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example6.java
```

For running this example, assume that a sample N-TRIPLES file named `test-include-badtriple.ntriples` was created as follows. In this file, the first and last triples have illegal typed literal values for objects.

```
<http://example.org/x>      <http://my.com/#age> "123.3456"^^<http://www.w3.org/2001/
XMLSchema#integer> .
<http://example.org/bob>    <http://purl.org/dc/elements/1.1/publisher>  "Bob
Hacker" .
<http://example.org/alice>  <http://purl.org/dc/elements/1.1/publisher>  "Alice
Hacker" .
```

```
<http://example.org/y>      <http://my.com/#age> "hello"^^<http://www.w3.org/2001/
XMLSchema#float> .
```

To run this example using the `test-include-badtriple.ntriples` file, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example6
jdbc:oracle:thin:@localhost:1521:ORCL scott  <password>  EX6 ./test-include-
badtriple.ntriples
```

The expected output of the `java` command might appear as follows:

```
testBulkLoad: start
hit illegal statement with object "123.3456"^^<http://www.w3.org/2001/
XMLSchema#integer>
process to 2
process to 3
hit illegal statement with object "hello"^^<http://www.w3.org/2001/XMLSchema#float>
testBulkLoad: 725ms.
```

8.10.7 Example7.java: Load Data from Sesame Store into Oracle Database

Example 8-10 (page 8-39) shows the `Example7.java` file, disables indexes on the application table, loads an RDF file in XML format into an in-memory Sesame store, loads the data into the Oracle database, and re-enables the indexes.

Example 8-10 Load Data from Sesame Store into Oracle Database

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import org.openrdf.model.Resource;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.sail.memory.MemoryStore;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.model.Statement;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example7
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
```

```
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];
    String filename = args[4]; // RDF/XML

    OraclePool op = new OraclePool(OraclePool.getOracleDataSource(jdbcUrl, user,
password));

    psOut.println("testBulkLoad: start");

    SailRepository srInMem = null;
    RepositoryConnection repConnInMem = null;
    { // build an in memory Sesame store for testing purpose
      srInMem = new SailRepository(new MemoryStore());
      srInMem.initialize();
      repConnInMem = srInMem.getConnection();
      File file = new File(filename);
      repConnInMem.setAutoCommit(false);
      repConnInMem.add(file, "http://my.com", RDFFormat.RDFXML);
      repConnInMem.commit();
    }

    OracleSailStore store = new OracleSailStore(op, model);
    OracleSailConnection osc = store.getConnection();

    // disable indexes on application table
    store.disableAllAppTabIndexes();

    long loadBegin = System.currentTimeMillis();
    // load all statements from in memory store to Oracle
    osc.getBulkUpdateHandler().addInBulk(
        repConnInMem.getStatements(null, null, null, false),
        null                       // tablespaceName
        );

    long loadEnd = System.currentTimeMillis();
    psOut.println("testBulkLoad: " + (loadEnd - loadBegin) + "ms.\n");

    // enable indexes on application table
    // Note: one can also specify to rebuild indexes in parallel.
    store.enableAllAppTabIndexes();

    repConnInMem.close();
    srInMem.shutDown();

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example7.java
```

For running this example, assume that a file (to be loaded into memory) with the same format and content as the file described in Example5.java: Bulk Load RDF/XML and Application Table Index Maintenance (page 8-35) has been created.

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example7
jdbc:oracle:thin:@localhost:1521:ORCL scott  <password>  EX7 ./test.rdfxml
```

The expected output of the `java` command might appear as follows:

```
testBulkLoad: start
testBulkLoad: 720ms
```

8.10.8 Example8.java: SPARQL ASK Query

Example 8-11 (page 8-41) shows the `Example8.java` file, which performs a SPARQL ASK query.

Example 8-11 SPARQL ASK Query

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.BooleanQuery;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example8
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);

    SailRepository sr = new SailRepository(store);
    RepositoryConnection repConn = sr.getConnection();
```

```
    ValueFactory vf = sr.getValueFactory();
    URI p   = vf.createURI("http://p");
    URI cls = vf.createURI("http://cls");

    repConn.clear();
    repConn.add(p, RDFS.DOMAIN, cls);
    repConn.commit();

    store.analyze();                // analyze the semantic model
    store.analyzeApplicationTable(); // and then the application table

    BooleanQuery tq = null;
    tq = repConn.prepareBooleanQuery(QueryLanguage.SPARQL, "ASK { ?x ?p <http://
cls> }" );
    boolean b = tq.evaluate();
    psOut.print("\nAnswer is " + Boolean.toString(b));

    repConn.close();
    sr.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example8.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example4
jdbc:oracle:thin:@localhost:1521:ORCL scott  <password>  EX8
```

The expected output of the `java` command might appear as follows:

```
Answer is true
```

8.10.9 Example9.java: SPARQL CONSTRUCT and DESCRIBE

Example 8-12 (page 8-42) shows the `Example9.java` file, which performs SPARQL CONSTRUCT and DESCRIBE queries.

Example 8-12 SPARQL CONSTRUCT and DESCRIBE

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
```

```
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.BooleanQuery;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.GraphQuery;
import org.openrdf.query.GraphQueryResult;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example9
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);

    SailRepository sr = new SailRepository(store);
    RepositoryConnection repConn = sr.getConnection();

    ValueFactory vf = sr.getValueFactory();
    URI p   = vf.createURI("http://p");
    URI cls = vf.createURI("http://cls");

    repConn.clear();
    repConn.add(p, RDFS.DOMAIN, cls);
    repConn.commit();

    store.analyze();                // analyze the semantic model
    store.analyzeApplicationTable(); // and then the application table

    GraphQuery tq = null;
    tq = repConn.prepareGraphQuery(QueryLanguage.SPARQL,
        "CONSTRUCT {?x <http://new_eq_p> ?o } WHERE { ?x ?p ?o }" );
    {
      psOut.println("Start construct query");
      GraphQueryResult result = tq.evaluate();
      while (result.hasNext()) {
        Statement stmt = (Statement) result.next();
        psOut.println(stmt.toString());
        // do something interesting with the values here...
      }
      result.close();
    }

    tq = repConn.prepareGraphQuery(QueryLanguage.SPARQL,
        "DESCRIBE <http://p> ");
    {
      psOut.println("Start describe query");
      GraphQueryResult result = tq.evaluate();
      while (result.hasNext()) {
```

```
        Statement stmt = (Statement) result.next();
        psOut.println(stmt.toString());
        // do something interesting with the values here...
      }
      result.close();
    }

    repConn.close();
    sr.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example9.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example9
jdbc:oracle:thin:@localhost:1521:ORCL scott <password> EX9
```

The expected output of the `java` command might appear as follows:

```
Start construct query
(http://p, http://new_eq_p, http://cls)
Start describe query
(http://p, http://www.w3.org/2000/01/rdf-schema#domain, http://cls)
```

8.10.10 Example10.java: Inference

Example 8-13 (page 8-44) shows the `Example10.java` file, which performs OWLPrime inference using the incremental (`INC`) and degree of parallelism (`DOP`) options.

Example 8-13 Inference

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.BooleanQuery;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.GraphQuery;
import org.openrdf.query.GraphQueryResult;
import org.openrdf.query.TupleQueryResult;
```

```
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example10
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    String[] rulebaseNames = new String[1];
    rulebaseNames[0] = "owlprime";

    Attachment attachment = Attachment.createInstance(
        Attachment.NO_ADDITIONAL_MODELS, rulebaseNames,
        InferenceMaintenanceMode.NO_UPDATE,
        QueryOptions.DEFAULT);

    OracleSailStore store = new OracleSailStore(op, model, attachment);
    OracleSailConnection osc = store.getConnection();

    ValueFactory vf = osc.getValueFactory();

    URI sub, pred, obj;
    {
      sub = vf.createURI("http://C1");
      pred = vf.createURI("http://www.w3.org/2000/01/rdf-schema#subClassOf");
      obj  = vf.createURI("http://C2");
      osc.addStatement(sub, pred, obj);

      sub = vf.createURI("http://C2");
      pred = vf.createURI("http://www.w3.org/2000/01/rdf-schema#subClassOf");
      obj  = vf.createURI("http://C3");
      osc.addStatement(sub, pred, obj);
    }

    osc.commit();
    osc.analyze();                 // analyze the semantic model
    osc.analyzeApplicationTable(); // and then the application table

    // parallel inference is certainly an overkill for this ontology
    osc.setInferenceOption("INC=T,DOP=4,RAW8=T");
    osc.performInference();
    osc.analyzeInferredGraph();

    TupleQuery tq = null;
    RepositoryConnection repConn = osc.asRepositoryConnection();
    tq = repConn.prepareTupleQuery(QueryLanguage.SPARQL,
        "SELECT ?s ?p ?o WHERE {?s ?p ?o } ");
    {
      TupleQueryResult result = tq.evaluate();
      int idx = 0;
      try {
```

```
        psOut.print("\nStart printing solution\n");
        while (result.hasNext()) {
          idx++;
          BindingSet bindingSet = result.next();
        }
      }
      finally {
        psOut.println("\ntotal # of solution " + Integer.toString(idx));
        result.close();
      }
    }


    {
      sub  = vf.createURI("http://C3");
      pred = vf.createURI("http://www.w3.org/2000/01/rdf-schema#subClassOf");
      obj  = vf.createURI("http://C4");
      osc.addStatement(sub, pred, obj);
    }

    // make a small change and then perform inference again
    osc.commit();
    osc.analyze();                 // analyze the semantic model
    osc.analyzeApplicationTable(); // and then the application table
    osc.performInference();
    osc.analyzeInferredGraph();

    {
      TupleQueryResult result = tq.evaluate();
      int idx = 0;
      try {
        psOut.print("\nStart printing solution\n");
        while (result.hasNext()) {
          idx++;
          BindingSet bindingSet = result.next();
        }
      }
      finally {
        psOut.println("\ntotal # of solution " + Integer.toString(idx));
        result.close();
      }
    }

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example10.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example10
jdbc:oracle:thin:@localhost:1521:ORCL scott <password> EX10
```

The expected output of the `java` command might appear as follows:

```
Start printing solution
```

```
total # of solution 3

Start printing solution

total # of solution 6
```

8.10.11 Example11.java: Named Graph Query

Example 8-14 (page 8-47) shows the `Example9.java` file, which performs a named graph query.

Example 8-14 Named Graph Query

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.BooleanQuery;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.GraphQuery;
import org.openrdf.query.GraphQueryResult;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example11
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];
    String trigFile = args[4];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));

    OracleSailStore store = new OracleSailStore(op, model);
    OracleSailConnection osc = store.getConnection();
    RepositoryConnection repConn = osc.asRepositoryConnection();
```

```
    repConn.setAutoCommit(false);

    // load the data incrementally since it is very small file
    repConn.add(new File(trigFile), "http://my.com/", RDFFormat.TRIG);

    osc.commit();
    osc.analyze();                 // analyze the semantic model
    osc.analyzeApplicationTable(); // and then the application table

    TupleQuery tq = null;
    tq = repConn.prepareTupleQuery(QueryLanguage.SPARQL,
        "SELECT ?g ?s ?p ?o WHERE {?g <http://purl.org/dc/elements/1.1/publisher> ?
o1 . GRAPH ?g {?s ?p ?o}}");
    {
      TupleQueryResult result = tq.evaluate();
      int idx = 0;
      try {
        while (result.hasNext()) {
          idx++;
          BindingSet bindingSet = result.next();
          psOut.print("\nsolution " + bindingSet.toString());
        }
      }
      finally {
        psOut.println("\ntotal # of solution " + Integer.toString(idx));
        result.close();
      }
    }

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example9.java
```

For running this example, assume that the `test.trig` file described in Example2.java: Add a Data File (in TRIG format) (page 8-30) has been created.

To run this example using the `test.trig` data file, enter the following command:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example11
jdbc:oracle:thin:@localhost:1521:ORCL scott  <password>  EX11 ./test.trig
```

The expected output of the `java` command might appear as follows:

```
solution [g=http://example.org/alice;s=_:node14r2238h1x1;p=http://xmlns.com/foaf/0.1/
mbox;o=mailto:alice@work.example.org]
solution [g=http://example.org/alice;s=_:node14r2238h1x1;p=http://xmlns.com/foaf/0.1/
name;o="Alice"]
solution [g=http://example.org/bob;s=_:node14r2238h1x1;p=http://xmlns.com/foaf/0.1/
mbox;o=mailto:bob@oldcorp.example.org]
total # of solution 3
```

8.10.12 Example12.java: Indexes on Application Table *[Advanced]*

Example 8-15 (page 8-49) shows the `Example12.java` file, an advanced example that creates indexes on the application table.

Example 8-15 Indexes on Application Table [Advanced]

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.BooleanQuery;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.GraphQuery;
import org.openrdf.query.GraphQueryResult;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example12
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));

    OracleSailStore store = new OracleSailStore(op, model);

    // create an index with compress 2, parallel 4
    store.createAppTabIndex("spoc", 2, 4);

    // create another index w/o parallel or compression
    store.createAppTabIndex("cspo", 0, 1);

    // will print out the default index (cpso) as well
    psOut.println("index signatures before dropping indexes:");
    for (String sig : store.getAppTabIndexSignatures()) {
      psOut.println("index signature:" + sig);
    }

    store.dropAllAppTabIndexes();
```

```
    psOut.println("index signatures after dropping indexes...");
    for (String sig : store.getAppTabIndexSignatures()) {
      psOut.println("index signature:" + sig);
    }

    store.shutDown();

    // clean up oracle tables and models
    OracleUtils.dropSemanticModelAndTables(op, model);

    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example12.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example12
jdbc:oracle:thin:@localhost:1521:ORCL scott <password> EX12
```

The expected output of the `java` command might appear as follows:

```
index signatures before dropping indexes:
index signature:spoc
index signature:cpso
index signature:cspo
index signatures after dropping indexes...
```

8.10.13 Example13.java: Uniqueness Constraint on Application Table *[Advanced]*

Example 8-16 (page 8-50) shows the `Example13.java` file, an advanced example that enables and disables the uniqueness constraint on the application table.

Example 8-16 Uniqueness Constraint on Application Table [Advanced]

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.BooleanQuery;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.GraphQuery;
```

```
import org.openrdf.query.GraphQueryResult;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example13
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));

    OracleSailStore store = new OracleSailStore(op, model);
    store.enableUniquenessConstraint();    ValueFactory f = store.getValueFactory();

    OracleSailConnection osc = store.getConnection();

    // create some resources and literals to make statements out of
    URI alice = f.createURI("http://example.org/people/alice");
    URI name = f.createURI("http://example.org/ontology/name");
    URI person = f.createURI("http://example.org/ontology/Person");

    osc.addStatement(alice, name, person);
    osc.addStatement(alice, name, person);
    psOut.println("size of store after adding duplicate triple with uniqueness ON :"
+ osc.size());

    osc.clear();
    store.disableUniquenessConstraint();    osc.addStatement(alice, name, person);
    osc.addStatement(alice, name, person);
    psOut.println("size of store after adding duplicate triple with uniqueness OFF:"
+ osc.size());

    osc.close();

    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example13.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example13
jdbc:oracle:thin:@localhost:1521:ORCL scott <password> EX13
```

The expected output of the `java` command might appear as follows:

```
size of store after adding duplicate triple with uniqueness ON :1
size of store after adding duplicate triple with uniqueness OFF:2
```

8.10.14 Example14.java: Query Timeout and Parallel Execution *[Advanced]*

Example 8-17 (page 8-52) shows the `Example14.java` file, an advanced example that specifies a query timeout and parallel execution (`TIMEOUT` and `DOP` options).

Example 8-17 Query Timeout and Parallel Execution [Advanced]

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example14
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);

    OracleSailConnection osc = store.getConnection();
    ValueFactory f = osc.getValueFactory();

    // create some resources and literals to make statements out of
    URI alice = f.createURI("http://example.org/people/alice");
    URI name = f.createURI("http://example.org/ontology/name");
    URI person = f.createURI("http://example.org/ontology/Person");
    Literal alicesName = f.createLiteral("Alice");

    osc.addStatement(alice, RDF.TYPE, person);
    osc.addStatement(alice, name, alicesName);
```

```
    osc.commit();

    osc.analyze();
    osc.analyzeApplicationTable();

    // Run a query while specifying parallel execution (through DOP=2)
    // and max query execution time (through tmieout=3)
    String queryString =
      " PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#dop=2,timeout=3> " +
      " SELECT * WHERE {?x ?p ?y} ";

    TupleQuery tupleQuery = osc.asRepositoryConnection().prepareTupleQuery(
        QueryLanguage.SPARQL, queryString);

    TupleQueryResult result = tupleQuery.evaluate();
    try {
      while (result.hasNext()) {
        BindingSet bindingSet = result.next();
        psOut.println("value of x: " + bindingSet.getValue("x"));
      }
    }
    finally {
      result.close();
    }

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example14.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example14
jdbc:oracle:thin:@localhost:1521:ORCL scott <password> EX14
```

The expected output of the `java` command might appear as follows:

```
value of x: http://example.org/people/alice
value of x: http://example.org/people/alice
```

8.10.15 Example15.java: Get COUNT of Matches *[Advanced]*

Example 8-18 (page 8-53) shows the `Example15.java` file, an advanced example that returns the total number (COUNT) of matches (and only of matches).

Example 8-18 Get COUNT of Matches [Advanced]

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
```

```
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example15
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);

    OracleSailConnection osc = store.getConnection();
    ValueFactory f = osc.getValueFactory();

    // create some resources and literals to make statements out of
    URI alice = f.createURI("http://example.org/people/alice");
    URI name = f.createURI("http://example.org/ontology/name");
    URI person = f.createURI("http://example.org/ontology/Person");
    Literal alicesName = f.createLiteral("Alice");

    osc.addStatement(alice, RDF.TYPE, person);
    osc.addStatement(alice, name, alicesName);
    osc.commit();

    osc.analyze();
    osc.analyzeApplicationTable();

    // Run a query and only return the number of matches (the count!)
    String queryString =
      " PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#count_only> " +
      " SELECT ?totalCount WHERE {?s ?p ?y} ";

    TupleQuery tupleQuery = osc.asRepositoryConnection().prepareTupleQuery(
        QueryLanguage.SPARQL, queryString);

    TupleQueryResult result = tupleQuery.evaluate();
    try {
      if (result.hasNext()) {
        BindingSet bindingSet = result.next();
        psOut.println("number of matches: " + bindingSet.getValue("totalCount"));
```

```
      }
    }
    finally {
      result.close();
    }

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example15.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example15
jdbc:oracle:thin:@localhost:1521:ORCL scott <password> EX15
```

The expected output of the `java` command might appear as follows:

```
number of matches: "2"^^<http://www.w3.org/2001/XMLSchema#integer>
```

8.10.16 Example16.java: Specify Bind Variable for Constant in Query Pattern *[Advanced]*

Example 8-19 (page 8-55) shows the `Example16.java` file, an advanced example that specifies a bind variable for a constant in the SPARQL query pattern.

Example 8-19 Specify Bind Variable for Constant in Query Pattern [Advanced]

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example16
{
  public static void main(String[] args) throws SQLException, SailException,
```

```
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);

    OracleSailConnection osc = store.getConnection();
    ValueFactory f = osc.getValueFactory();

    // create some resources and literals to make statements out of
    URI alice = f.createURI("http://example.org/people/alice");
    URI name = f.createURI("http://example.org/ontology/name");
    URI person = f.createURI("http://example.org/ontology/Person");
    Literal alicesName = f.createLiteral("Alice");

    osc.addStatement(alice, RDF.TYPE, person);
    osc.addStatement(alice, name, alicesName);
    osc.commit();

    osc.analyze();
    osc.analyzeApplicationTable();

    String queryString =
      " PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#use_bind_var(1)> " +
      " SELECT ?p WHERE { <http://example.org/people/alice> <http://www.w3.org/
1999/02/22-rdf-syntax-ns#type> ?p} ";

    TupleQuery tupleQuery = osc.asRepositoryConnection().prepareTupleQuery(
        QueryLanguage.SPARQL, queryString);

    TupleQueryResult result = tupleQuery.evaluate();
    try {
      if (result.hasNext()) {
        BindingSet bindingSet = result.next();
        psOut.println("solution " + bindingSet.toString());
      }
    }
    finally {
      result.close();
    }

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example16.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example16
jdbc:oracle:thin:@localhost:1521:ORCL scott <password> EX16
```

The expected output of the `java` command might appear as follows:

```
solution [p=http://example.org/ontology/Person]
```

8.10.17 Example17.java: Specify Bind Variable for Constant in Different Position in Query Pattern *[Advanced]*

Example 8-20 (page 8-57) shows the `Example17.java` file, an advanced example that specifies a bind variable for a constant in a different position in the SPARQL query pattern than in `Example16.java` (Example 8-19 (page 8-55)).

Example 8-20 Specify Bind Variable for Constant in Different Position in Query Pattern [Advanced]

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;

import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example17
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);

    OracleSailConnection osc = store.getConnection();
    ValueFactory f = osc.getValueFactory();

    // create some resources and literals to make statements out of
    URI alice = f.createURI("http://example.org/people/alice");
```

```
    URI name = f.createURI("http://example.org/ontology/name");
    URI person = f.createURI("http://example.org/ontology/Person");
    Literal alicesName = f.createLiteral("Alice");

    osc.addStatement(alice, RDF.TYPE, person);
    osc.addStatement(alice, name, alicesName);
    osc.commit();

    osc.analyze();
    osc.analyzeApplicationTable();

    String queryString =
      " PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#use_bind_var(2)> " +
      " SELECT ?s WHERE { ?s <http://example.org/ontology/name> \"Alice\" } ";

    TupleQuery tupleQuery = osc.asRepositoryConnection().prepareTupleQuery(
        QueryLanguage.SPARQL, queryString);

    TupleQueryResult result = tupleQuery.evaluate();
    try {
      if (result.hasNext()) {
        BindingSet bindingSet = result.next();
        psOut.println("solution " + bindingSet.toString());
      }
    }
    finally {
      result.close();
    }

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example17.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example17
jdbc:oracle:thin:@localhost:1521:ORCL scott <password> EX17
```

The expected output of the `java` command might appear as follows:

```
solution [s=http://example.org/people/alice]
```

8.10.18 Example18.java: Build URIs from Internal Numeric IDs *[Advanced]*

Example 8-21 (page 8-58) shows the `Example18.java` file, an advanced example that builds URIs from internal numeric IDs.

Example 8-21 Build URIs from Internal Numeric IDs [Advanced]

```
import org.openrdf.model.Statement;
import org.openrdf.model.URI;
import org.openrdf.model.ValueFactory;
import info.aduna.iteration.CloseableIteration;

import java.io.*;
import java.sql.SQLException;
```

```
import oracle.spatial.rdf.client.sesame.*;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.SailConnection;
import org.openrdf.sail.SailException;
import org.openrdf.model.vocabulary.RDFS;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFParseException;
import org.openrdf.model.Literal;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryLanguage;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.MalformedQueryException;

public class Example18
{
  public static void main(String[] args) throws SQLException, SailException,
    RepositoryException, IOException, RDFParseException, QueryEvaluationException,
    MalformedQueryException
  {
    PrintStream psOut = System.out;
    String jdbcUrl  = args[0];
    String user     = args[1];
    String password = args[2];
    String model    = args[3];

    OraclePool op = new OraclePool(
        OraclePool.getOracleDataSource(jdbcUrl, user, password));
    OracleSailStore store = new OracleSailStore(op, model);

    OracleSailConnection osc = store.getConnection();
    ValueFactory f = osc.getValueFactory();

    // create some resources and literals to make statements out of
    URI alice = f.createURI("http://example.org/people/alice");
    URI name = f.createURI("http://example.org/ontology/name");
    URI person = f.createURI("http://example.org/ontology/Person");
    Literal alicesName = f.createLiteral("Alice");

    osc.addStatement(alice, RDF.TYPE, person);
    osc.addStatement(alice, name, alicesName);
    osc.commit();

    osc.analyze();
    osc.analyzeApplicationTable();

    String queryString =
      " PREFIX ORACLE_SEM_FS_NS: <http://oracle.com/semtech#build-uri-for-id(?p)> " +
      " SELECT ?p WHERE { <http://example.org/people/alice> <http://www.w3.org/
1999/02/22-rdf-syntax-ns#type> ?p} ";

    TupleQuery tupleQuery = osc.asRepositoryConnection().prepareTupleQuery(
        QueryLanguage.SPARQL, queryString);

    TupleQueryResult result = tupleQuery.evaluate();
    try {
```

```
      if (result.hasNext()) {
        BindingSet bindingSet = result.next();
        psOut.println("solution " + bindingSet.toString());
      }
    }
    finally {
      result.close();
    }

    osc.close();
    store.shutDown();
    op.close();
  }
}
```

To compile this example, enter the following command:

```
usr/local/packages/jdk16/bin/javac -classpath $CP Example18.java
```

To run this example, enter a command in the following format:

```
/usr/local/packages/jdk16/bin/java -classpath $CP Example18
jdbc:oracle:thin:@localhost:1521:ORCL scott <password> EX18
```

The expected output of the `java` command might appear as follows:

```
solution [p=rdfvid:428072448720587401]
```

9

User-Defined Inferencing and Querying

Note:

The capabilities described in this chapter are intended for advanced users. You are assumed to be familiar with the main concepts and techniques described in RDF Semantic Graph Overview (page 1-1) and OWL Concepts (page 2-1).

This chapter describes RDF Semantic Graph extension architectures that, effective with Oracle Database 12*c* Release 1 (12.1), enable the addition of user-defined capabilities:

- The inference extension architecture enables you to add user-defined inferencing to the presupplied inferencing support, as explained in User-Defined Inferencing (page 9-1).
- The query extension architecture enables you to add user-defined functions and aggregates to be used in SPARQL queries, both through the SEM_MATCH table function and through the support for Apache Jena and OpenRDF Sesame, as explained in User-Defined Functions and Aggregates (page 9-28).

9.1 User-Defined Inferencing

The RDF Semantic Graph inference extension architecture enables you to add user-defined inferencing to the presupplied inferencing support.

This section contains the following major topics:

- Problem Solved and Benefit Provided by User-Defined Inferencing (page 9-1)
- API Support for User-Defined Inferencing (page 9-2)
- User-Defined Inference Extension Function Examples (page 9-4)

9.1.1 Problem Solved and Benefit Provided by User-Defined Inferencing

Before Oracle Database 12*c* Release 1 (12.1), the Oracle Database inference engine provided native support for OWL 2 RL,RDFS, SKOS, SNOMED (core EL), and user-defined rules, which covered a wide range of applications and requirements. However, there was the limitation that **no new RDF resources** could be created as part of the rules deduction process.

As an example of the capabilities and the limitation before Oracle Database 12*c* Release 1 (12.1), consider the following straightforward inference rule:

```
?C   rdfs:subClassOf  ?D .
?x    rdf:type  ?C  .  ==>  ?x   rdf:type  ?D
```

The preceding rule says that any instance `x` of a subclass `C` will be an instance of `C`'s superclass, `D`. The consequent part of the rule mentions two variables `?x` and `?D`. However, these variables must already exist in the antecedents of the rule, which further implies that these RDF resources must already exist in the knowledge base. In other words, for example, you can derive that `John` is a `Student` only if you know that `John` *exists* as a `GraduateStudent` and if an axiom specifies that the `GraduateStudent` class is a subclass of the `Student` class.

Another example of a limitation is that before Oracle Database 12*c* Release 1 (12.1), the inference functions did not support combining a person's first name and last name to produce a full name as a *new* RDF resource in the inference process. Specifically, this requirement can be captured as a rule like the following:

```
?x   :firstName  ?fn
?x   :lastName   ?ln  ==>  ?x  :fullName  concatenate(?fn ?ln)
```

Effective with Oracle Database 12*c* Release 1 (12.1), the RDF Semantic Graph inference extension architecture opens the inference process so that users can implement their own inference extension functions and integrate them into the native inference process. This architecture:

- Supports rules that require the generation of new RDF resources.

  Examples might include concatenation of strings or other string operations, mathematical calculations, and web service callouts.

- Allows implementation of certain existing rules using customized optimizations.

  Although the native OWL inference engine has optimizations for many rules and these rules work efficiently for a variety of large-scale ontologies, for some new untested ontologies a customized optimization of a particular inference component may work even better. In such a case, you can disable a particular inference component in the SEM_APIS.CREATE_ENTAILMENT (page 11-15) call and specify a customized inference extension function (using the `inf_ext_user_func_name` parameter) that implements the new optimization.

- Allows the inference engine to be extended with sophisticated inference capabilities.

  Examples might include integrating geospatial reasoning, time interval reasoning, and text analytical functions into the native database inference process.

9.1.2 API Support for User-Defined Inferencing

The primary application programming interface (API) for user-defined inferencing is the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, specifically the last parameter:

```
inf_ext_user_func_name  IN VARCHAR2 DEFAULT NULL
```

The `inf_ext_user_func_name` parameter, if specified, identifies one or more user-defined inference functions that implement the specialized logic that you want to use. Each user-defined inference function must follow the requirements and guidelines in User-Defined Inference Function Requirements (page 9-2).

9.1.2.1 User-Defined Inference Function Requirements

Each user-defined inference function that is specified in the `inf_ext_user_func_name` parameter in the call to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure must:

- Have a name that starts with the following string: SEM_INF_
- Be created with definer's rights, not invoker's rights. (For an explanation of definer's rights and invoker's rights, see *Oracle Database Security Guide*.)

The format of the user-defined inference function must be that shown in the following example for a hypothetical function named SEM_INF_EXAMPLE:

```
create or replace function sem_inf_example(
    src_tab_view         in  varchar2,
    resource_id_map_view in  varchar2,
    output_tab           in  varchar2,
    action               in  varchar2,
    num_calls            in  number,
    tplInferredLastRound in  number,
    options              in  varchar2 default null,
    optimization_flag    out number,
    diag_message         out varchar2
    )
return boolean
as
  pragma autonomous_transaction;
begin
  if (action = SDO_SEM_INFERENCE.INF_EXT_ACTION_START) then
    <... preparation work ...>
  end if;
  if (action = SDO_SEM_INFERENCE.INF_EXT_ACTION_RUN) then
    <... actual inference logic ...>
    commit;
  end if;
  if (action = SDO_SEM_INFERENCE.INF_EXT_ACTION_END) then
    <... clean up ...>
  end if;
return true;  -- succeed
end;
/
grant execute on sem_inf_example to MDSYS;
```

In the user-defined function format, the optimization_flag output parameter can specify one or more Oracle-defined names that are associated with numeric values. You can specify one or more of the following:

- SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_NONE indicates that the inference engine should not enable any optimizations for the extension function. (This is the default behavior of the inference engine when the optimization_flag parameter is not set.)
- SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_ALL_IDS indicates that all triples/quads inferred by the extension function use only resource IDs. In other words, the output_tab table only contains resource IDs (columns gid, sid, pid, and oid) and does not contain any lexical values (columns g, s, p, and o are all null). Enabling this optimization flag allows the inference engine to skip resource ID lookups.
- SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_NEWDATA_ONLY indicates that all the triples/quads inferred by the extension function are new and do not already exist in src_tab_view. Enabling this optimization flag allows the inference engine to skip checking for duplicates between the output_tab table and src_tab_view. Note that the src_tab_view contains triples/quads from

previous rounds of reasoning, including triples/quads inferred from extension functions.

- `SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_UNIQDATA_ONLY` indicates that all the triples/quads inferred by the extension function are unique and do not already exist in the `output_tab` table. Enabling this optimization flag allows the inference engine to skip checking for duplicates within the `output_tab` table (for example, no need to check for the same triple inferred twice by an extension function). Note that the `output_tab` table is empty at the beginning of each round of reasoning for an extension function, so uniqueness of the data must only hold for the current round of reasoning.
- `SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_IGNORE_NULL` indicates that the inference engine should ignore an inferred triple or quad if the subject, predicate, or object resource is null. The inference engine considers a resource null if both of its columns in the `output_tab` table are null (for example, subject is null if the `s` and `sid` columns are both null). Enabling this optimization flag allows the inference engine to skip invalid triples/quads in the `output_tab` table. Note that the inference engine interprets null graph columns (`g` and `gid`) as the default graph.

To specify more than one value for the `optimization_flag` output parameter, use the plus sign (+) to concatenate the values. For example:

```
optimization_flag := SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_ALL_IDS +
                     SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_NEWDATA_ONLY +
                     SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_UNIQDATA_ONLY;
```

For more information about using the `optimization_flag` output parameter, see Example 3: Optimizing Performance (page 9-10).

9.1.3 User-Defined Inference Extension Function Examples

The following examples demonstrate how to use user-defined inference extension functions to create entailments.

- Example 1: Adding Static Triples (page 9-5), Example 2: Adding Dynamic Triples (page 9-7), and Example 3: Optimizing Performance (page 9-10) cover the basics of user-defined inference extensions.

  Example 1: Adding Static Triples (page 9-5) and Example 2: Adding Dynamic Triples (page 9-7) focus on adding new, inferred triples.

  Example 3: Optimizing Performance (page 9-10) focuses on optimizing performance.

- Example 4: Temporal Reasoning (Several Related Examples) (page 9-13) and Example 5: Spatial Reasoning (page 9-21) demonstrate how to handle special data types efficiently by leveraging native Oracle types and operators.

  Example 4: Temporal Reasoning (Several Related Examples) (page 9-13) focuses on the `xsd:dateTime` data type.

  Example 5: Spatial Reasoning (page 9-21) focuses on geospatial data types.

- Example 6: Calling a Web Service (page 9-24) makes a web service call to the Oracle Geocoder service.

The first three examples assume that the model `EMPLOYEES` exists and contains the following semantic data, displayed in Turtle format:

```
:John   :firstName  "John"  ;
        :lastName   "Smith" .

:Mary   :firstName  "Mary"  ;
        :lastName   "Smith" ;
        :name       "Mary Smith" .

:Alice  :firstName  "Alice" .

:Bob    :firstName  "Bob" ;
        :lastName   "Billow" .
```

For requirements and guidelines for creating user-defined inference extension functions, see API Support for User-Defined Inferencing (page 9-2).

9.1.3.1 Example 1: Adding Static Triples

The most basic method to infer new data in a user-defined inference extension function is adding static data. Static data does not depend on any existing data in a model. This is not a common case for a user-defined inference extension function, but it demonstrates the basics of adding triples to an entailment. Inserting static data is more commonly done during the preparation phase (that is, `action='START'`) to expand on the existing ontology.

The following user-defined inference extension function (`sem_inf_static`) adds three static triples to an entailment:

```
-- this user-defined rule adds static triples
create or replace function sem_inf_static(
    src_tab_view         in  varchar2,
    resource_id_map_view in  varchar2,
    output_tab           in  varchar2,
    action               in  varchar2,
    num_calls            in  number,
    tplInferredLastRound in  number,
    options              in  varchar2 default null,
    optimization_flag    out number,
    diag_message         out varchar2
    )
return boolean
as
  query varchar2(4000);
  pragma autonomous_transaction;
begin
  if (action = 'RUN') then
    -- generic query we use to insert triples
    query :=
      'insert /*+ parallel append */ into ' || output_tab ||
      ' ( s,  p,  o) VALUES ' ||
      ' (:1, :2, :3) ';

    -- execute the query with different values
    execute immediate query using
      '<http://example.org/S1>', '<http://example.org/P2>', '"O1"';

    execute immediate query using
      '<http://example.org/S2>', '<http://example.org/P2>', '"2"^^xsd:int';

    -- duplicate quad
    execute immediate query using
      '<http://example.org/S2>', '<http://example.org/P2>', '"2"^^xsd:int';
```

```
    execute immediate query using
      '<http://example.org/S3>', '<http://example.org/P3>', '"3.0"^^xsd:double';

    -- commit our changes
    commit;
  end if;

  -- return true to indicate success
  return true;
end sem_inf_static;
/
show errors;
```

The `sem_inf_static` function inserts new data by executing a SQL insert query, with `output_tab` as the target table for insertion. The `output_tab` table will only contain triples added by the `sem_inf_static` function during the current call (see the `num_calls` parameter). The inference engine will always call a user-defined inference extension function at least three times, once for each possible value of the action parameter (`'START'`, `'RUN'`, and `'END'`). Because `sem_inf_static` does not need to perform any preparation or cleanup, the function only adds data during the `RUN` phase. The extension function can be called more than once during the `RUN` phase, depending on the data inferred during the current round of reasoning.

Although the `sem_inf_static` function makes no checks for existing triples (to prevent duplicate triples), the inference engine will not generate duplicate triples in the resulting entailment. The inference engine will filter out duplicates from the `output_tab` table (the data inserted by the extension function) and from the final entailment (the model or models and other inferred data). Setting the appropriate optimization flags (using the `optimization_flag` parameter) will disable this convenience feature and improve performance. (See Example 3: Optimizing Performance (page 9-10) for more information about optimization flags.)

Although the table definition for `output_tab` shows a column for graph names, the inference engine will ignore and override all graph names on triples added by extension functions when performing Global Inference (default behavior of SEM_APIS.CREATE_ENTAILMENT (page 11-15)) and Named Graph Global Inference (NGGI). To add triples to specific named graphs in a user-defined extension function, use NGLI (Named Graph Local Inference). During NGLI, all triples must belong to a named graph (that is, the `gid` and `g` columns of `output_tab` cannot both be null).

The MDSYS user must have execute privileges on the `sem_inf_static` function to use the function for reasoning. The following example shows how to grant the appropriate privileges on the `sem_inf_static` function and create an entailment using the function (along with OWLPRIME inference logic):

```
-- grant appropriate privileges
grant execute on sem_inf_static to mdsys;

-- create the entailment
begin
  sem_apis.create_entailment(
    'EMPLOYEES_INF'
  , sem_models('EMPLOYEES')
  , sem_rulebases('OWLPRIME')
  , passes => SEM_APIS.REACH_CLOSURE
  , inf_ext_user_func_name => 'sem_inf_static'
);
```

```
end;
/
```

The following example displays the newly entailed data:

```
-- formatting
column s format a23;
column p format a23;
column o format a23;
set linesize 100;

-- show results
select s, p, o from table(SEM_MATCH(
    'select ?s ?p ?o where { ?s ?p ?o } order by ?s ?p ?o'
  , sem_models('EMPLOYEES')
  , sem_rulebases('OWLPRIME')
  , null, null, null
  , 'INF_ONLY=T'));
```

The preceding query returns the three unique static triples added by `sem_inf_static`, with no duplicates:

```
S                       P                       O
----------------------- ----------------------- -----------------------
http://example.org/S1  http://example.org/P2  O1
http://example.org/S2  http://example.org/P2  2
http://example.org/S3  http://example.org/P3  3E0
```

9.1.3.2 Example 2: Adding Dynamic Triples

Adding static data is useful, but it is usually done during the preparation (that is, `action='START'`) phase. Adding *dynamic* data involves looking at existing data in the model and generating new data based on the existing data. This is the most common case for a user-defined inference extension function.

The following user-defined inference extension function (`sem_inf_dynamic`) concatenates the first and last names of employees to create a new triple that represents the full name.

```
-- this user-defined rule adds static triples
create or replace function sem_inf_dynamic(
    src_tab_view         in  varchar2,
    resource_id_map_view in  varchar2,
    output_tab           in  varchar2,
    action               in  varchar2,
    num_calls            in  number,
    tplInferredLastRound in  number,
    options              in  varchar2 default null,
    optimization_flag    out number,
    diag_message         out varchar2
    )
return boolean
as
  firstNamePropertyId number;
  lastNamePropertyId  number;
  fullNamePropertyId  number;

  sqlStmt    varchar2(4000);
  insertStmt varchar2(4000);
  pragma autonomous_transaction;
begin
  if (action = 'RUN') then
```

```
    -- retrieve ID of resource that already exists in the data (will
    -- throw exception if resource does not exist). These will improve
    -- performance of our SQL queries.
    firstNamePropertyId := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/firstName');
    lastNamePropertyId  := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/lastName');
    fullNamePropertyId  := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/name');

    -- SQL query to find all employees and their first and last names
    sqlStmt :=
      'select ids1.sid employeeId,
              values1.value_name firstName,
              values2.value_name lastName
       from   ' || resource_id_map_view || ' values1,
              ' || resource_id_map_view || ' values2,
              ' || src_tab_view || '         ids1,
              ' || src_tab_view || '         ids2
       where  ids1.sid = ids2.sid
         AND  ids1.pid = ' || to_char(firstNamePropertyId,'TM9') || '
         AND  ids2.pid = ' || to_char(lastNamePropertyId,'TM9')  || '
         AND  ids1.oid = values1.value_id
         AND  ids2.oid = values2.value_id
       /* below ensures we have NEWDATA (a no duplicate optimization flag) */
         AND  not exists
               (select 1
                from   ' || src_tab_view || '
                where  sid = ids1.sid AND
                       pid = ' || to_char(fullNamePropertyId,'TM9') || ')';

    -- create the insert statement that concatenates the first and
    -- last names from our sqlStmt into a new triple.
    insertStmt :=
      'insert /*+ parallel append */
       into ' || output_tab || ' (sid, pid, o)
       select employeeId, ' || to_char(fullNamePropertyId,'TM9') || ', '''''' ||
firstName || '' '' || lastName ||  ''''''
       from   (' || sqlStmt || ')';

    -- execute the insert statement
    execute immediate insertStmt;

    -- commit our changes
    commit;

    -- set our optimization flags indicating we already checked for
    -- duplicates in the model (src_tab_view)
    optimization_flag := SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_NEWDATA_ONLY;
  end if;

  -- return true to indicate success
  return true;
end sem_inf_dynamic;
/
show errors;
```

The `sem_inf_dynamic` function inserts new data using two main steps. First, the function builds a SQL query that collects all first and last names from the existing data. The `sqlStmt` variable stores this SQL query. Next, the function inserts new triples based on the first and last names it collects, to form a full name for each employee. The

insertStmt variable stores this SQL query. Note that the insertStmt query includes the sqlStmt query because it is performing an INSERT with a subquery.

The sqlStmt query performs a join across two main views: the resource view (resource_id_map_view) and the existing data view (src_tab_view). The existing data view contains all existing triples but stores the values of those triples using numeric IDs instead of lexical values. Because the sqlStmt query must extract the lexical values of the first and last names of an employee, it joins with the resource view twice (once for the first name and once for the last name).

The sqlStmt query contains the PARALLEL SQL hint to help improve performance. Parallel execution on a balanced hardware configuration can significantly improve performance. (See Example 3: Optimizing Performance (page 9-10) for more information.)

The insertStmt query also performs a duplicate check to avoid adding a triple if it already exists in the existing data view (src_tab_view). The function indicates it has performed this check by enabling the INF_EXT_OPT_FLAG_NEWDATA_ONLY optimization flag. Doing the check inside the extension function improves overall performance of the reasoning. Note that the existing data view does not contain the new triples currently being added by the sem_inf_dynamic function, so duplicates may still exist within the output_tab table. If the sem_inf_dynamic function additionally checked for duplicates within the output_tab table, then it could also enable the INF_EXT_OPT_FLAG_UNIQUEDATA_ONLY optimization flag.

Both SQL queries use numeric IDs of RDF resources to perform their joins and inserts. Using IDs instead of lexical values improves the performance of the queries. The sem_inf_dynamic function takes advantage of this performance benefit by looking up the IDs of the lexical values it plans to use. In this case, the function looks up three URIs representing the first name, last name, and full name properties. If the sem_inf_dynamic function inserted all new triples purely as IDs, then it could enable the INF_EXT_OPT_FLAG_ALL_IDS optimization flag. For this example, however, the new triples each contain a single, new, lexical value: the full name of the employee.

To create an entailment with the sem_inf_dynamic function, grant execution privileges to the MDSYS user, then pass the function name to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, as follows:

```
-- grant appropriate privileges
grant execute on sem_inf_dynamic to mdsys;

-- create the entailment
begin
  sem_apis.create_entailment(
    'EMPLOYEES_INF'
  , sem_models('EMPLOYEES')
  , sem_rulebases('OWLPRIME')
  , passes => SEM_APIS.REACH_CLOSURE
  , inf_ext_user_func_name => 'sem_inf_dynamic'
);
end;
/
```

The entailment should contain the following two new triples added by sem_inf_dynamic:

```
S                      P                      O
---------------------- ---------------------- ----------------------
```

```
http://example.org/Bob   http://example.org/name  Bob Billow
http://example.org/John  http://example.org/name  John Smith
```

Note that the `sem_inf_dynamic` function in the preceding example did not infer a full name for Mary Smith, because Mary Smith already had her full name specified in the existing data.

9.1.3.3 Example 3: Optimizing Performance

Several techniques can improve the performance of an inference extension function. One such technique is to use the numeric IDs of resources rather than their lexical values in queries. By only using resource IDs, the extension function avoids having to join with the resource view (`resource_id_map_view`), and this can greatly improve query performance. Inference extension functions can obtain additional performance benefits by also using resource IDs when adding new triples to the `output_tab` table (that is, using only using the `gid`, `sid`, `pid`, and `oid` columns of the `output_tab` table).

The following user-defined inference extension function (`sem_inf_related`) infers a new property, `:possibleRelative`, for employees who share the same last name. The SQL queries for finding such employees use only resource IDs (no lexical values, no joins with the resource view). Additionally, the inference extension function in this example inserts the new triples using only resource IDs, allowing the function to enable the `INF_EXT_OPT_FLAG_ALL_IDS` optimization flag.

```
-- this user-defined rule adds static triples
create or replace function sem_inf_related(
    src_tab_view         in  varchar2,
    resource_id_map_view in  varchar2,
    output_tab           in  varchar2,
    action               in  varchar2,
    num_calls            in  number,
    tplInferredLastRound in  number,
    options              in  varchar2 default null,
    optimization_flag    out number,
    diag_message         out varchar2
    )
return boolean
as
  lastNamePropertyId  number;
  relatedPropertyId   number;

  sqlStmt    varchar2(4000);
  insertStmt varchar2(4000);
  pragma autonomous_transaction;
begin
  if (action = 'RUN') then
    -- retrieve ID of resource that already exists in the data (will
    -- throw exception if resource does not exist).
    lastNamePropertyId := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/lastName');

    -- retreive ID of resource or generate a new ID if resource does
    -- not already exist
    relatedPropertyId := sdo_sem_inference.oracle_orardf_add_res('http://example.org/
possibleRelative');

    -- SQL query to find all employees that share a last name
    sqlStmt :=
      'select ids1.sid employeeId,
```

```
              ids2.sid relativeId
       from   ' || src_tab_view || '         ids1,
              ' || src_tab_view || '         ids2
       where  ids1.pid = ' || to_char(lastNamePropertyId,'TM9') || '
         AND  ids2.pid = ' || to_char(lastNamePropertyId,'TM9') || '
         AND  ids1.oid  = ids2.oid
       /* avoid employees related to themselves */
         AND  ids1.sid != ids2.sid
       /* below ensures we have NEWDATA (a no duplicate optimization flag) */
         AND  not exists
               (select 1
                from   ' || src_tab_view || '
                where  sid = ids1.sid
                  AND  pid = ' || to_char(relatedPropertyId,'TM9') || '
                  AND  oid = ids2.sid)
       /* below ensures we have UNIQDATA (a no duplicate optimization flag) */
         AND  not exists
               (select 1
                from   ' || output_tab || '
                where  sid = ids1.sid
                  AND  pid = ' || to_char(relatedPropertyId,'TM9') || '
                  AND  oid = ids2.sid)';

    -- create the insert statement that only uses resource IDs
    insertStmt :=
      'insert /*+ parallel append */
       into ' || output_tab || ' (sid, pid, oid)
       select employeeId, ' || to_char(relatedPropertyId,'TM9') || ', relativeId
       from   (' || sqlStmt || ')';

    -- execute the insert statement
    execute immediate insertStmt;

    -- commit our changes
    commit;

    -- set flag indicating our new triples
    --   1) are specified using only IDs
    --   2) produce no duplicates with the model (src_tab_view)
    --   3) produce no duplicates in the output (output_tab)
    optimization_flag := SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_ALL_IDS +
                         SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_NEWDATA_ONLY +
                         SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_UNIQDATA_ONLY;
  end if;

  -- return true to indicate success
  return true;
end sem_inf_related;
/
show errors;
```

The `sem_inf_related` function has a few key differences from previous examples. First, the `sem_inf_related` function queries purely with resource IDs and inserts new triples using only resource IDs. Because all the added triples in the `output_tab` table only use resource IDs, the function can enable the `INF_EXT_OPT_FLAG_ALL_IDS` optimization flag. For optimal performance, functions should try to use resource IDs over lexical values. However, sometimes this is not possible, as in Example 2: Adding Dynamic Triples (page 9-7), which concatenates lexical values to form a new lexical value. Note that in cases like Example 2: Adding Dynamic Triples (page 9-7), it is usually better to join with the resource

view (`resource_id_map_view`) than to embed calls to `oracle_orardf_res2vid` within the SQL query. This is due to the overhead of calling the function for each possible match as opposed to joining with another table.

Another key difference in the `sem_inf_related` function is the use of the `oracle_orardf_add_res` function (compared to `oracle_orardf_res2vid`). Unlike the `res2vid` function, the `add_res` function will add a resource to the resource view (`resource_id_map_view`) if the resource does not already exist. Inference extensions functions should use the `add_res` function if adding the resource to the resource view is not a concern. Calling the function multiple times will not generate duplicate entries in the resource view.

The last main difference is the additional `NOT EXISTS` clause in the SQL query. The first `NOT EXISTS` clause avoids adding any triples that may be duplicates of triples already in the model or triples inferred by other rules (`src_tab_view`). Checking for these duplicates allows `sem_inf_related` to enable the `INF_EXT_OPT_FLAG_NEWDATA_ONLY` optimization flag. The second `NOT EXISTS` clause avoids adding triples that may be duplicates of triples already added by the `sem_inf_related` function to the output_tab table during the current round of reasoning (see the `num_calls` parameter). Checking for these duplicates allows `sem_inf_related` to enable the `INF_EXT_OPT_FLAG_UNIQDATA_ONLY` optimization flag.

Like the `sem_inf_dynamic` example, `sem_inf_related` example uses a `PARALLEL` SQL query hint in its insert statement. Parallel execution on a balanced hardware configuration can significantly improve performance. For a data-intensive application, a good I/O subsystem is usually a critical component to the performance of the whole system.

To create an entailment with the `sem_inf_dynamic` function, grant execution privileges to the MDSYS user, then pass the function name to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, as follows:

```
-- grant appropriate privileges
grant execute on sem_inf_related to mdsys;

-- create the entailment
begin
  sem_apis.create_entailment(
    'EMPLOYEES_INF'
  , sem_models('EMPLOYEES')
  , sem_rulebases('OWLPRIME')
  , passes => SEM_APIS.REACH_CLOSURE
  , inf_ext_user_func_name => 'sem_inf_related'
);
end;
/
```

The entailment should contain the following two new triples added by `sem_inf_related`:

```
S                      P                                   O
---------------------- ----------------------------------
-----------------------
http://example.org/John  http://example.org/possibleRelative  http://example.org/Mary
http://example.org/Mary  http://example.org/possibleRelative  http://example.org/John
```

9.1.3.4 Example 4: Temporal Reasoning (Several Related Examples)

User-defined extension functions enable you to better leverage certain data types (like `xsd:dateTime`) in the triples. For example, with user-defined extension functions, it is possible to infer relationships between triples based on the difference between two `xsd:dateTime` values. The three examples in this section explore two different temporal reasoning rules and how to combine them into one entailment. The examples assume the models `EVENT` and `EVENT_ONT` exist and contain the following semantic data:

EVENT_ONT

```
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix :     <http://example.org/event/> .

# we model two types of events
:Meeting      rdfs:subClassOf :Event .
:Presentation rdfs:subClassOf :Event .

# events have topics
:topic        rdfs:domain     :Event .

# events have start and end times
:startTime    rdfs:domain     :Event ;
              rdfs:range      xsd:dateTime .
:endTime      rdfs:domain     :Event ;
              rdfs:range      xsd:dateTime .

# duration (in minutes) of an event
:lengthInMins rdfs:domain      :Event ;
              rdfs:range       xsd:integer .

# overlaps property identifies conflicting events
:overlaps     rdfs:domain      :Event ;
              rdf:type         owl:SymmetricProperty .
:noOverlap    rdfs:domain      :Event ;
              rdf:type         owl:SymmetricProperty .
```

EVENT_TBOX

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix :    <http://example.org/event/> .

:m1 rdf:type   :Meeting ;
    :topic     "Beta1 launch" ;
    :startTime "2012-04-01T09:30:00-05:00"^^xsd:dateTime ;
    :endTime   "2012-04-01T11:00:00-05:00"^^xsd:dateTime .

:m2 rdf:type   :Meeting ;
    :topic     "Standards compliance" ;
    :startTime "2012-04-01T12:30:00-05:00"^^xsd:dateTime ;
    :endTime   "2012-04-01T13:30:00-05:00"^^xsd:dateTime .

:p1 rdf:type   :Presentation ;
    :topic     "OWL Reasoners" ;
    :startTime "2012-04-01T11:00:00-05:00"^^xsd:dateTime ;
    :endTime   "2012-04-01T13:00:00-05:00"^^xsd:dateTime .
```

The examples are the following:

- Example 4a: Duration Rule (page 9-14)
- Example 4b: Overlap Rule (page 9-17)
- Example 4c: Duration and Overlap Rules (page 9-20)

9.1.3.4.1 Example 4a: Duration Rule

The following user-defined inference extension function (`sem_inf_durations`) infers the duration in minutes of events, given the start and end times of an event. For example, an event starting at 9:30 AM and ending at 11:00 AM has duration of 90 minutes. The following extension function extracts the start and end times for each event, converts the `xsd:dateTime` values into Oracle timestamps, then computes the difference between the timestamps. Notice that this extension function can handle time zones.

```
create or replace function sem_inf_durations(
    src_tab_view         in  varchar2,
    resource_id_map_view in  varchar2,
    output_tab           in  varchar2,
    action               in  varchar2,
    num_calls            in  number,
    tplInferredLastRound in  number,
    options              in  varchar2 default null,
    optimization_flag    out number,
    diag_message         out varchar2
    )
return boolean
as
  eventClassId        number;
  rdfTypePropertyId   number;
  startTimePropertyId number;
  endTimePropertyId   number;
  durationPropertyId  number;

  xsdTimeFormat       varchar2(100);
  sqlStmt             varchar2(4000);
  insertStmt          varchar2(4000);

  pragma autonomous_transaction;
begin
  if (action = 'RUN') then
    -- retrieve ID of resource that already exists in the data (will
    -- throw exception if resource does not exist).
    eventClassId        := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/event/Event');
    startTimePropertyId := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/event/startTime');
    endTimePropertyId   := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/event/endTime');
    durationPropertyId  := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/event/lengthInMins');
    rdfTypePropertyId   := sdo_sem_inference.oracle_orardf_res2vid('http://
www.w3.org/1999/02/22-rdf-syntax-ns#type');

    -- set the TIMESTAMP format we will use to parse XSD times
    xsdTimeFormat := 'YYYY-MM-DD"T"HH24:MI:SSTZH:TZM';

    -- query we use to extract the event ID and start/end times.
```

```
    sqlStmt :=
      'select ids1.sid eventId,
              TO_TIMESTAMP_TZ(values1.value_name,''YYYY-MM-DD"T"HH24:MI:SSTZH:TZM'')
startTime,
              TO_TIMESTAMP_TZ(values2.value_name,''YYYY-MM-DD"T"HH24:MI:SSTZH:TZM'')
endTime
       from   ' || resource_id_map_view || ' values1,
              ' || resource_id_map_view || ' values2,
              ' || src_tab_view || '         ids1,
              ' || src_tab_view || '         ids2,
              ' || src_tab_view || '         ids3
       where  ids1.sid = ids3.sid
         AND  ids3.pid = ' || to_char(rdfTypePropertyId,'TM9') || '
         AND  ids3.oid = ' || to_char(eventClassId,'TM9')      || '
         AND  ids1.sid = ids2.sid
         AND  ids1.pid = ' || to_char(startTimePropertyId,'TM9') || '
         AND  ids2.pid = ' || to_char(endTimePropertyId,'TM9')   || '
         AND  ids1.oid = values1.value_id
         AND  ids2.oid = values2.value_id
       /* ensures we have NEWDATA */
         AND  not exists
              (select 1
               from   ' || src_tab_view || '
               where  sid = ids3.sid
                 AND  pid = ' || to_char(durationPropertyId,'TM9') || ')
       /* ensures we have UNIQDATA */
         AND  not exists
              (select 1
               from   ' || output_tab || '
               where  sid = ids3.sid
                 AND  pid = ' || to_char(durationPropertyId,'TM9') || ')';

    -- compute the difference (in minutes) between the two Oracle
    -- timestamps from our sqlStmt query.  Store the minutes as
    -- xsd:integer.
    insertStmt :=
      'insert /*+ parallel append */ into ' || output_tab || ' (sid, pid, o)
       select eventId,
              ' || to_char(durationPropertyId,'TM9') || ',
              ''"'' || minutes || ''"^^xsd:integer''
       from   (
         select eventId,
                (extract(day    from (endTime - startTime))*24*60 +
                 extract(hour   from (endTime - startTime))*60 +
                 extract(minute from (endTime - startTime))) minutes
         from   (' || sqlStmt || '))';

    -- execute the query
    execute immediate insertStmt;

    -- commit our changes
    commit;
  end if;

  -- we already checked for duplicates in src_tab_view (NEWDATA) and
  -- in output_tab (UNIQDATA)
  optimization_flag := SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_NEWDATA_ONLY +
                       SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_UNIQDATA_ONLY;

  -- return true to indicate success
  return true;
```

```
  -- handle any exceptions
  exception
    when others then
      diag_message := 'error occurred: ' || SQLERRM;
      return false;
end sem_inf_durations;
/
show errors;
```

The `sem_inf_durations` function leverages built-in Oracle temporal functions to compute the event durations. First, the function converts the `xsd:dateTime` literal value to an Oracle `TIMESTAMP` object using the `TO_TIMESTAMP_TZ` function. Taking the difference between two Oracle `TIMESTAMP` objects produces an `INTERVAL` object that represents a time interval. Using the `EXTRACT` operator, the `sem_inf_durations` function computes the duration of each event in minutes by extracting the days, hours, and minutes out of the duration intervals.

Because the `sem_inf_durations` function checks for duplicates against both data in the existing model (`src_tab_view`) and data in the `output_tab` table, it can enable the `INF_EXT_OPT_FLAG_NEWDATA_ONLY` and `INF_EXT_OPT_FLAG_UNIQDATA_ONLY` optimization flags. (See Example 3: Optimizing Performance (page 9-10) for more information about optimization flags.)

Notice that unlike previous examples, `sem_inf_durations` contains an exception handler. Exception handlers are useful for debugging issues in user-defined inference extension functions. To produce useful debugging messages, catch exceptions in the extension function, set the `diag_message` parameter to reflect the error, and return `FALSE` to indicate that an error occurred during execution of the extension function. The `sem_inf_durations` function catches all exceptions and sets the `diag_message` value to the exception message.

To create an entailment with the `sem_inf_durations` function, grant execution privileges to the MDSYS user, then pass the function name to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, as follows:

```
-- grant appropriate privileges
grant execute on sem_inf_durations to mdsys;

-- create the entailment
begin
  sem_apis.create_entailment(
    'EVENT_INF'
  , sem_models('EVENT', 'EVENT_ONT')
  , sem_rulebases('OWLPRIME')
  , passes => SEM_APIS.REACH_CLOSURE
  , inf_ext_user_func_name => 'sem_inf_durations'
);
end;
/
```

In addition to the triples inferred by OWLPRIME, the entailment should contain the following three new triples added by `sem_inf_durations`:

```
S                           P                                    O
--------------------------- ------------------------------------ ---------
http://example.org/event/m1 http://example.org/event/lengthInMins 90
http://example.org/event/m2 http://example.org/event/lengthInMins 60
http://example.org/event/p1 http://example.org/event/lengthInMins 120
```

9.1.3.4.2 Example 4b: Overlap Rule

The following user-defined inference extension function (`sem_inf_overlap`) infers whether two events overlap. Two events overlap if one event starts while the other event is in progress. The function extracts the start and end times for every pair of events, converts the `xsd:dateTime` values into Oracle timestamps, then computes whether one event starts within the other.

```
create or replace function sem_inf_overlap(
    src_tab_view         in  varchar2,
    resource_id_map_view in  varchar2,
    output_tab           in  varchar2,
    action               in  varchar2,
    num_calls            in  number,
    tplInferredLastRound in  number,
    options              in  varchar2 default null,
    optimization_flag    out number,
    diag_message         out varchar2
    )
return boolean
as
  eventClassId        number;
  rdfTypePropertyId   number;
  startTimePropertyId number;
  endTimePropertyId   number;
  overlapsPropertyId  number;
  noOverlapPropertyId number;

  xsdTimeFormat       varchar2(100);
  sqlStmt             varchar2(4000);
  insertStmt          varchar2(4000);

  pragma autonomous_transaction;
begin
  if (action = 'RUN') then
    -- retrieve ID of resource that already exists in the data (will
    -- throw exception if resource does not exist).
    eventClassId        := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/event/Event');
    startTimePropertyId := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/event/startTime');
    endTimePropertyId   := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/event/endTime');
    overlapsPropertyId  := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/event/overlaps');
    noOverlapPropertyId := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/event/noOverlap');
    rdfTypePropertyId   := sdo_sem_inference.oracle_orardf_res2vid('http://
www.w3.org/1999/02/22-rdf-syntax-ns#type');

    -- set the TIMESTAMP format we will use to parse XSD times
    xsdTimeFormat := 'YYYY-MM-DD"T"HH24:MI:SSTZH:TZM';

    -- query we use to extract the event ID and start/end times.
    sqlStmt :=
      'select idsA1.sid eventAId,
              idsB1.sid eventBId,
              TO_TIMESTAMP_TZ(valuesA1.value_name,''YYYY-MM-
DD"T"HH24:MI:SSTZH:TZM'') startTimeA,
              TO_TIMESTAMP_TZ(valuesA2.value_name,''YYYY-MM-
```

```
DD"T"HH24:MI:SSTZH:TZM'') endTimeA,
             TO_TIMESTAMP_TZ(valuesB1.value_name,''YYYY-MM-
DD"T"HH24:MI:SSTZH:TZM'') startTimeB,
             TO_TIMESTAMP_TZ(valuesB2.value_name,''YYYY-MM-
DD"T"HH24:MI:SSTZH:TZM'') endTimeB
       from   ' || resource_id_map_view || ' valuesA1,
              ' || resource_id_map_view || ' valuesA2,
              ' || resource_id_map_view || ' valuesB1,
              ' || resource_id_map_view || ' valuesB2,
              ' || src_tab_view || '         idsA1,
              ' || src_tab_view || '         idsA2,
              ' || src_tab_view || '         idsA3,
              ' || src_tab_view || '         idsB1,
              ' || src_tab_view || '         idsB2,
              ' || src_tab_view || '         idsB3
       where  idsA1.sid = idsA3.sid
         AND  idsA3.pid = ' || to_char(rdfTypePropertyId,'TM9') || '
         AND  idsA3.oid = ' || to_char(eventClassId,'TM9')      || '
         AND  idsB1.sid = idsB3.sid
         AND  idsB3.pid = ' || to_char(rdfTypePropertyId,'TM9') || '
         AND  idsB3.oid = ' || to_char(eventClassId,'TM9')      || '
       /* only do half the checks, our TBOX ontology will handle symmetries */
         AND  idsA1.sid < idsB1.sid
       /* grab values of startTime and endTime for event A */
         AND  idsA1.sid = idsA2.sid
         AND  idsA1.pid = ' || to_char(startTimePropertyId,'TM9') || '
         AND  idsA2.pid = ' || to_char(endTimePropertyId,'TM9')   || '
         AND  idsA1.oid = valuesA1.value_id
         AND  idsA2.oid = valuesA2.value_id
       /* grab values of startTime and endTime for event B */
         AND  idsB1.sid = idsB2.sid
         AND  idsB1.pid = ' || to_char(startTimePropertyId,'TM9') || '
         AND  idsB2.pid = ' || to_char(endTimePropertyId,'TM9')   || '
         AND  idsB1.oid = valuesB1.value_id
         AND  idsB2.oid = valuesB2.value_id
       /* ensures we have NEWDATA */
         AND  not exists
               (select 1
                from   ' || src_tab_view || '
                where  sid = idsA1.sid
                  AND  oid = idsB1.sid
                  AND  pid in (' || to_char(overlapsPropertyId,'TM9')  || ',' ||
                                    to_char(noOverlapPropertyId,'TM9') || '))
       /* ensures we have UNIQDATA */
         AND  not exists
               (select 1
                from   ' || output_tab   || '
                where  sid = idsA1.sid
                  AND  oid = idsB1.sid
                  AND  pid in (' || to_char(overlapsPropertyId,'TM9')  || ',' ||
                                    to_char(noOverlapPropertyId,'TM9') || '))';

   -- compare the two event times
   insertStmt :=
     'insert /*+ parallel append */ into ' || output_tab || ' (sid, pid, oid)
      select eventAId, overlapStatusId, eventBId
      from   (
        select eventAId,
               (case
                when (startTimeA < endTimeB and
                      startTimeA > startTimeB) then
```

```
                  ' || to_char(overlapsPropertyId,'TM9') || '
                when (startTimeB < endTimeA and
                      startTimeB > startTimeA) then
                  ' || to_char(overlapsPropertyId,'TM9') || '
                else
                  ' || to_char(noOverlapPropertyId,'TM9') || '
                end) overlapStatusId,
                eventBId
        from   (' || sqlStmt || '))';

    -- execute the query
    execute immediate insertStmt;

    -- commit our changes
    commit;
  end if;

  -- we only use ID values in the output_tab and we check for
  -- duplicates with our NOT EXISTS clause.
  optimization_flag := SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_ALL_IDS +
                       SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_NEWDATA_ONLY +
                       SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_UNIQDATA_ONLY;

  -- return true to indicate success
  return true;

  -- handle any exceptions
  exception
    when others then
      diag_message := 'error occurred: ' || SQLERRM;
      return false;
end sem_inf_overlap;
/
show errors;
```

The `sem_inf_overlap` function is similar to the `sem_inf_durations` function in Example 4b: Overlap Rule (page 9-17). The main difference between the two is that the query in `sem_inf_overlap` contains more joins and enables the INF_EXT_OPT_FLAG_ALL_IDS optimization flag because it does not need to generate new lexical values. (See Example 3: Optimizing Performance (page 9-10) for more information about optimization flags.)

To create an entailment with the `sem_inf_overlap` function, grant execution privileges to the MDSYS user, then pass the function name to the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure, as follows:

```
-- grant appropriate privileges
grant execute on sem_inf_overlap to mdsys;

-- create the entailment
begin
  sem_apis.create_entailment(
    'EVENT_INF'
  , sem_models('EVENT', 'EVENT_ONT')
  , sem_rulebases('OWLPRIME')
  , passes => SEM_APIS.REACH_CLOSURE
  , inf_ext_user_func_name => 'sem_inf_overlap'
);
end;
/
```

In addition to the triples inferred by OWLPRIME, the entailment should contain the following six new triples added by `sem_inf_overlap`:

```
S                           P                                   O
--------------------------- ----------------------------------
---------------------------
http://example.org/event/m1  http://example.org/event/noOverlap  http://example.org/
event/m2
http://example.org/event/m1  http://example.org/event/noOverlap  http://example.org/
event/p1
http://example.org/event/m2  http://example.org/event/noOverlap  http://example.org/
event/m1
http://example.org/event/m2  http://example.org/event/overlaps   http://example.org/
event/p1
http://example.org/event/p1  http://example.org/event/noOverlap  http://example.org/
event/m1
http://example.org/event/p1  http://example.org/event/overlaps   http://example.org/
event/m2
```

9.1.3.4.3 Example 4c: Duration and Overlap Rules

The example in this section uses the extension functions from Example 4a: Duration Rule (page 9-14) (`sem_inf_durations`) and Example 4b: Overlap Rule (page 9-17) (`sem_inf_overlap`) together to produce a single entailment. The extension functions are left unmodified for this example.

To create an entailment using multiple extension functions, use a comma to separate each extension function passed to the `inf_ext_user_func_name` parameter of SEM_APIS.CREATE_ENTAILMENT (page 11-15). The following example assumes that the MDSYS user has already been granted the appropriate privileges on the extension functions.

```
-- use multiple user-defined inference functions
begin
  sem_apis.create_entailment(
    'EVENT_INF'
  , sem_models('EVENT', 'EVENT_ONT')
  , sem_rulebases('OWLPRIME')
  , passes => SEM_APIS.REACH_CLOSURE
  , inf_ext_user_func_name => 'sem_inf_durations,sem_inf_overlap'
);
end;
/
```

In addition to the triples inferred by OWLPRIME, the entailment should contain the following nine new triples added by `sem_inf_durations` and `sem_inf_overlap`:

```
S                           P                                    O
--------------------------- -----------------------------------
---------------------------
http://example.org/event/m1  http://example.org/event/lengthInMins  90
http://example.org/event/m1  http://example.org/event/noOverlap     http://
example.org/event/m2
http://example.org/event/m1  http://example.org/event/noOverlap     http://
example.org/event/p1
http://example.org/event/m2  http://example.org/event/lengthInMins  60
http://example.org/event/m2  http://example.org/event/noOverlap     http://
example.org/event/m1
http://example.org/event/m2  http://example.org/event/overlaps      http://
example.org/event/p1
http://example.org/event/p1  http://example.org/event/lengthInMins  120
http://example.org/event/p1  http://example.org/event/noOverlap     http://
```

```
example.org/event/m1
http://example.org/event/p1  http://example.org/event/overlaps      http://
example.org/event/m2
```

Notice that the extension functions, `sem_inf_durations` and `sem_inf_overlap`, did not need to use the same optimization flags. It is possible to use extension functions with contradictory optimization flags (for example, one function using `INF_EXT_OPT_FLAG_ALL_IDS` and another function inserting all new triples as lexical values).

9.1.3.5 Example 5: Spatial Reasoning

User-defined inference extension functions can also leverage geospatial data types, like WKT (well-known text), to perform spatial reasoning. For example, with user-defined extension functions, it is possible to infer a "contains" relationship between geometric entities, such as states and cities.

The example in this section demonstrates how to infer whether a geometry (a US state) contains a point (a US city). This example assumes the RDF network already has a spatial index (described in section 1.6.6.2). This example also assumes the model `STATES` exists and contains the following semantic data:

```
@prefix orageo: <http://xmlns.oracle.com/rdf/geo/> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix :       <http://example.org/geo/> .

:Colorado rdf:type  :State ;
          :boundary "Polygon((-109.0448 37.0004, -102.0424 36.9949, -102.0534
41.0006, -109.0489 40.9996, -109.0448 37.0004))"^^orageo:WKTLiteral .
:Utah     rdf:type  :State ;
          :boundary "Polygon((-114.0491 36.9982, -109.0462 37.0026, -109.0503
40.9986, -111.0471 41.0006, -111.0498 41.9993, -114.0395 41.9901, -114.0491
36.9982))"^^orageo:WKTLiteral .
:Wyoming  rdf:type  :State ;
          :boundary "Polygon((-104.0556 41.0037, -104.0584 44.9949, -111.0539
44.9998, -111.0457 40.9986, -104.0556 41.0037))"^^orageo:WKTLiteral

:StateCapital rdfs:subClassOf :City ;

:Denver   rdf:type  :StateCapital ;
          :location "Point(-104.984722 39.739167)"^^orageo:WKTLiteral .
:SaltLake rdf:type  :StateCaptial ;
          :location "Point(-111.883333 40.75)"^^orageo:WKTLiteral .
:Cheyenne rdf:type  :StateCapital ;
          :location "Point(-104.801944 41.145556)"^^orageo:WKTLiteral .
```

The following user-defined inference extension function (`sem_inf_capitals`) searches for capital cities within each state using the WKT geometries. If the function finds a capital city, it infers the city is the capital of the state containing it.

```
create or replace function sem_inf_capitals(
    src_tab_view         in  varchar2,
    resource_id_map_view in  varchar2,
    output_tab           in  varchar2,
    action               in  varchar2,
    num_calls            in  number,
    tplInferredLastRound in  number,
    options              in  varchar2 default null,
    optimization_flag    out number,
    diag_message         out varchar2
```

```
    )
return boolean
as
  stateClassId        number;
  capitalClassId      number;

  boundaryPropertyId  number;
  locationPropertyId  number;
  rdfTypePropertyId   number;
  capitalPropertyId   number;

  defaultSRID         number := 8307;

  xsdTimeFormat       varchar2(100);
  sqlStmt             varchar2(4000);
  insertStmt          varchar2(4000);

  pragma autonomous_transaction;
begin
  if (action = 'RUN') then
    -- retrieve ID of resource that already exists in the data (will
    -- throw exception if resource does not exist).
    stateClassId       := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/geo/State');
    capitalClassId     := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/geo/StateCapital');
    boundaryPropertyId := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/geo/boundary');
    locationPropertyId := sdo_sem_inference.oracle_orardf_res2vid('http://
example.org/geo/location');
    rdfTypePropertyId  := sdo_sem_inference.oracle_orardf_res2vid('http://www.w3.org/
1999/02/22-rdf-syntax-ns#type');

    -- retreive ID of resource or generate a new ID if resource does
    -- not already exist
    capitalPropertyId := sdo_sem_inference.oracle_orardf_add_res('http://
example.org/geo/capital');

    -- query we use to extract the capital cities contained within state boundaries
    sqlStmt :=
      'select idsA1.sid stateId,
              idsB1.sid cityId
       from   ' || resource_id_map_view || ' valuesA,
              ' || resource_id_map_view || ' valuesB,
              ' || src_tab_view || '         idsA1,
              ' || src_tab_view || '         idsA2,
              ' || src_tab_view || '         idsB1,
              ' || src_tab_view || '         idsB2
       where  idsA1.pid = ' || to_char(rdfTypePropertyId,'TM9') || '
         AND  idsA1.oid = ' || to_char(stateClassId,'TM9')      || '
         AND  idsB1.pid = ' || to_char(rdfTypePropertyId,'TM9') || '
         AND  idsB1.oid = ' || to_char(capitalClassId,'TM9')    || '
       /* grab geometric lexical values */
         AND  idsA2.sid = idsA1.sid
         AND  idsA2.pid = ' || to_char(boundaryPropertyId,'TM9')|| '
         AND  idsA2.oid = valuesA.value_id
         AND  idsB2.sid = idsB1.sid
         AND  idsB2.pid = ' || to_char(locationPropertyId,'TM9')|| '
         AND  idsB2.oid = valuesB.value_id
       /* compare geometries to see if city is contained by state */
         AND  SDO_RELATE(
```

```
                SDO_RDF.getV$GeometryVal(
                  valuesA.value_type,
                  valuesA.vname_prefix,
                  valuesA.vname_suffix,
                  valuesA.literal_type,
                  valuesA.language_type,
                  valuesA.long_value,
                  ' || to_char(defaultSRID,'TM9') || '),
                SDO_RDF.getV$GeometryVal(
                  valuesB.value_type,
                  valuesB.vname_prefix,
                  valuesB.vname_suffix,
                  valuesB.literal_type,
                  valuesB.language_type,
                  valuesB.long_value,
                  ' || to_char(defaultSRID,'TM9') || '),
                ''mask=CONTAINS'') = ''TRUE''
       /* ensures we have NEWDATA and only check capitals not assigned to a state */
         AND  not exists
              (select 1
               from   ' || src_tab_view || '
               where  pid = ' || to_char(capitalPropertyId,'TM9') || '
                 AND  (sid = idsA1.sid OR oid = idsB1.sid))
       /* ensures we have UNIQDATA and only check capitals not assigned to a state */
         AND  not exists
              (select 1
               from   ' || output_tab   || '
               where  pid = ' || to_char(capitalPropertyId,'TM9') || '
                 AND  (sid = idsA1.sid OR oid = idsB1.sid))';

    -- insert new triples using only IDs
    insertStmt :=
      'insert /*+ parallel append */ into ' || output_tab || ' (sid, pid, oid)
       select stateId, ' || to_char(capitalPropertyId,'TM9') || ', cityId
       from   (' || sqlStmt || ')';

    -- execute the query
    execute immediate insertStmt;

    -- commit our changes
    commit;
  end if;

  -- we only use ID values in the output_tab and we check for
  -- duplicates with our NOT EXISTS clauses.
  optimization_flag := SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_ALL_IDS +
                       SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_NEWDATA_ONLY +
                       SDO_SEM_INFERENCE.INF_EXT_OPT_FLAG_UNIQDATA_ONLY;

  -- return true to indicate success
  return true;

  -- handle any exceptions
  exception
    when others then
      diag_message := 'error occurred: ' || SQLERRM;
      return false;
end sem_inf_capitals;
/
show errors;
```

The sem_inf_capitals function is similar to the sem_inf_durations function in Example 4a: Duration Rule (page 9-14), in that both functions must convert the lexical values of some triples into Oracle types to leverage native Oracle operators. In the case of sem_inf_capitals, the function converts the WKT lexical values encoding polygons and points into the Oracle Spatial and Graph SDO_GEOMETRY type, using the SDO_RDF.getV$GeometryVal function. The getV$GeometryVal function requires arguments mostly provided by the resource view (resource_id_map_view) and an additional argument, an ID to a spatial reference system (SRID). The getV$GeometryVal function will convert the geometry into the spatial reference system specified by SRID. The sem_inf_capitals function uses the default Oracle Spatial and Graph reference system, WGS84 Longitude-Latitude, specified by SRID value 8307. (For more information about support in RDF Semantic Graph for spatial references systems, see Spatial Support (page 1-66).)

After converting the WKT values into SDO_GEOMETRY types using the getV $GeometryVal function, the sem_inf_capitals function compares the state geometry with the city geometry to see if the state contains the city. The SDO_RELATE operator performs this comparison and returns the literal value 'TRUE' when the state contains the city. The SDO_RELATE operator can perform various different types of comparisons. (See *Oracle Spatial and Graph Developer's Guide* for more information about SDO_RELATE and other spatial operators.)

To create an entailment with the sem_inf_capitals function, grant execution privileges to the MDSYS user, then pass the function name to the SEM_APIS.CREATE_ENTAILMENT (page 11-15)procedure, as follows:

```
-- grant appropriate privileges
grant execute on sem_inf_capitals to mdsys;

-- create the entailment
begin
  sem_apis.create_entailment(
    'STATES_INF'
  , sem_models('STATES')
  , sem_rulebases('OWLPRIME')
  , passes => SEM_APIS.REACH_CLOSURE
  , inf_ext_user_func_name => 'sem_inf_capitals'
);
end;
/
```

In addition to the triples inferred by OWLPRIME, the entailment should contain the following three new triples added by sem_inf_capitals:

```
S                              P                              O
------------------------------ ------------------------------
------------------------------
http://example.org/geo/Colorado  http://example.org/geo/capital  http://
example.org/geo/Denver
http://example.org/geo/Utah      http://example.org/geo/capital  http://
example.org/geo/SaltLake
http://example.org/geo/Wyoming   http://example.org/geo/capital  http://
example.org/geo/Cheyenne
```

9.1.3.6 Example 6: Calling a Web Service

This section contains a user-defined inference extension function (sem_inf_geocoding) and a related helper procedure (geocoding), which enable you to make a web service call to the Oracle Geocoder service. The user-defined inference extension function looks for the object values of triples using predicate

<urn:streetAddress>, makes callouts to the Oracle public Geocoder service endpoint at http://maps.oracle.com/geocoder/gcserver, and inserts the longitude and latitude information as two separate triples.

For example, assume that the semantic model contains the following assertion:

```
<urn:NEDC>  <urn:streetAddress>  "1 Oracle Dr., Nashua, NH"
```

In this case, an inference call using sem_inf_geocoding will produce the following new assertions:

```
<urn:NEDC> <http://www.w3.org/2003/01/geo/wgs84_pos#long>  "-71.46421"
<urn:NEDC> <http://www.w3.org/2003/01/geo/wgs84_pos#lat>   "42.75836"
<urn:NEDC> <http://www.opengis.net/geosparql#asWKT>  "POINT(-71.46421
42.75836)"^^<http://www.opengis.net/geosparql#wktLiteral>
<urn:NEDC> <http://xmlns.oracle.com/rdf/geo/asWKT>   "POINT(-71.46421
42.75836)"^^<http://xmlns.oracle.com/rdf/geo/WKTLiteral>
```

The sem_inf_geocoding function is defined as follows:

```
create or replace function sem_inf_geocoding(
    src_tab_view         in  varchar2,
    resource_id_map_view in  varchar2,
    output_tab           in  varchar2,
    action               in  varchar2,
    num_calls            in  number,
    tplInferredLastRound in  number,
    options              in  varchar2 default null,
    optimization_flag    out number,
    diag_message         out varchar2
    )
return boolean
as
  pragma autonomous_transaction;
  iCount integer;

  nLong number;
  nLat  number;
  nWKT  number;
  nOWKT number;
  nStreetAddr number;

  sidTab    dbms_sql.number_table;
  oidTab    dbms_sql.number_table;

  vcRequestBody varchar2(32767);
  vcStmt        varchar2(32767);
  vcStreeAddr   varchar2(3000);

  type cur_type is ref cursor;
  cursorFind    cur_type;
  vcLong varchar2(100);
  vcLat  varchar2(100);
begin
  if (action = 'START') then
    nLat := sdo_sem_inference.oracle_orardf_add_res('http://www.w3.org/2003/01/geo/
wgs84_pos#lat');
    nLong := sdo_sem_inference.oracle_orardf_add_res('http://www.w3.org/2003/01/geo/
wgs84_pos#long');
    nWKT  := sdo_sem_inference.oracle_orardf_add_res('http://www.opengis.net/
geosparql#asWKT');
    nOWKT := sdo_sem_inference.oracle_orardf_add_res('http://
```

```
xmlns.oracle.com/rdf/geo/asWKT');
  end if;

  if (action = 'RUN') then
    nStreetAddr := sdo_sem_inference.oracle_orardf_res2vid('<urn:streetAddress>');
    nLat := sdo_sem_inference.oracle_orardf_res2vid('http://www.w3.org/2003/01/geo/
wgs84_pos#lat');
    nLong := sdo_sem_inference.oracle_orardf_res2vid('http://www.w3.org/2003/01/geo/
wgs84_pos#long');
    nWKT  := sdo_sem_inference.oracle_orardf_res2vid('http://www.opengis.net/
geosparql#asWKT');
    nOWKT := sdo_sem_inference.oracle_orardf_res2vid('http://
xmlns.oracle.com/rdf/geo/asWKT');

    vcStmt := '
      select /*+ parallel */ distinct s1.sid as s_id, s1.oid as o_id
        from ' || src_tab_view || ' s1
       where s1.pid = :1
         and not exists ( select 1
                            from   ' || src_tab_view || ' x
                           where  x.sid = s1.sid
                             and  x.pid = :2
                        ) ';
    open cursorFind for vcStmt using nStreetAddr, nLong;

    loop
      fetch cursorFind bulk collect into sidTab, oidTab limit 10000;
      for i in 1..sidTab.count loop
        vcStreeAddr := sdo_sem_inference.oracle_orardf_vid2lit(oidTab(i));
        -- dbms_output.put_line('Now processing street addr ' || vcStreeAddr);
        geocoding(vcStreeAddr, vcLong, vcLat);
        execute immediate 'insert into ' || output_tab || '(sid,pid,oid,gid,s,p,o,g)
            values(:1, :2, null, null, null, null, :3, null) '
            using sidTab(i), nLong, '"'||vcLong||'"';
        execute immediate 'insert into ' || output_tab || '(sid,pid,oid,gid,s,p,o,g)
            values(:1, :2, null, null, null, null, :3, null) '
            using sidTab(i), nLat, '"'||vcLat||'"';
        execute immediate 'insert into ' || output_tab || '(sid,pid,oid,gid,s,p,o,g)
            values(:1, :2, null, null, null, null, :3, null) '
            using sidTab(i), nWKT, '"POINT('|| vcLong || ' ' ||vcLat ||')"^^<http://
www.opengis.net/geosparql#wktLiteral>';
        execute immediate 'insert into ' || output_tab || '(sid,pid,oid,gid,s,p,o,g)
            values(:1, :2, null, null, null, null, :3, null) '
            using sidTab(i), nOWKT, '"POINT('|| vcLong || ' ' ||vcLat ||')"^^<http://
xmlns.oracle.com/rdf/geo/WKTLiteral>';
      end loop;
      exit when cursorFind%notfound;
    end loop;
    commit;
  end if;
  return true;
end;
/
grant execute on sem_inf_geocoding to mdsys;
```

The `sem_inf_geocoding` function makes use of the following helper procedure named `geocoding`, which does the actual HTTP communication with the Geocoder web service endpoint. Note that proper privileges are required to connect to the web server.

```
create or replace procedure geocoding(addr varchar2,
                                      vcLong out varchar2,
                                      vcLat  out varchar2
                                     )
as
  httpReq  utl_http.req;
  httpResp utl_http.resp;

  vcRequestBody varchar2(32767);

  vcBuffer  varchar2(32767);
  idxLat integer;
  idxLatEnd integer;
begin
  vcRequestBody := utl_url.escape('xml_request=<?xml version="1.0" standalone="yes"?>
    <geocode_request vendor="elocation">
      <address_list>
      <input_location id="27010">
      <input_address match_mode="relax_street_type">
        <unformatted country="US">
           <address_line value="'|| addr ||'"/>
        </unformatted>
       </input_address>
      </input_location>
    </address_list>
    </geocode_request>
  ');
  dbms_output.put_line('request ' || vcRequestBody);

  -- utl_http.set_proxy('<your_proxy_here_if_necessary>', null);
  httpReq := utl_http.begin_request (
    'http://maps.oracle.com/geocoder/gcserver', 'POST');

  utl_http.set_header(httpReq, 'Content-Type', 'application/x-www-form-urlencoded');
  utl_http.set_header(httpReq, 'Content-Length', lengthb(vcRequestBody));

  utl_http.write_text(httpReq, vcRequestBody);

  httpResp := utl_http.get_response(httpReq);

  utl_http.read_text(httpResp, vcBuffer, 32767);
  utl_http.end_response(httpResp);

  -- dbms_output.put_line('response ' || vcBuffer);
  -- Here we are doing some simple string parsing out of an XML.
  -- It is more robust to use XML functions instead.
  idxLat := instr(vcBuffer, 'longitude="');
  idxLatEnd := instr(vcBuffer, '"', idxLat + 12);
  vcLong := substr(vcBuffer, idxLat + 11, idxLatEnd - idxLat - 11);
  dbms_output.put_line('long = ' || vcLong);

  idxLat := instr(vcBuffer, 'latitude="');
  idxLatEnd := instr(vcBuffer, '"', idxLat + 11);
  vcLat := substr(vcBuffer, idxLat + 10, idxLatEnd - idxLat - 10);
  dbms_output.put_line('lat = ' || vcLat);
exception
  when others then
    dbms_output.put_line('geocoding: error ' || dbms_utility.format_error_backtrace
|| ' '
                                             || dbms_utility.format_error_stack);
```

```
end;
/
```

9.2 User-Defined Functions and Aggregates

The RDF Semantic Graph query extension architecture enables you to add user-defined functions and aggregates to be used in SPARQL queries, both through the SEM_MATCH table function and through the support for Apache Jena and OpenRDF Sesame.

The SPARQL 1.1 Standard provides several functions used mainly for filtering and categorizing data obtained by a query. However, you may need specialized functions not supported by the standard.

Some simple examples include finding values that belong to a specific type, or obtaining values with a square sum value that is greater than a certain threshold. Although this can be done by means of combining functions, it may be useful to have a single function that handles the calculations, which also allows for a simpler and shorter query.

The RDF Semantic Graph query extension allows you to include your own query functions and aggregates. This architecture allows:

- Custom query functions that can be used just like built-in SPARQL query functions, as explained in API Support for User-Defined Functions (page 9-29)
- Custom aggregates that can be used just like built-in SPARQL aggregates, as explained in API Support for User-Defined Aggregates (page 9-32)

9.2.1 Data Types for User-Defined Functions and Aggregates

The SDO_RDF_TERM object type is used to represent an RDF term when creating user-defined functions and aggregates.

SDO_RDF_TERM has the following attributes, which correspond to columns in the MDSYS.RDF_VALUE$ table (see Table 1-3 (page 1-5) in Statements (page 1-5) for a description of these attributes). The CTX1 attribute is reserved for future use and does not have a corresponding column in MDSYS.RDF_VALUE$.

```
SDO_RDF_TERM(
  VALUE_TYPE   VARCHAR2(10),
  VALUE_NAME    VARCHAR2(4000),
  VNAME_PREFIX   VARCHAR2(4000),
  VNAME_SUFFIX   VARCHAR2(512),
  LITERAL_TYPE   VARCHAR2(1000),
  LANGUAGE_TYPE   VARCHAR2(80),
  LONG_VALUE   CLOB,
  CTX1   VARCHAR2(4000) )
```

The following constructors are available for creating SDO_RDF_TERM objects. The first constructor populates each attribute from a single, lexical RDF term string. The second and third constructors receive individual attribute values as input. Only the first RDF term string constructor sets values for VNAME_PREFIX and VNAME_SUFFIX. These values are initialized to null by the other constructors.

```
SDO_RDF_TERM (
  rdf_term_str  VARCHAR2)
  RETURN SELF;

SDO_RDF_TERM (
```

```
  value_type  VARCHAR2,
  value_name  VARCHAR2,
  literal_type  VARCHAR2,
  language_type  VARCHAR2,
  long_value  CLOB)
  RETURN SELF;

SDO_RDF_TERM (
  value_type  VARCHAR2,
  value_name  VARCHAR2,
  literal_type  VARCHAR2,
  language_type  VARCHAR2,
  long_value  CLOB,
  ctx1 VARCHAR2)
  RETURN SELF;
```

The SDO_RDF_TERM_LIST type is used to hold a list of SDO_RDF_TERM objects and is defined as `VARRAY(32767) of SDO_RDF_TERM`.

9.2.2 API Support for User-Defined Functions

A user-defined function is created by implementing a PL/SQL function with a specific signature, and a specific URI is used to invoke the function in a SPARQL query pattern.

After each successful inference extension function call, a commit is executed to persist changes made in the inference extension function call. If an inference extension function is defined as autonomous by specifying `pragma autonomous_transaction`, then it should either commit or roll back at the end of its implementation logic. Note that the inference engine may call an extension function multiple times when creating an entailment (once per round). Commits and rollbacks from one call will not affect other calls.

9.2.2.1 PL/SQL Function Implementation

Each user-defined function must be implemented by a PL/SQL function with a signature in the following format:

```
FUNCTION user_function_name (params IN SDO_RDF_TERM_LIST)
  RETURN SDO_RDF_TERM
```

This signature supports an arbitrary number of RDF term arguments, which are passed in using a single SDO_RDF_TERM_LIST object, and returns a single RDF term as output, which is represented as a single SDO_RDF_TERM object. Type checking or other verifications for these parameters are not performed. You should take steps to validate the data according to the function goals.

Note that PL/SQL supports callouts to functions written in other programming languages, such as C and Java, so the PL/SQL function that implements a user-defined query function can serve only as a wrapper for functions written in other programming languages.

9.2.2.2 Invoking User-Defined Functions from a SPARQL Query Pattern

After a user-defined function is implemented in PL/SQL, it can be invoked from a SPARQL query pattern using a function URI constructed from the prefix `<http://xmlns.oracle.com/rdf/extensions/>` followed by *`schema.package_name.function_name`* if the corresponding PL/SQL function is

part of a PL/SQL package, or *schema.function_name* if the function is not part of a PL/SQL package. The following are two example function URIs:

```
<http://xmlns.oracle.com/rdf/extensions/my_schema.my_package.my_function>(arg_1, …,
arg_n)

<http://xmlns.oracle.com/rdf/extensions/my_schema.my_function>(arg_1, …, arg_n)
```

9.2.2.3 User-Defined Function Examples

This section presents examples of the implementation of a user-defined function and the use of that function in a FILTER clause, in a SELECT expression, and in a BIND operation.

For the examples, assume that the following data, presented here in N-triple format, exists inside a model called MYMODEL:

```
<a>  <p>  "1.0"^^xsd:double .
<b>  <p>  "1.5"^^xsd:float .
<c>  <p>  "3"^^xsd:decimal .
<d>  <p>  "4"^^xsd:string .
```

Example 9-1 User-Defined Function to Calculate Sum of Two Squares

Example 9-1 (page 9-30) shows the implementation of a simple function that receives two values and calculates the sum of the squares of each value.

```
CREATE OR REPLACE FUNCTION sum_squares (params IN MDSYS.SDO_RDF_TERM_LIST)
   RETURN MDSYS.SDO_RDF_TERM
   AS
     retTerm    SDO_RDF_TERM;
     sqr1       NUMBER;
     sqr2       NUMBER;
     addVal     NUMBER;
     val1       SDO_RDF_TERM;
     val2       SDO_RDF_TERM;
   BEGIN
     -- Set the return value to null.
     retTerm := SDO_RDF_TERM(NULL,NULL,NULL,NULL,NULL);
     -- Obtain the data from the first two parameters.
     val1 := params(1);
     val2 := params(2);
     -- Convert the value stored in the sdo_rdf_term to number.
     -- If any exception occurs, return the null value.
     BEGIN
       sqr1 := TO_NUMBER(val1.value_name);
       sqr2 := TO_NUMBER(val2.value_name);
       EXCEPTION WHEN OTHERS THEN RETURN retTerm;
     END;
     -- Compute the square sum of both values.
       addVal := (sqr1 * sqr1) + (sqr2 * sqr2);
     -- Set the return value to the desired rdf term type.
     retTerm := SDO_RDF_TERM('LIT',to_char(addVal),
                'http://www.w3.org/2001/XMLSchema#integer','',NULL);
     - Return the new value.
     RETURN retTerm;
END;
/
SHOW ERRORS;
```

Note that the sum_squares function in Example 9-1 (page 9-30) does not verify the data type of the value received. It is intended as a demonstration only, and relies on

TO_NUMBER to obtain the numeric value stored in the VALUE_NAME field of SDO_RDF_TERM.

Example 9-2 User-Defined Function Used in a FILTER Clause

Example 9-2 (page 9-31) shows the `sum_squares` function (from Example 9-1 (page 9-30)) used in a FILTER clause.

```
SELECT s, o
FROM table(sem_match(
'SELECT  ?s ?o
 WHERE { ?s ?p ?o
 FILTER (<http://xmlns.oracle.com/rdf/extensions/schema.sum_squares>(?o,?o) > 2)}',
sem_models('MYMODEL'),null,null,null,null,''));
```

The query in Example 9-2 (page 9-31) returns the following result:

```
s                    o
-------------------- --------------------
b                    1.5
c                    3
d                    4
```

Example 9-3 User-Defined Function Used in a SELECT Expression

Example 9-3 (page 9-31) shows the `sum_squares` function (from Example 9-1 (page 9-30)) used in an expression in the SELECT clause.

```
SELECT s, o, sqr_sum
FROM table(sem_match(
'SELECT  ?s ?o
       (<http://xmlns.oracle.com/rdf/extensions/schema.sum_squares>(?o,?o) AS
        ?sqr_sum)
 WHERE { ?s ?p ?o }',
sem_models('MYMODEL'),null,null,null,null,''));
```

The query in Example 9-3 (page 9-31) returns the following result:

```
s                    o                    sqr_sum
-------------------- -------------------- --------------------
a                    1                    2
b                    1.5                  4.5
c                    3                    18
d                    4                    32
```

Example 9-4 User-Defined Function Used in a BIND Operation

Example 9-4 (page 9-31) shows the `sum_squares` function (from Example 9-1 (page 9-30)) used in a BIND operation.

```
SELECT s, o, sqr_sum
FROM table(sem_match(
'SELECT  ?s ?o ?sqr_sum
 WHERE { ?s ?p ?o .
 BIND (<http://xmlns.oracle.com/rdf/extensions/schema.sum_squares>(?o,?o) AS
       ?sqr_sum)}',
sem_models('MYMODEL'),null,null,null,null,''));
```

The query in Example 9-4 (page 9-31) returns the following result:

```
s                    o                    sqr_sum
-------------------- -------------------- --------------------
a                    1                    2
```

```
b                 1.5               4.5
c                 3                 18
d                 4                 32
```

9.2.3 API Support for User-Defined Aggregates

User-defined aggregates are implemented by defining a PL/SQL object type that implements a set of interface methods. After the user-defined aggregate is created, a specific URI is used to invoke it.

9.2.3.1 ODCIAggregate Interface

User-defined aggregates use the `ODCIAggregate` PL/SQL interface. For more detailed information about this interface, see the chapter about user-defined aggregate functions in *Oracle Database Data Cartridge Developer's Guide*.

The `ODCIAggregate` interface is implemented by a PL/SQL object type that implements four main functions:

- `ODCIAggregateInitialize`
- `ODCIAggregateIterate`
- `ODCIAggregateMerge`
- `ODCIAggregateTerminate`

As with user-defined functions (described in API Support for User-Defined Functions (page 9-29)), user-defined aggregates receive an arbitrary number of RDF term arguments, which are passed in as an SDO_RDF_TERM_LIST object, and return a single RDF term value, which is represented as an SDO_RDF_TERM object.

This scheme results in the following signatures for the PL/SQL `ODCIAggregate` interface functions (with *my_aggregate_obj_type* representing the actual object type name):

```
STATIC FUNCTION ODCIAggregateInitialize(
        sctx IN OUT my_aggregate_obj_type)
RETURN NUMBER

MEMBER FUNCTION ODCIAggregateIterate(
        self       IN OUT my_aggregate_obj_type
       ,value      IN MDSYS.SDO_RDF_TERM_LIST)
RETURN NUMBER

MEMBER FUNCTION ODCIAggregateMerge(
        self IN OUT my_aggregate_obj_type
       ,ctx2 IN     my_aggregate_obj_type)
RETURN NUMBER

MEMBER FUNCTION ODCIAggregateTerminate (
        self IN my_aggregate_obj_type
       ,return_value OUT MDSYS.SDO_RDF_TERM
       ,flags IN NUMBER)
RETURN NUMBER
```

9.2.3.2 Invoking User-Defined Aggregates

After a user-defined aggregate is implemented in PL/SQL, it can be invoked from a SPARQL query by referring to an aggregate URI constructed from the prefix

<http://xmlns.oracle.com/rdf/aggExtensions/> followed by *schema_name.aggregate_name*. The following is an example aggregate URI:

```
<http://xmlns.oracle.com/rdf/aggExtensions/schema.my_aggregate>(arg_1, …, arg_n)
```

The DISTINCT modifier can be used with user-defined aggregates, as in the following example:

```
<http://xmlns.oracle.com/rdf/aggExtensions/schema.my_aggregate>(DISTINCT arg_1)
```

In this case, only distinct argument values are passed to the aggregate. Note, however, that the DISTINCT modifier can only be used with aggregates that have exactly one argument.

9.2.3.3 User-Defined Aggregate Examples

This section presents examples of implementing and using a user-defined aggregate. For the examples, assume that the following data, presented here in N-triple format, exists inside a model called MYMODEL:

```
<a>  <p>  "1.0"^^xsd:double .
<b>  <p>  "1.5"^^xsd:float .
<c>  <p>  "3"^^xsd:decimal .
<c>  <p>  "4"^^xsd:decimal .
<d>  <p>  "4"^^xsd:string .
```

Example 9-5 User-Defined Aggregate Implementation

Example 9-5 (page 9-33) shows the implementation of a simple user-defined aggregate (countSameType). This aggregate has two arguments: the first is any RDF term, and the second is a constant data type URI. The aggregate counts how many RDF terms from the first argument position have a data type equal to the second argument.

```
-- Aggregate type creation
CREATE OR REPLACE TYPE countSameType authid current_user AS OBJECT(

count NUMBER, -- Variable to store the number of same-type terms.

-- Mandatory Functions for aggregates
STATIC FUNCTION ODCIAggregateInitialize(
        sctx IN OUT countSameType)
RETURN NUMBER,

MEMBER FUNCTION ODCIAggregateIterate(
         self       IN OUT countSameType
       , value      IN MDSYS.SDO_RDF_TERM_LIST)
RETURN NUMBER,

MEMBER FUNCTION ODCIAggregateMerge(
        self IN OUT countSameType
       ,ctx2 IN     countSameType)
RETURN NUMBER,

MEMBER FUNCTION ODCIAggregateTerminate (
        self IN countSameType
       ,return_value OUT MDSYS.SDO_RDF_TERM
       ,flags IN NUMBER)
RETURN NUMBER
);
/
SHOW ERRORS;
```

```
-- Interface function for the user-defined aggregate
CREATE OR REPLACE FUNCTION countSameAs (input MDSYS.SDO_RDF_TERM_LIST) RETURN
MDSYS.SDO_RDF_TERM
PARALLEL_ENABLE AGGREGATE USING countSameType;
/
show errors;

-- User-defined aggregate body
CREATE OR REPLACE TYPE BODY countSameType IS

STATIC FUNCTION ODCIAggregateInitialize(
         sctx            IN OUT countSameType)
RETURN NUMBER IS
BEGIN
  sctx := countSameType (0); -- Aggregate initialization
  RETURN ODCIConst.Success;
END;

MEMBER FUNCTION ODCIAggregateIterate(
         self           IN OUT countSameType
       , value          IN MDSYS.SDO_RDF_TERM_LIST )
RETURN NUMBER IS
BEGIN
  -- Increment count if the first argument has a literal type
  -- URI equal to the value of the second argument
  IF (value(1).literal_type = value(2).value_name) THEN
    self.count := self.count + 1;
  END IF;
  RETURN ODCIConst.Success;
END;

MEMBER FUNCTION ODCIAggregateMerge(
         self            IN OUT countSameType
        ,ctx2            IN countSameType)
RETURN NUMBER IS
BEGIN
  -- Sum count to merge parallel threads.
  self.count := self.count + ctx2.count;
  RETURN ODCIConst.Success;
END;

MEMBER FUNCTION ODCIAggregateTerminate(
         self            IN countSameType
        ,return_value    OUT MDSYS.SDO_RDF_TERM
        ,flags           IN NUMBER)
RETURN NUMBER IS
BEGIN
   -- Set the return value
   return_value := MDSYS.SDO_RDF_TERM('LIT',to_char(self.count),
     'http://www.w3.org/2001/XMLSchema#decimal',NULL,NULL); RETURN
ODCIConst.Success;
END;

END;
/
SHOW ERRORS;
```

Example 9-6 User-Defined Aggregate Used Without a GROUP BY Clause

Example 9-6 (page 9-34) shows the `countSameType` aggregate (from Example 9-5 (page 9-33)) used over an entire query result group.

```
FROM o
from table(sem_match(
'SELECT
 (<http://xmlns.oracle.com/rdf/aggExtensions/schema.countSameType>(?o,xsd:decimal)
  AS ?o)
 WHERE { ?s ?p ?o }',
sem_models('MYMODEL'),null,null,null,null,''));
```

The query in Example 9-6 (page 9-34) returns the following result:

```
o
--------------------
2
```

Example 9-7 User-Defined Aggregate Used With a GROUP BY Clause

Example 9-7 (page 9-35) shows the `countSameType` aggregate (from Example 9-5 (page 9-33)) used over a set of groups formed from a GROUP BY clause.

```
select s, o
from table(sem_match(
'SELECT ?s
 (<http://xmlns.oracle.com/rdf/aggExtensions/schema.countSameType>(?o,xsd:decimal)
  AS ?o)
 WHERE { ?s ?p ?o } GROUP BY ?s',
sem_models('MYMODEL'),null,null,null,null,''));
```

The query in Example 9-7 (page 9-35) returns the following result:

```
s                    o
-------------------- --------------------
a                    0
b                    0
c                    2
d                    0
```

10

RDF Views: Relational Data as RDF

This chapter explains how to create and use RDF views over relational data in Oracle Spatial and Graph RDF Semantic Graph. Relational data is viewed as virtual RDF triples using one of the two forms of RDB2RDF mapping described in W3C documents on Direct Mapping and R2RML mapping:

- *R2RML: RDB to RDF Mapping Language*, W3C Recommendation (`http://www.w3.org/TR/r2rml/`)
- *A Direct Mapping of Relational Data to RDF*, W3C Recommendation (`http://www.w3.org/TR/rdb-direct-mapping/`)

This chapter contains the following major sections:

- Why Use RDF Views on Relational Data? (page 10-1)
- API Support for RDF Views (page 10-1)
- Example: Using an RDF View with Direct Mapping (page 10-5)
- Combining Native RDF Data with Virtual RDB2RDF Data (page 10-8)

10.1 Why Use RDF Views on Relational Data?

Using RDF views on relational data enables you to integrate data available from different sources. You can exploit the advantages of relational data without the need for physical storage of the RDF triples that correspond to the relational data. Before RDF views were included in RDF Semantic Graph in Oracle Database 12*c* Release 1 (12.1), you needed to write custom SQL queries or use non-standard mappings and physically store the generated RDF triples in an RDF model.

The simplest way to create a mapping of relational data to RDF data is by calling the SEM_APIS.CREATE_RDFVIEW_MODEL (page 11-25) procedure to create an RDF view and supplying the list of tables or views whose content you would like to be viewed as RDF. This provides a direct mapping of those relational tables or views.

To get a more customized mapping, you can write an R2RML mapping document (in RDF using Turtle, for example) to specify the desired mapping, load the mapping document (after converting it to N-Triple format) into a staging table (for the table definition, see Bulk Loading Semantic Data Using a Staging Table (page 1-74)), and then call the SEM_APIS.CREATE_RDFVIEW_MODEL (page 11-25) procedure to create an RDF view by supplying the name of the staging table.

10.2 API Support for RDF Views

Subprograms are included in the SEM_APIS package (documented in SEM_APIS Package Subprograms (page 11-1)) for creating, dropping, and exporting (that is, materializing the content of) RDF views. An RDF view is created as an RDF model, but

the RDF model physically contains only metadata. The actual data is still stored in the relational tables for which the RDF view has been created.

For the examples in the rest of this section, assume that the following relational tables exist in the invoker's schema:

```
CREATE TABLE dept (
  deptno NUMBER CONSTRAINT pk_DeptTab_deptno PRIMARY KEY,
  dname VARCHAR2(30),
  loc VARCHAR2(30)
);

CREATE TABLE emp (
  empno NUMBER PRIMARY KEY,
  ename VARCHAR2(30),
  job VARCHAR2(20),
  deptno NUMBER REFERENCES dept (deptno)
);
```

Note that if these tables are in a different schema (for example, SCOTT) than the invoker's, when specifying the names of these tables, you need to use schema-qualified table names: "SCOTT"."DEPT" and "SCOTT"."EMP".

10.2.1 Creating an RDF View with Direct Mapping

Example 10-1 (page 10-2) creates an RDF view model using direct mapping of two tables, EMP and DEPT, with a base prefix of http://empdb/. The (virtual) RDF terms are generated according to *A Direct Mapping of Relational Data to RDF*, W3C Recommendation (http://www.w3.org/TR/rdb-direct-mapping/).

Example 10-1 Creating an RDF View with Direct Mapping

```
BEGIN
  sem_apis.create_rdfview_model(
    model_name => 'empdb_model',
    tables => SYS.ODCIVarchar2List('EMP', 'DEPT'),
    prefix => 'http://empdb/',
    options => 'KEY_BASED_REF_PROPERTY=T'
  );
END;
/
```

To see the properties that are generated, enter the following statement (which assumes that the objects are created in the schema of a user named TESTUSER):

```
SELECT DISTINCT p
  FROM TABLE(SEM_MATCH(
    '{?s ?p ?o}',
    SEM_Models('empdb_model'),
    NULL,
    NULL,
    NULL));

P
--------------------------------------------------------------------------------
http://empdb/TESTUSER.EMP#DEPTNO
http://empdb/TESTUSER.DEPT#LOC
http://empdb/TESTUSER.EMP#JOB
http://empdb/TESTUSER.DEPT#DEPTNO
http://empdb/TESTUSER.EMP#ENAME
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
```

```
http://empdb/TESTUSER.DEPT#DNAME
http://empdb/TESTUSER.EMP#EMPNO
http://empdb/TESTUSER.EMP#ref-DEPTNO

9 rows selected.
```

Example 10-2 Using CONFORMANCE=T

Example 10-2 (page 10-3) is essentially the same as Example 10-1 (page 10-2), but it uses the `CONFORMANCE=T` option (see the `options` parameter description for SEM_APIS.CREATE_RDFVIEW_MODEL (page 11-25)). Notice in the output that the schema name is not included in the list of properties; for example, the first output record in Example 10-2 (page 10-3) is `http://empdb/DEPT#LOC`, whereas its counterpart generated by Example 10-1 (page 10-2) is `http://empdb/TESTUSER.DEPT#LOC`.

```
BEGIN
  sem_apis.create_rdfview_model(
    model_name => 'empdb_model',
    tables => SYS.ODCIVarchar2List('EMP', 'DEPT'),
    prefix => 'http://empdb/',
    options => 'CONFORMANCE=T'
  );
END;
/

SELECT DISTINCT p
  FROM TABLE(SEM_MATCH(
    '{?s ?p ?o}',
    SEM_Models('empdb_model'),
    NULL,
    NULL,
    NULL));

P
--------------------------------------------------------------------------------
http://empdb/DEPT#LOC
http://empdb/EMP#ref-DEPTNO
http://empdb/EMP#ENAME
http://empdb/DEPT#DEPTNO
http://empdb/EMP#JOB
http://empdb/EMP#EMPNO
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://empdb/DEPT#DNAME
http://empdb/EMP#DEPTNO

9 rows selected.
```

10.2.2 Creating an RDF View with an R2RML Mapping

If you wanted to create an RDF view using the two tables EMP and DEPT, but with your own customizations, you could create an R2RML mapping document specified using Turtle, such as the following:

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix ex: <http://example.com/ns#>.

ex:TriplesMap_Dept
    rr:logicalTable [ rr:tableName "DEPT" ];
    rr:subjectMap [
```

```
        rr:template "http://data.example.com/department/{DEPTNO}";
        rr:class ex:Department;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:deptNum;
        rr:objectMap [ rr:column "DEPTNO" ; rr:datatype xsd:integer ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:deptName;
        rr:objectMap [ rr:column "DNAME" ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:deptLocation;
        rr:objectMap [ rr:column "LOC" ];
    ].

ex:TriplesMap_Emp
    rr:logicalTable [ rr:tableName "EMP" ];
    rr:subjectMap [
        rr:template "http://data.example.com/employee/{EMPNO}";
        rr:class ex:Employee;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:empNum;
        rr:objectMap [ rr:column "EMPNO" ; rr:datatype xsd:integer ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:empName;
        rr:objectMap [ rr:column "ENAME" ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:jobType;
        rr:objectMap [ rr:column "JOB" ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:worksForDeptNum;
        rr:objectMap [ rr:column "DEPTNO" ; rr:dataType xsd:integer ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:worksForDept;
        rr:objectMap [
          rr:parentTriplesMap ex:TriplesMap_Dept ;
          rr:joinCondition [ rr:child "DEPTNO"; rr:parent "DEPTNO" ]]].
```

Then, load your R2RML mapping (converted into N-Triples format) into a staging table, such as SCOTT.R2RTAB, and grant the SELECT privilege for this table to MDSYS.

Next, call SEM_APIS.CREATE_RDFVIEW_MODEL (page 11-25), as in Example 10-3 (page 10-4).

Example 10-3 Creating an RDF View with an R2RML Mapping

```
BEGIN
  sem_apis.create_rdfview_model(
    model_name => 'empdb_model',
    tables => NULL,
    r2rml_table_owner => 'SCOTT',
    r2rml_table_name => 'R2RTAB'
  );
END;
/
```

10.2.3 Dropping an RDF View

An RDF view can be dropped using the SEM_APIS.DROP_RDFVIEW_MODEL (page 11-37) procedure, as shown in Example 10-4 (page 10-5).

Example 10-4 Dropping an RDF View

```
BEGIN
  sem_apis.drop_rdfview_model(
    model_name => 'empdb_model'
  );
END;
/
```

10.2.4 Exporting Virtual Content of an RDF View into a Staging Table

The content of an RDF view is virtual; that is, the RDF triples corresponding to the underlying relational data, as mapped by direct mapping or R2RML mapping, are not materialized and stored anywhere. You may, however, want to materialize and store these virtual RDF triples in an RDF model for your testing purposes. The SEM_APIS.EXPORT_RDFVIEW_MODEL (page 11-47) subprogram lets you store the RDF triples of an RDF view in a staging table. The staging table can then be used for loading into an RDF model.

Example 10-5 (page 10-5) materializes (in N-Triples format) the content of RDF view `empdb_model` into the staging table SCOTT.RDFTAB.

Example 10-5 Exporting an RDF View

```
BEGIN
  sem_apis.export_rdfview_model(
    model_name => 'empdb_model',
    rdf_table_owner => 'SCOTT',
    rdf_table_name => 'RDFTAB'
  );
END;
```

10.3 Example: Using an RDF View with Direct Mapping

Example 10-6 (page 10-5) shows a simple workflow using an RDF view with direct mapping. In it, you:

1. Create two relational tables (EMP and DEPT).
2. Insert data into the tables.
3. Create an RDF view model (`empdb_model`) using direct mapping of the two tables.
4. Query the RDF view using SPARQL in a SEM_MATCH-based SQL query.

Example 10-7 (page 10-7) shows the output of the statements in Example 10-6 (page 10-5).

Example 10-6 Using an RDF View with Direct Mapping

```
-- Use the following relational tables.

CREATE TABLE dept (
  deptno NUMBER CONSTRAINT pk_DeptTab_deptno PRIMARY KEY,
  dname VARCHAR2(30),
```

```
  loc VARCHAR2(30)
);

CREATE TABLE emp (
  empno NUMBER PRIMARY KEY,
  ename VARCHAR2(30),
  job VARCHAR2(20),
  deptno NUMBER REFERENCES dept (deptno)
);

-- Insert some data.

INSERT INTO dept (deptno, dname, loc)
  VALUES (1, 'Sales', 'Boston');
INSERT INTO dept (deptno, dname, loc)
  VALUES (2, 'Manufacturing', 'Chicago');
INSERT INTO dept (deptno, dname, loc)
  VALUES (3, 'Marketing', 'Boston');

INSERT INTO emp (empno, ename, job, deptno)
  VALUES (1, 'Alvarez', 'SalesRep', 1);
INSERT INTO emp (empno, ename, job, deptno)
  VALUES (2, 'Baxter', 'Supervisor', 2);
INSERT INTO emp (empno, ename, job, deptno)
  VALUES (3, 'Chen', 'Writer', 3);
INSERT INTO emp (empno, ename, job, deptno)
  VALUES (4, 'Davis', 'Technician', 2);

-- Create an RDF view model using direct mapping of two tables, EMP and DEPT,
-- with a base prefix of http://empdb/.
-- Specify KEY_BASED_REF_PROPERTY=T for the options parameter.

BEGIN
  sem_apis.create_rdfview_model(
    model_name => 'empdb_model',
    tables => SYS.ODCIVarchar2List('EMP', 'DEPT'),
    prefix => 'http://empdb/',
    options => 'KEY_BASED_REF_PROPERTY=T'
  );
END;
/

-- Query an RDF view using SPARQL in a SEM_MATCH-based SQL query.
-- The next statament is a query against an RDF view named empdb_model
-- to find the employees who work for any department located in Boston.

SELECT emp
  FROM TABLE(SEM_MATCH(
  '{?emp emp:ref-DEPTNO ?dept . ?dept dept:LOC "Boston"}',
  SEM_Models('empdb_model'),
  NULL,
  SEM_ALIASES(
    SEM_ALIAS('dept','http://empdb/TESTUSER.DEPT#'),
    SEM_ALIAS('emp','http://empdb/TESTUSER.EMP#')
  ),
  null));

-- The preceding query is functionally comparable to this:
SELECT e.empno FROM emp e, dept d WHERE e.deptno = d.deptno AND d.loc = 'Boston';
```

Example 10-7 Output of Example 10-6 ***(page 10-5)***

```
SQL> -- Use the following relational tables.
SQL>
SQL> CREATE TABLE dept (
  2    deptno NUMBER CONSTRAINT pk_DeptTab_deptno PRIMARY KEY,
  3    dname VARCHAR2(30),
  4    loc VARCHAR2(30)
  5  );

Table created.

SQL>
SQL> CREATE TABLE emp (
  2    empno NUMBER PRIMARY KEY,
  3    ename VARCHAR2(30),
  4    job VARCHAR2(20),
  5    deptno NUMBER REFERENCES dept (deptno)
  6  );

Table created.

SQL>
SQL> -- Insert some data.
SQL>
SQL> INSERT INTO dept (deptno, dname, loc)
  2    VALUES (1, 'Sales', 'Boston');

1 row created.

SQL> INSERT INTO dept (deptno, dname, loc)
  2    VALUES (2, 'Manufacturing', 'Chicago');

1 row created.

SQL> INSERT INTO dept (deptno, dname, loc)
  2    VALUES (3, 'Marketing', 'Boston');

1 row created.

SQL>
SQL> INSERT INTO emp (empno, ename, job, deptno)
  2    VALUES (1, 'Alvarez', 'SalesRep', 1);

1 row created.

SQL> INSERT INTO emp (empno, ename, job, deptno)
  2    VALUES (2, 'Baxter', 'Supervisor', 2);

1 row created.

SQL> INSERT INTO emp (empno, ename, job, deptno)
  2    VALUES (3, 'Chen', 'Writer', 3);

1 row created.

SQL> INSERT INTO emp (empno, ename, job, deptno)
  2    VALUES (4, 'Davis', 'Technician', 2);

1 row created.
```

```
SQL>
SQL> -- Create an RDF view model using direct mapping of two tables, EMP and DEPT,
SQL> -- with a base prefix of http://empdb/.
SQL> -- Specify KEY_BASED_REF_PROPERTY=T for the options parameter.
SQL>
SQL> BEGIN
  2    sem_apis.create_rdfview_model(
  3       model_name => 'empdb_model',
  4       tables => SYS.ODCIVarchar2List('EMP', 'DEPT'),
  5       prefix => 'http://empdb/',
  6       options => 'KEY_BASED_REF_PROPERTY=T'
  7    );
  8  END;
  9  /

PL/SQL procedure successfully completed.

SQL>
SQL> -- Query an RDF view using SPARQL in a SEM_MATCH-based SQL query.
SQL> -- The next statament is a query against an RDF view named empdb_model
SQL> -- to find the employees who work for any department located in Boston.
SQL>
SQL> SELECT emp
  2    FROM TABLE(SEM_MATCH(
  3       '{?emp emp:ref-DEPTNO ?dept . ?dept dept:LOC "Boston"}',
  4       SEM_Models('empdb_model'),
  5       NULL,
  6       SEM_ALIASES(
  7         SEM_ALIAS('dept','http://empdb/TESTUSER.DEPT#'),
  8         SEM_ALIAS('emp','http://empdb/TESTUSER.EMP#')
  9       ),
 10       null));

EMP
--------------------------------------------------------------------------------
http://empdb/TESTUSER.EMP/EMPNO=1
http://empdb/TESTUSER.EMP/EMPNO=3

SQL>
SQL> -- The preceding query is functionally comparable to this:
SQL> SELECT e.empno FROM emp e, dept d WHERE e.deptno = d.deptno AND d.loc =
'Boston';

     EMPNO
----------
         1
         3
```

10.4 Combining Native RDF Data with Virtual RDB2RDF Data

You can combine native triple data with virtual RDB2RDF triple data in a single SEM_MATCH query by means of the SERVICE keyword (explained in Graph Patterns: Support for SPARQL 1.1 Federated Query (page 1-61)). The SERVICE keyword is overloaded through the use of special SERVICE URLs that signify local (virtual) RDF data. The following prefixes are used to denote special SERVICE URLs:

- Native models - `oram`: <http://xmlns.oracle.com/models/>
- Native virtual models - `oravm`: <http://xmlns.oracle.com/virtual_models/>

- RDB2RDF models - `orardbm`: <http://xmlns.oracle.com/rdb_models/>

Example 10-8 Querying Multiple Data Sets

Example 10-8 (page 10-9) queries multiple data sets. In this query, the first triple pattern `{ ?x rdf:type :Person }` will go against native model `m1` as usual, but `{ ?x :name ?name }` will go against the local native model `m2`, and `{ ?x :email ?email }` will go against the local RDB2RDF model `rdfview1`.

```
select * from table (sem_match(
'SELECT ?x ?name ?email
 WHERE {
  ?x rdf:type :Person .
  OPTIONAL { SERVICE oram:m2 { ?x :name ?name } }
  OPTIONAL { SERVICE orardbm:rdfview1 { ?x :email ?email } }
}'
sem_models('m1'), null, null, null, null, ' '));
```

Overloaded SERVICE use is only allowed with a single model specified in the `models` argument of SEM_MATCH. Overloaded SERVICE queries do not allow multiple models or a rulebase as input. A virtual model that contains multiple models and/or entailments should be used instead for such combinations. In addition, the `index_status` argument for SEM_MATCH will only check the entailment contained in the virtual model passed as input in the models parameter. This means the status of entailments that are referenced in overloaded SERVICE calls will not be checked.

Example 10-9 Querying Virtual RDB2RDF Data and Native RDF Data

Example 10-9 (page 10-9) queries two data sets: the `empdb_model` from Example 10-6 (page 10-5) and a native model named `people`.

```
-- Create native model people --
create table atab (gval varchar2(4000), tri sdo_rdf_triple_s);

execute sem_apis.create_sem_model('people','atab','tri');

create table stab(RDF$STC_GRAPH varchar2(4000), RDF$STC_sub varchar2(4000),
                  RDF$STC_pred varchar2(4000), RDF$STC_obj varchar2(4000));
grant select on stab to mdsys;
grant insert on atab to mdsys;

insert into stab values (null, '<http://empdb/TESTUSER.EMP/EMPNO=1>', '<http://
people.org/age>', '"35"^^<http://www.w3.org/2001/XMLSchema#int>');
insert into stab values (null, '<http://empdb/TESTUSER.EMP/EMPNO=2>', '<http://
people.org/age>', '"39"^^<http://www.w3.org/2001/XMLSchema#int>');
insert into stab values (null, '<http://empdb/TESTUSER.EMP/EMPNO=3>', '<http://
people.org/age>', '"30"^^<http://www.w3.org/2001/XMLSchema#int>');
insert into stab values (null, '<http://empdb/TESTUSER.EMP/EMPNO=4>', '<http://
people.org/age>', '"42"^^<http://www.w3.org/2001/XMLSchema#int>');
commit;

exec sem_apis.bulk_load_from_staging_table('people','testuser','stab');

-- Querying multiple datasets --
SELECT emp, age
  FROM TABLE(SEM_MATCH(
    'SELECT ?emp ?age WHERE{
       ?emp peop:age ?age
       SERVICE orardbm:empdb_model { ?emp emp:ref-DEPTNO ?dept . ?dept dept:LOC
"Boston" }
    }',
    SEM_Models('people'),
```

```
NULL,
SEM_ALIASES(
      SEM_ALIAS('dept','http://empdb/TESTUSER.DEPT#'),
      SEM_ALIAS('emp','http://empdb/TESTUSER.EMP#'),
      SEM_ALIAS('peop','http://people.org/')
    ),
NULL));
```

Part II

Reference and Supplementary Information

This document has the following parts:

- Part I provides conceptual and usage information about RDF Semantic Graph.
- Part II provides reference information about RDF Semantic Graph subprograms; it also provides supplementary information in appendixes and a glossary.

Part II contains the following chapters with reference and supplementary information:

- SEM_APIS Package Subprograms (page 11-1)
- SEM_OLS Package Subprograms (page 12-1)
- SEM_PERF Package Subprograms (page 13-1)
- SEM_RDFCTX Package Subprograms (page 14-1)
- SEM_RDFSA Package Subprograms (page 15-1)
- Enabling_ Downgrading_ or Removing RDF Semantic Graph Support (page A-1)
- SEM_MATCH Support for Spatial Queries (page B-1)
- Glossary (page 1)

To understand the examples in the reference chapters, you must understand the conceptual and data type information in RDF Semantic Graph Overview (page 1-1) and OWL Concepts (page 2-1).

11

SEM_APIS Package Subprograms

The SEM_APIS package contains subprograms (functions and procedures) for working with the Resource Description Framework (RDF) and Web Ontology Language (OWL) in an Oracle database. To use the subprograms in this chapter, you must understand the conceptual and usage information in RDF Semantic Graph Overview (page 1-1) and OWL Concepts (page 2-1).

This chapter provides reference information about the subprograms, listed in alphabetical order.

11.1 SEM_APIS.ADD_DATATYPE_INDEX

Format

```
SEM_APIS.ADD_DATATYPE_INDEX(
     datatype        IN VARCHAR2,
     tablespace_name IN VARCHAR2 default NULL,
     parallel        IN PLS_INTEGER default NULL,
     online          IN BOOLEAN default FALSE,
     options         IN VARCHAR2 default NULL);
```

Description

Adds a data type index for the specified data type to the semantic network.

Parameters

datatype
URI of the data type to index.

tablespace_name
Destination tablespace for the index.

parallel
Degree of parallelism to use when building the index.

online
TRUE allows DML operations affecting the index during creation of the index; FALSE (the default) does not allow DML operations affecting the index during creation of the index.

options
String specifying options for index creation using the form *OPTION_NAME=option_value*. Supported options associated with spatial index creation are SRID, TOLERANCE, and DIMENSIONS (see Indexing Spatial Data (page 1-68)). Supported options associated with text index creation are

PREFIX_INDEX, PREFIX_MIN_LENGTH, PREFIX_MAX_LENGTH, and SUBSTRING_INDEX (see Full-Text Search (page 1-65)). The option name keywords are case sensitive and must be specified in uppercase.

Usage Notes

You must have DBA privileges to call this procedure.

For more information about data type indexing, see Using Data Type Indexes (page 1-88).

For information about creating a data type index on RDF spatial data, see Indexing Spatial Data (page 1-68).

Examples

The following example creates an index on xsd:string typed literals and plain literals in the MY_TBS tablespace.

```
EXECUTE SEM_APIS.ADD_DATATYPE_INDEX('http://www.w3.org/2001/XMLSchema#string',
tablespace_name=>'MY_TBS', parallel=>4);
```

11.2 SEM_APIS.ADD_SEM_INDEX

Format

```
SEM_APIS.ADD_SEM_INDEX(
     index_code  IN VARCHAR2);
```

Description

Creates a semantic network index that results in creation of a nonunique B-tree index in UNUSABLE status for each of the existing models and entailments of the semantic network.

Parameters

index_code
Index code string.

Usage Notes

You must have DBA privileges to call this procedure.

For an explanation of semantic network indexes, see Using Semantic Network Indexes (page 1-86).

Examples

The following example creates a semantic network index with the index code string pcsm on the models and entailments of the semantic network.

```
EXECUTE SEM_APIS.ADD_SEM_INDEX('pscm');
```

11.3 SEM_APIS.ALTER_DATATYPE_INDEX

Format

```
SEM_APIS.ALTER_DATATYPE_INDEX(
     datatype        IN VARCHAR2,
     command         IN VARCHAR2,
     tablespace_name IN VARCHAR2 default NULL,
     parallel        IN PLS_INTEGER default NULL,
     online          IN BOOLEAN default FALSE);
```

Description

Alters a data type index.

Parameters

datatype
URI of the data type to index.

options
String specifying the command to be performed: `REBUILD` to rebuild the data type index, or `UNUSABLE` to marks the data type index as unusable. The value for this parameter is not case-sensitive.

tablespace_name
Destination tablespace for the index.

parallel
Degree of parallelism to use when rebuilding the index.

online
`TRUE` allows DML operations affecting the index during rebuilding of the index; `FALSE` (the default) does not allow DML operations affecting the index during rebuilding of the index.

Usage Notes

You must have DBA privileges to call this procedure.

For an explanation of data type indexes, see Using Data Type Indexes (page 1-88).

Examples

The following example rebuilds the index on `xsd:string` typed literals and plain literals in the MY_TBS tablespace.

```
EXECUTE SEM_APIS.ALTER_DATATYPE_INDEX('http://www.w3.org/2001/XMLSchema#string',
command=>'REBUILD', tablespace_name=>'MY_TBS', parallel=>4);
```

11.4 SEM_APIS.ALTER_ENTAILMENT

Format

```
SEM_APIS.ALTER_ENTAILMENT(
     entailment_name IN VARCHAR2,
```

```
    command         IN VARCHAR2,
    tablespace_name IN VARCHAR2,
    parallel        IN NUMBER(38) DEFAULT NULL);
```

Description

Alters an entailment (rules index). Currently, the only action supported is to move the entailment to a specified tablespace.

Parameters

entailment_name
Name of the entailment.

command
Must be the string `MOVE`.

tablespace_name
Name of the destination tablespace.

parallel
Degree of parallelism to be associated with the operation. For more information about parallel execution, see *Oracle Database VLDB and Partitioning Guide*.

Usage Notes

For an explanation of entailments, see Entailments (Rules Indexes) (page 1-11).

Examples

The following example moves the entailment named `rdfs_rix_family` to the tablespace named `my_tbs`.

```
EEXECUTE SEM_APIS.ALTER_ENTAILMENT('rdfs_rix_family', 'MOVE',  'my_tbs');
```

11.5 SEM_APIS.ALTER_MODEL

Format

```
SEM_APIS.ALTER_MODEL(
    model_name      IN VARCHAR2,
    command         IN VARCHAR2,
    tablespace_name IN VARCHAR2,
    parallel        IN NUMBER(38) DEFAULT NULL);
```

Description

Alters a model. Currently, the only action supported is to move the model to a specified tablespace.

Parameters

model_name
Name of the model.

command
Must be the string `MOVE`.

tablespace_name
Name of the destination tablespace.

parallel
Degree of parallelism to be associated with the operation. For more information about parallel execution, see *Oracle Database VLDB and Partitioning Guide*.

Usage Notes

For an explanation of models, see Semantic Data Modeling (page 1-3) and Semantic Data in the Database (page 1-3).

This procedure is not supported on version-enabled RDF models, as explained in Special Considerations When Using Workspace Manager Support for RDF Data (page 6-5).

Examples

The following example moves the model named `family` to the tablespace named `my_tbs`.

```
EEXECUTE SEM_APIS.ALTER_MODEL('family', 'MOVE',  'my_tbs');
```

11.6 SEM_APIS.ALTER_SEM_INDEX_ON_ENTAILMENT

Format

```
SEM_APIS.ALTER_SEM_INDEX_ON_ENTAILMENT(
     entailment_name  IN VARCHAR2,
     index_code       IN VARCHAR2,
     command          IN VARCHAR2,
     tablespace_name  IN VARCHAR2 DEFAULT NULL,
     use_compression  IN BOOLEAN DEFAULT NULL,
     parallel         IN NUMBER(38) DEFAULT NULL,
     online           IN BOOLEAN DEFAULT FALSE);
```

Description

Alters a semantic network index on an entailment.

Parameters

entailment_name
Name of the entailment.

index_code
Index code string.

command
String value containing one of the following commands: `REBUILD` rebuilds the semantic network index on the entailment, or `UNUSABLE` marks as unusable the semantic network index on the entailment. The value for this parameter is not case-sensitive.

tablespace_name
Name of the destination tablespace for the rebuild operation.

use_compression
Specifies whether compression should be used when rebuilding the index.

parallel
Degree of parallelism to be associated with the operation. For more information about parallel execution, see *Oracle Database VLDB and Partitioning Guide*.

online
TRUE allows DML operations affecting the index during the rebuilding of the index; FALSE (the default) does not allow DML operations affecting the index during the rebuilding of the index.

Usage Notes

For an explanation of semantic network indexes, see Using Semantic Network Indexes (page 1-86).

Examples

The following example rebuilds (and makes usable if it is unusable) the semantic network index on the entailment named `rdfs_rix_family`.

```
EXECUTE SEM_APIS.ALTER_SEM_INDEX_ON_ENTAILMENT('rdfs_rix_family', 'pscm', 'rebuild');
```

11.7 SEM_APIS.ALTER_SEM_INDEX_ON_MODEL

Format

```
SEM_APIS.ALTER_SEM_INDEX_ON_MODEL(
     model_name       IN VARCHAR2,
     index_code       IN VARCHAR2,
     command          IN VARCHAR2,
     tablespace_name  IN VARCHAR2 DEFAULT NULL,
     use_compression  IN BOOLEAN DEFAULT NULL,
     parallel         IN NUMBER(38) DEFAULT NULL,
     online           IN BOOLEAN DEFAULT FALSE);
```

Description

Alters a semantic network index on a model.

Parameters

model_name
Name of the model.

index_code
Index code string.

command
String value containing one of the following commands: REBUILD rebuilds the semantic network index on the model, or UNUSABLE marks as unusable the semantic network index on the model. The value for this parameter is not case-sensitive.

tablespace_name
Name of the destination tablespace for the rebuild operation.

use_compression
Specifies whether compression should be used when rebuilding the index.

parallel
Degree of parallelism to be associated with the operation. For more information about parallel execution, see *Oracle Database VLDB and Partitioning Guide*.

online
TRUE allows DML operations affecting the index during the rebuilding of the index; FALSE (the default) does not allow DML operations affecting the index during the rebuilding of the index.

Usage Notes

For an explanation of semantic network indexes, see Using Semantic Network Indexes (page 1-86).

Examples

The following example rebuilds (and makes usable if it is unusable) the semantic network index on the model named `family`.

```
EXECUTE SEM_APIS.ALTER_SEM_INDEX_ON_MODEL('family', 'pscm', 'rebuild');
```

11.8 SEM_APIS.ANALYZE_ENTAILMENT

Format

```
SEM_APIS.ANALYZE_ENTAILMENT(
     entailment_name  IN VARCHAR2,
     estimate_percent IN NUMBER DEFAULT to_estimate_percent_type
(get_param('ESTIMATE_PERCENT')),
     method_opt       IN VARCHAR2 DEFAULT get_param('METHOD_OPT'),
     degree           IN NUMBER DEFAULT to_degree_type(get_param('DEGREE')),
     cascade          IN BOOLEAN DEFAULT to_cascade_type(get_param('CASCADE')),
     no_invalidate    IN BOOLEAN DEFAULT to_no_invalidate_type
(get_param('NO_INVALIDATE')),
     force            IN BOOLEAN DEFAULT FALSE);
```

Description

Collects statistics for a specified entailment (rules index).

Parameters

entailment_name
Name of the entailment.

estimate_percent
Percentage of rows to estimate in the internal table partition containing information about the entailment (NULL means compute). The valid range is [0.000001,100]. Use the constant DBMS_STATS.AUTO_SAMPLE_SIZE to have Oracle determine the appropriate sample size for good statistics. This is the usual default.

method_opt
Accepts either of the following options, or both in combination, for the internal table partition containing information about the entailment:

- `FOR ALL [INDEXED | HIDDEN] COLUMNS [size_clause]`
- `FOR COLUMNS` [size clause] column | attribute [size_clause] [,column | attribute [size_clause]...]

`size_clause` is defined as `size_clause := SIZE {integer | REPEAT | AUTO | SKEWONLY}`

`column` is defined as `column := column_name | (extension)`

- `integer` : Number of histogram buckets. Must be in the range [1,254].
- `REPEAT` : Collects histograms only on the columns that already have histograms.
- `AUTO` : Oracle determines the columns to collect histograms based on data distribution and the workload of the columns.
- `SKEWONLY` : Oracle determines the columns to collect histograms based on the data distribution of the columns.
- `column_name` : name of a column
- `extension:` Can be either a column group in the format of (`column_name`, `column_name` [, ...]) or an expression.

The usual default is `FOR ALL COLUMNS SIZE AUTO`.

degree
Degree of parallelism for the internal table partition containing information about the entailment. The usual default for `degree` is `NULL`, which means use the table default value specified by the `DEGREE` clause in the `CREATE TABLE` or `ALTER TABLE` statement. Use the constant `DBMS_STATS.DEFAULT_DEGREE` to specify the default value based on the initialization parameters. The `AUTO_DEGREE` value determines the degree of parallelism automatically. This is either 1 (serial execution) or `DEFAULT_DEGREE` (the system default value based on number of CPUs and initialization parameters) according to size of the object.

cascade
Gathers statistics on the indexes for the internal table partition containing information about the entailment. Use the constant `DBMS_STATS.AUTO_CASCADE` to have Oracle determine whether index statistics are to be collected or not. This is the usual default.

no_invalidate
Does not invalidate the dependent cursors if set to `TRUE`. The procedure invalidates the dependent cursors immediately if set to `FALSE`. Use `DBMS_STATS.AUTO_INVALIDATE`. to have Oracle decide when to invalidate dependent cursors. This is the usual default.

force
`TRUE` gathers statistics even if the entailment is locked; `FALSE` (the default) does not gather statistics if the entailment is locked.

Usage Notes

Index statistics collection can be parellelized except for cluster, domain, and join indexes.

This procedure internally calls the DBMS_STATS.GATHER_TABLE_STATS procedure, which collects statistics for the internal table partition that contains information about the entailment. The DBMS_STATS.GATHER_TABLE_STATS procedure is documented in *Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

For information about entailments, see Entailments (Rules Indexes) (page 1-11).

Examples

The following example collects statistics for the entailment named `rdfs_rix_family`.

```
EXECUTE SEM_APIS.ANALYZE_ENTAILMENT('rdfs_rix_family');
```

11.9 SEM_APIS.ANALYZE_MODEL

Format

```
SEM_APIS.ANALYZE_MODEL(
     model_name       IN VARCHAR2,
     estimate_percent IN NUMBER DEFAULT to_estimate_percent_type
(get_param('ESTIMATE_PERCENT')),
     method_opt       IN VARCHAR2 DEFAULT get_param('METHOD_OPT'),
     degree           IN NUMBER DEFAULT to_degree_type(get_param('DEGREE')),
     cascade          IN BOOLEAN DEFAULT to_cascade_type(get_param('CASCADE')),
     no_invalidate    IN BOOLEAN DEFAULT to_no_invalidate_type
(get_param('NO_INVALIDATE')),
     force            IN BOOLEAN DEFAULT FALSE);
```

Description

Collects optimizer statistics for a specified model.

Parameters

model_name
Name of the model.

estimate_percent
Percentage of rows to estimate in the internal table partition containing information about the model (`NULL` means compute). The valid range is [0.000001,100]. Use the constant `DBMS_STATS.AUTO_SAMPLE_SIZE` to have Oracle determine the appropriate sample size for good statistics. This is the usual default.

method_opt
Accepts either of the following options, or both in combination, for the internal table partition containing information about the model:

- `FOR ALL [INDEXED | HIDDEN] COLUMNS [size_clause]`
- `FOR COLUMNS` [size clause] column | attribute [size_clause] [,column | attribute [size_clause]...]

`size_clause` is defined as `size_clause := SIZE {integer | REPEAT | AUTO | SKEWONLY}`

`column` is defined as `column := column_name | (extension)`

- `integer` : Number of histogram buckets. Must be in the range [1,254].
- `REPEAT` : Collects histograms only on the columns that already have histograms.
- `AUTO` : Oracle determines the columns to collect histograms based on data distribution and the workload of the columns.
- `SKEWONLY` : Oracle determines the columns to collect histograms based on the data distribution of the columns.
- `column_name` : name of a column
- `extension:` Can be either a column group in the format of (`column_name`, `column_name` [, ...]) or an expression.

The usual default is `FOR ALL COLUMNS SIZE AUTO`.

degree
Degree of parallelism for the internal table partition containing information about the model. The usual default for `degree` is `NULL`, which means use the table default value specified by the `DEGREE` clause in the `CREATE TABLE` or `ALTER TABLE` statement. Use the constant `DBMS_STATS.DEFAULT_DEGREE` to specify the default value based on the initialization parameters. The `AUTO_DEGREE` value determines the degree of parallelism automatically. This is either 1 (serial execution) or `DEFAULT_DEGREE` (the system default value based on number of CPUs and initialization parameters) according to size of the object.

cascade
Gathers statistics on the indexes for the internal table partition containing information about the model. Use the constant `DBMS_STATS.AUTO_CASCADE` to have Oracle determine whether index statistics are to be collected or not. This is the usual default.

no_invalidate
Does not invalidate the dependent cursors if set to `TRUE`. The procedure invalidates the dependent cursors immediately if set to `FALSE`. Use `DBMS_STATS.AUTO_INVALIDATE`. to have Oracle decide when to invalidate dependent cursors. This is the usual default.

force
`TRUE` gathers statistics even if the model is locked; `FALSE` (the default) does not gather statistics if the model is locked.

Usage Notes

Index statistics collection can be parellelized except for cluster, domain, and join indexes.

This procedure internally calls the DBMS_STATS.GATHER_TABLE_STATS procedure, which collects optimizer statistics for the internal table partition that contains information about the model. The DBMS_STATS.GATHER_TABLE_STATS procedure is documented in *Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example collects statistics for the semantic model named `family`.

```
EXECUTE SEM_APIS.ANALYZE_MODEL('family');
```

11.10 SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE

Format

```
SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE(
     model_name    IN VARCHAR2,
     table_owner   IN VARCHAR2,
     table_name    IN VARCHAR2,
     flags         IN VARCHAR2 DEFAULT NULL,
     debug         IN INTEGER DEFAULT NULL,
     start_comment IN VARCHAR2 DEFAULT NULL,
     end_comment   IN VARCHAR2 DEFAULT NULL);
```

Description

Loads semantic data from a staging table.

Parameters

model_name
Name of the model.

table_owner
Name of the schema that owns the staging table that holds semantic data to be loaded.

table_name
Name of the staging table that holds semantic data to be loaded.

flags
An optional quoted string with one or more of the following keyword specifications:

- `DEL_BATCH_DUPS=USE_INSERT` allows the use of an insertion-based strategy for duplicate elimination that may lead to faster processing if the input data contains many duplicates.
- `MBV_METHOD=SHADOW` allows the use of a different value loading strategy that may lead to faster processing for large loads.
- `PARALLEL_CREATE_INDEX` allows internal indexes to be created in parallel, which may improve the performance of the bulk load processing.
- `PARALLEL=<integer>` allows much of the processing used during bulk load to be done in parallel using the specified degree of parallelism to be associated with the operation.
- `PARSE` allows parsing of triples retrieved from the staging table (also parses triples containing graph names).
- `<task>_JOIN_HINT=<join_type>`, where *<task>* can be any of the following internal tasks performed during bulk load: `IZC` (is zero collisions), `MBV` (merge batch values), or `MBT` (merge batch triples, used when adding triples to a non-empty model), and where *<join_type>* can be `USE_NL` and `USE_HASH`.

debug
(Reserved for future use)

start_comment
Optional comment about the start of the load operation.

end_comment
Optional comment about the end of the load operation.

Usage Notes

You must first load semantic data into a staging table before calling this procedure. See Bulk Loading Semantic Data Using a Staging Table (page 1-74) for more information.

Using BULK_LOAD_FROM_STAGING_TABLE with Fine Grained Access Control (VPD and OLS)

When fine-grained access control (explained in Fine-Grained Access Control for RDF Data (page 5-1)) is enabled for a specific model using VPD or for the entire network using OLS, only a user with FULL access privileges to the associated policy may perform the bulk load operation. Full access privileges to a VPD policy are granted by setting the context attribute defined by the package constant `sem_rdfsa.VPD_FULL_ACCESS` to 1 for the namespace associated with the VPD policy (see VPD Policy for RDF Data (page C-3)). When OLS is enabled, full access privileges to the OLS policy are granted using the SA_USER_ADMIN.SET_USER_PRIVS procedure.

The data loaded into a model with VPD enabled is immediately available for access and does not need any maintenance operations. However, when the OLS is used, the label column in the tables storing the RDF triples must be maintained. By default, with OLS enabled, the label column in the tables storing the RDF triples is set to null. If you have FULL access, you can reset the labels for the newly inserted triples as well as any resources introduced by the new batch of triples by using appropriate subprograms (SEM_RDFSA.SET_RESOURCE_LABEL (page 15-16) and SEM_RDFSA.SET_PREDICATE_LABEL (page 15-14)).

Optionally, you can define a numeric column named RDF$STC_CTXT1 in the staging table and the application table, to assign the sensitivity label of the triple before the data is loaded into the desired model. Such labels are automatically applied to the corresponding triples stored in the MDSYS.RDF_LINK$ table. The labels for the newly introduced resources may still have to be applied separately before or after the load, and the system does not validate the labels assigned during bulk load operation.

The RDF$STC_CTXT1 column in the application table has no significance, and it may be dropped after the bulk load operation.

Examples

The following example loads semantic data stored in the staging table named STAGE_TABLE in schema SCOTT into the semantic model named `family`. The example includes some join hints.

```
EXECUTE SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE('family', 'scott', 'stage_table',
flags => 'IZC_JOIN_HINT=USE_HASH MBV_JOIN_HINT=USE_HASH');
```

11.11 SEM_APIS.CLEANUP_FAILED

Format

```
SEM_APIS.CLEANUP_FAILED(
     rdf_object_type  IN VARCHAR2,
     rdf_object_name  IN VARCHAR2);
```

Description

Drops (deletes) a specified rulebase or entailment if it is in a failed state.

Parameters

rdf_object_type
Type of the RDF object: `RULEBASE` for a rulebase or `RULES_INDEX` for an entailment (rules index).

rdf_object_name
Name of the RDF object of type `rdf_object_type`.

Usage Notes

This procedure checks to see if the specified RDF object is in a failed state; and if the object is in a failed state, the procedure deletes the object.

A rulebase or entailment is in a failed state if a system failure occurred during the creation of that object. You can check if a rulebase or entailment is in a failed state by checking to see if the value of the STATUS column is `FAILED` in the SDO_RULEBASE_INFO view (described in Inferencing: Rules and Rulebases (page 1-8)) or the SDO_RULES_INDEX_INFO view (described in Entailments (Rules Indexes) (page 1-11)), respectively.

If the rulebase or entailment is not in a failed state, this procedure performs no action and returns a successful status.

An exception is generated if the RDF object is currently being used.

Examples

The following example deletes the rulebase named `family_rb` if (and only if) that rulebase is in a failed state.

```
EXECUTE SEM_APIS.CLEANUP_FAILED('RULEBASE', 'family_rb');
```

11.12 SEM_APIS.COMPOSE_RDF_TERM

Format

```
SEM_APIS.COMPOSE_RDF_TERM(
     value_name    IN VARCHAR2,
     value_type    IN VARCHAR2,
     literal_type  IN VARCHAR2,
     language_type IN VARCHAR2
     ) RETURN VARCHAR2;
```

or

```
SEM_APIS.COMPOSE_RDF_TERM(
     value_name    IN VARCHAR2,
     value_type    IN VARCHAR2,
     literal_type  IN VARCHAR2,
     language_type IN VARCHAR2,
     long_value    IN CLOB,
     options       IN VARCHAR2 DEFAULT NULL,
     ) RETURN CLOB;
```

Description

Creates and returns an RDF term using the specified parameters.

Parameters

value_name
Value name. Must match a value in the VALUE_NAME column in the MDSYS.RDF_VALUE$ table (described in Statements (page 1-5)) or in the *var* attribute returned from SEM_MATCH table function.

value_type
The type of text information. Must match a value in the VALUE_TYPE column in the MDSYS.RDF_VALUE$ table (described in Statements (page 1-5)) or in the *var* $RDFVTYP attribute returned from SEM_MATCH table function.

literal_type
For typed literals, the type information; otherwise, null. Must either be a null value or match a value in the LITERAL_TYPE column in the MDSYS.RDF_VALUE$ table (described in Statements (page 1-5)) or in the *var*$RDFLTYP attribute returned from SEM_MATCH table function.

language_type
Language tag. Must match a value in the LANGUAGE_TYPE column in the MDSYS.RDF_VALUE$ table (described in Statements (page 1-5)) or in the *var* $RDFLANG attribute returned from SEM_MATCH table function.

long_value
The character string if the length of the lexical value is greater than 4000 bytes. Must match a value in the LONG_VALUE column in the MDSYS.RDF_VALUE$ table (described in Statements (page 1-5)) or in the *var*$RDFCLOB attribute returned from SEM_MATCH table function.

options
(Reserved for future use.)

Usage Notes

If you specify an inconsistent combination of values for the parameters, this function returns a null value. If a null value is returned but you believe that the values for the parameters are appropriate (reflecting columns from the same row in the MDSYS.RDF_VALUE$ table or from a SEM_MATCH query for the same variable), contact Oracle Support.

Examples

The following example returns, for each member of the family whose height is known, the RDF term for the height and also just the value portion of the height.

```
SELECT x, SEM_APIS.COMPOSE_RDF_TERM(h, h$RDFVTYP, h$RDFLTYP, h$RDFLANG)
       h_rdf_term, h
  FROM TABLE(SEM_MATCH(
    '{?x :height ?h}',
    SEM_Models('family'),
    null,
    SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
    null))
ORDER BY x;
X
--------------------------------------------------------------------------------
H_RDF_TERM
--------------------------------------------------------------------------------
H
--------------------------------------------------------------------------------
http://www.example.org/family/Cathy
"5.8"^^<http://www.w3.org/2001/XMLSchema#decimal>
5.8

http://www.example.org/family/Cindy
"6"^^<http://www.w3.org/2001/XMLSchema#decimal>
6

http://www.example.org/family/Jack
"6"^^<http://www.w3.org/2001/XMLSchema#decimal>
6

http://www.example.org/family/Tom
"5.75"^^<http://www.w3.org/2001/XMLSchema#decimal>
5.75

4 rows selected.
```

The following example returns the RDF terms for a few of the values stored in the MDSYS.RDF_VALUE$ table.

```
SELECT SEM_APIS.COMPOSE_RDF_TERM(value_name, value_type, literal_type,
     language_type)
  FROM MDSYS.RDF_VALUE$ WHERE ROWNUM < 5;

SEM_APIS.COMPOSE_RDF_TERM(VALUE_NAME,VALUE_TYPE,LITERAL_TYPE,LANGUAGE_TYPE)
--------------------------------------------------------------------------------
<http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#Property>
```

11.13 SEM_APIS.CREATE_ENTAILMENT

Format

```
SEM_APIS.CREATE_ENTAILMENT(
     entailment_name_in IN VARCHAR2,
     models_in          IN SEM_MODELS,
     rulebases_in       IN SEM_RULEBASES,
     passes             IN NUMBER DEFAULT SEM_APIS.REACH_CLOSURE,
     inf_components_in  IN VARCHAR2 DEFAULT NULL,
     options            IN VARCHAR2 DEFAULT NULL,
     delta_in           IN SEM_MODELS DEFAULT NULL,
     label_gen          IN RDFSA_LABELGEN DEFAULT NULL,
```

```
include_named_g    IN SEM_GRAPHS DEFAULT NULL,
include_default_g  IN SEM_MODELS DEFAULT NULL,
include_all_g      IN SEM_MODELS DEFAULT NULL,
inf_ng_name        IN VARCHAR2 DEFAULT NULL,
inf_ext_user_func_name IN VARCHAR2 DEFAULT NULL);
```

Description

Creates an entailment (rules index) that can be used to perform OWL or RDFS inferencing, and optionally use user-defined rules.

Parameters

entailment_name_in
Name of the entailment to be created.

models_in
One or more model names. Its data type is SEM_MODELS, which has the following definition: `TABLE OF VARCHAR2(25)`

rulebases_in
One or more rulebase names. Its data type is SEM_RULEBASES, which has the following definition: `TABLE OF VARCHAR2(25)`. Rules and rulebases are explained in Inferencing: Rules and Rulebases (page 1-8).

passes
The number of rounds that the inference engine should run. The default value is `SEM_APIS.REACH_CLOSURE`, which means the inference engine will run till a closure is reached. If the number of rounds specified is less than the number of actual rounds needed to reach a closure, the status of the entailment will then be set to `INCOMPLETE`.

inf_components_in
A comma-delimited string of keywords representing inference components, for performing selective or component-based inferencing. If this parameter is null, the default set of inference components is used. See the Usage Notes for more information about inference components.

options
A comma-delimited string of options to control the inference process by overriding the default inference behavior. To enable an option, specify *`option-name`*`=T`; to disable an option, you can specify *`option-name`*`=F` (the default). The available option-name values are `COL_COMPRESS`, `DEST_MODEL`, `DISTANCE`, `DOP`, `ENTAIL_ANYWAY`, `HASH_PART`, `INC`, `LOCAL_NG_INF`, `OPT_SAMEAS`, `RAW8`, `PROOF`, and `USER_RULES`. See the Usage Notes for explanations of each value.

delta_in
If incremental inference is in effect, specifies one or more models on which to perform incremental inference. Its data type is SEM_MODELS, which has the following definition: `TABLE OF VARCHAR2(25)`

The triples in the first model in `delta_in` are copied to the first model in `models_in`, and the entailment (rules index) in `rules_index_in` is updated; then the triples in the second model (if any) in `delta_in` are copied to the second model (if any) in `models_in`, and the entailment in `rules_index_in` is updated; and so

on until all triples are copied and the entailment is updated. (The `delta_in` parameter has no effect if incremental inference is not enabled for the entailment.)

label_gen
An instance of MDSYS.RDFSA_LABELGEN or a subtype of it, defining the logic for generating Oracle Label Security (OLS) labels for inferred triples. What you specify for this parameter depends on whether you use the default label generator or a custom label generator:

- If you use the default label generator, specify one of the following constants: `SEM_RDFSA.LABELGEN_RULE` for Use Rule Label, `SEM_RDFSA.LABELGEN_SUBJECT` for Use Subject Label, `SEM_RDFSA.LABELGEN_PREDICATE` for Use Predicate Label, `SEM_RDFSA.LABELGEN_OBJECT` for Use Object Label, `SEM_RDFSA.LABELGEN_DOMINATING` for Use Dominating Label, `SEM_RDFSA.LABELGEN_ANTECED` for Use Antecedent Labels. For a detailed explanation of each constant, see Generating Labels for Inferred Triples (page 5-13).
- If you use a custom label generator, specify the custom label generator type. For information about creating and implementing a custom label generator, see Using Labels Based on Application Logic (page 5-16).

include_named_g
Causes all triples from the specified named graphs (across all source models) to participate in named graph based global inference (NGGI, explained in Named Graph Based Global Inference (NGGI) (page 2-15)). For example, `include_named_g => sem_graphs('<urn:G1>','<urn:G2>')` implies that triples from named graphs `G1` and `G2` will be included in NGGI.

Its data type is SEM_GRAPHS, which has the following definition: `TABLE OF VARCHAR2(4000)`.

include_default_g
Causes all triples with a null graph name in the specified models to participate in named graph based global inference (NGGI, explained in Named Graph Based Global Inference (NGGI) (page 2-15)). For example, `include_default_g => sem_models('m1')` causes all triples with a null graph name from model `M1` to be included in NGGI.

include_all_g
Causes all triples, regardless of their graph name values, in the specified models to participate in named graph based global inference (NGGI, explained in Named Graph Based Global Inference (NGGI) (page 2-15)). For example, `include_all_g => sem_models('m2')`causes all triples in model `M2` to be included in NGGI.

inf_ng_name
Assigns the specified graph name to all the new triples inferred by the named graph based global inference (NGGI, explained in Named Graph Based Global Inference (NGGI) (page 2-15)).

inf_ext_user_func_name
The name of a user-defined inference function, or a comma-delimited list of names of user-defined functions. For information about creating user-defined inference functions, including format requirements and options for certain parameters, see API Support for User-Defined Inferencing (page 9-2). (For information about user-defined

inferencing, including examples, see User-Defined Inferencing and Querying (page 9-1).)

Usage Notes

For the `inf_components_in` parameter, you can specify any combination of the following keywords: `SCOH`, `COMPH`, `DISJH`, `SYMMH`, `INVH`, `SPIH`, `MBRH`, `SPOH`, `DOMH`, `RANH`, `EQCH`, `EQPH`, `FPH`, `IFPH`, `DOM`, `RAN`, `SCO`, `DISJ`, `COMP`, `INV`, `SPO`, `FP`, `IFP`, `SYMM`, `TRANS`, `DIF`, `SAM`, `CHAIN`, `HASKEY`, `ONEOF`, `INTERSECT`, `INTERSECTSCOH`, `MBRLST`, `PROPDISJH`, `SKOSAXIOMS`, `SNOMED`, `SVFH`, `THINGH`, `THINGSAM`, `UNION`, `RDFP1`, `RDFP2`, `RDFP3`, `RDFP4`, `RDFP6`, `RDFP7`, `RDFP8AX`, `RDFP8BX`, `RDFP9`, `RDFP10`, `RDFP11`, `RDFP12A`, `RDFP12B`, `RDFP12C`, `RDFP13A`, `RDFP13B`, `RDFP13C`, `RDFP14A`, `RDFP14BX`, `RDFP15`, `RDFP16`, `RDFS2`, `RDFS3`, `RDFS4a`, `RDFS4b`, `RDFS5`, `RDFS6`, `RDFS7`, `RDFS8`, `RDFS9`, `RDFS10`, `RDFS11`, `RDFS12`, `RDFS13`. For an explanation of the meaning of these keywords, see Table 11-1 (page 11-18), where the keywords are listed in alphabetical order.

The default set of inference components for the OWLPrime vocabulary includes the following: `SCOH`, `COMPH`, `DISJH`, `SYMMH`, `INVH`, `SPIH`, `SPOH`, `DOMH`, `RANH`, `EQCH`, `EQPH`, `FPH`, `IFPH`, `SAMH`, `DOM`, `RAN`, `SCO`, `DISJ`, `COMP`, `INV`, `SPO`, `FP`, `IFP`, `SYMM`, `TRANS`, `DIF`, `RDFP14A`, `RDFP14BX`, `RDFP15`, `RDFP16`. However, note the following:

- Component `SAM` is not in this default OWLPrime list, because it tends to generate many new triples for some ontologies.
- Effective with Release 11.2, the native OWL inference engine supports the following new inference components: `CHAIN`, `HASKEY`, `INTERSECT`, `INTERSECTSCOH`, `MBRLST`, `ONEOF`, `PROPDISJH`, `SKOSAXIOMS`, `SNOMED`, `SVFH`, `THINGH`, `THINGSAM`, `UNION`. However, for backward compatibility, the OWLPrime rulebase and any existing rulebases do not include these new components by default; instead, to use these new inference components, you must specify them explicitly, and they are included in Table 11-1 (page 11-18). The following example creates an OWLPrime entailment for two OWL ontologies named `LUBM` and `UNIV`. Because of the additional inference components specified, this entailment will include the new semantics introduced in those inference components.

```
EXECUTE sem_apis.create_entailment('lubm1000_idx',sem_models('lubm','univ'),
    sem_rulebases('owlprime'), SEM_APIS.REACH_CLOSURE,
    'INTERSECT,INTERSECTSCOH,SVFH,THINGH,THINGSAM,UNION');
```

Table 11-1 Inferencing Keywords for inf_components_in Parameter

Keyword	Explanation
CHAIN	Captures the property chain semantics defined in OWL 2. Only chains of length 2 are supported. By default, this is included in the `SKOSCORE` rulebase. Subproperty chaining is an OWL 2 feature, and for backward compatibility this component is not by default included in the `OWLPrime` rulebase. (For information about property chain handling, see Property Chain Handling (page 3-4).) (New as of Release 11.2.)
COMPH	Performs inference based on owl:complementOf assertions and the interaction of owl:complementOf with other language constructs.
DIF	Generates owl:differentFrom assertions based on the symmetricity of owl:differentFrom.

Table 11-1 (Cont.) Inferencing Keywords for inf_components_in Parameter

Keyword	Explanation
DISJ	Infers owl:differentFrom relationships at instance level using owl:disjointWith assertions.
DISJH	Performs inference based on owl:disjointWith assertions and their interactions with other language constructs.
DOM	Performs inference based on RDFS2.
DOMH	Performs inference based on rdfs:domain assertions and their interactions with other language constructs.
EQCH	Performs inference that are relevant to owl:equivalentClass.
EQPH	Performs inference that are relevant to owl:equivalentProperty.
FP	Performs instance-level inference using instances of owl:FunctionalProperty.
FPH	Performs inference using instances of owl:FunctionalProperty.
HASKEY	Covers the semantics behind "keys" defined in OWL 2. In OWL 2, a collection of properties can be treated as a key to a class expression. For efficiency, the size of the collection must not exceed 3. (New as of Release 11.2.)
IFP	Performs instance-level inference using instances of owl:InverseFunctionalProperty.
IFPH	Performs inference using instances of owl:InverseFunctionalProperty.
INTERSECT	Handles the core semantics of owl:intersectionOf. For example, if class C is the intersection of classes C1, C2 and C3, then C is a subclass of C1, C2, and C3. In addition, common instances of all C1, C2, and C3 are also instances of C. (New as of Release 11.2.)
INTERSECTSCOH	Handles the fact that an intersection is the maximal common subset. For example, if class C is the intersection of classes C1, C2, and C3, then any common subclass of all C1, C2, and C3 is a subclass of C. (New as of Release 11.2.)
INV	Performs instance-level inference using owl:inverseOf assertions.
INVH	Performs inference based on owl:inverseOf assertions and their interactions with other language constructs.
MBRLST	Captures the semantics that for any resource, every item in the list given as the value of the `skos:memberList` property is also a value of the `skos:member` property. (See S36 in the SKOS detailed specification.) By default, this is included in the `SKOSCORE` rulebase. (New as of Release 11.2.)

Table 11-1 (Cont.) Inferencing Keywords for inf_components_in Parameter

Keyword	Explanation
ONEOF	Generates classification assertions based on the definition of the enumeration classes. In OWL, class extensions can be enumerated explicitly with the `owl:oneOf` constructor. (New as of Release 11.2.)
PROPDISJH	Captures the interaction between `owl:propertyDisjointWith` and `rdfs:subPropertyOf`. By default, this is included in SKOSCORE rulebase. propertyDisjointWith is an OWL 2 feature, and for backward compatibility this component is not by default included in the `OWLPrime` rulebase. (New as of Release 11.2.)
RANH	Performs inference based on `rdfs:range` assertions and their interactions with other language constructs.
RDFP*	(The rules corresponding to components with a prefix of *RDFP* can be found in *Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary*, by H.J. Horst.)
RDFS2, ... RDFS13	RDFS2, RDFS3, RDFS4a, RDFS4b, RDFS5, RDFS6, RDFS7, RDFS8, RDFS9, RDFS10, RDFS11, RDFS12, and RDFS13 are described in Section 7.3 of *RDF Semantics* (`http://www.w3.org/TR/rdf-mt/`). Note that many of the RDFS components are not relevant for OWL inference.
SAM	Performs inference about individuals based on existing assertions for those individuals and owl:sameAs.
SAMH	Infers owl:sameAs assertions using transitivity and symmetricity of owl:sameAs.
SCO	Performs inference based on RDFS9.
SCOH	Generates the subClassOf hierarchy based on existing rdfs:subClassOf assertions. Basically, C1 rdfs:subClassOf C2 and C2 rdfs:subClassOf C3 will infer C1 rdfs:subClassOf C3 based on transitivity. SCOH is also an alias of RDFS11.
SKOSAXIOMS	Captures most of the axioms defined in the SKOS detailed specification. By default, this is included in the `SKOSCORE` rulebase. (New as of Release 11.2.)
SNOMED	Performs inference based on the semantics of the OWL 2 EL profile, which captures the expressiveness of SNOMED CT (Systematized Nomenclature of Medicine - Clinical Terms), which is one of the most expressive and complex medical terminologies. (New as of Release 11.2.)
SPIH	Performs inference based on interactions between rdfs:subPropertyOf and owl:inverseOf assertions.
SPO	Performs inference based on RDFS7.
SPOH	Generates rdfs:subPropertyOf hierarchy based on transitivity of rdfs:subPropertyOf. It is an alias of RDFS5.

Table 11-1 (Cont.) Inferencing Keywords for inf_components_in Parameter

Keyword	Explanation
SVFH	Handles the following semantics that involves the interaction between owl:someValuesFrom and rdfs:subClassOf. Consider two existential restriction classes C1 and C2 that both use the same restriction property. Assume further that the owl:someValuesFrom constraint class for C1 is a subclass of that for C2. Then C1 can be inferred as a subclass of C2. (New as of Release 11.2.)
SYMM	Performs instance-level inference using instances of owl:SymmetricProperty.
SYMH	Performs inference for properties of type owl:SymmetricProperty.
THINGH	Handles the semantics that any defined OWL class is a subclass of owl:Thing. The consequence of this rule is that instances of all defined OWL classes will become instances of owl:Thing. The size of the inferred graph will very likely be bigger with this component selected. (New as of Release 11.2.)
THINGSAM	Handles the semantics that instances of owl:Thing are equal to (owl:sameAs) themselves. This component is provided for the convenience of some applications. Note that an application does not have to select this inference component to figure out an individual is equal to itself; this kind of information can easily be built in the application logic. (New as of Release 11.2.)
TRANS	Calculates transitive closure for instances of owl:TransitiveProperty.
UNION	Captures the core semantics of the `owl:unionOf` construct. Basically, the `union` class is a superclass of all member classes. For backward compatibility, this component is not by default included in the `OWLPrime` rulebase. (New as of Release 11.2.)

To deselect a component, use the component name followed by a minus (-) sign. For example, `SCOH-` deselects inference of the `subClassOf` hierarchy.

For the `options` parameter, you can enable the following options to override the default inferencing behavior:

- `COL_COMPRESS=T` creates temporary, intermediate working tables. This option can reduce the space required for such tables, and can improve the performance of the CREATE_ENTAILMENT operation with large data sets.

  By default `COL_COMPRESS=T` uses the "compress for query level low" setting; however, you can add `CPQH=T` to change to the "compress for query level high" setting.

  Note:

  You can specify `COL_COMPRESS=T` only on systems that support Hybrid Columnar Compression (HCC). For information about HCC, see *Oracle Database Concepts*.

- `DEST_MODEL=<model_name>` specifies, for incremental inference, the destination model to which the `delta_in` model or models are to be added. The specified destination model must be one of the models specified in the `models_in` parameter.
- `DISTANCE=T` generates ancillary distance information that is useful for semantic operators.
- `DOP=n` specifies the degree of parallelism for parallel inference, which can improve inference performance. For information about parallel inference, see Using Parallel Inference (page 2-14).
- `ENTAIL_ANYWAY=T` forces OWL inferencing to proceed and reuse existing inferred data (entailment) when the entailment has a valid status. By default, SEM_APIS.CREATE_ENTAILMENT quits immediately if there is already a valid entailment for the combination of models and rulebases.
- `HASH_PART=n` creates the specified number of hash partitions for internal working tables. (The number must be a power of 2: 2, 4, 8, 16, 32, and so on.) You may want to specify a value if there are many distinct predicates in the semantic data model. In Oracle internal testing on benchmark ontologies, HASH_PART=32 worked well.
- `INC=T` enables incremental inference for the entailment. For information about incremental inference, see Performing Incremental Inference (page 2-13).
- `LOCAL_NG_INF=T` causes named graph based *local* inference (NGLI) to be used instead of named graph based global inference (NGGI). For information about NGLI, see Named Graph Based Local Inference (NGLI) (page 2-16).
- `OPT_SAMEAS=T` uses consolidated `owl:sameAs` entailment for the entailment. If you specify this option, you cannot specify `PROOF=T`. For information about optimizing `owl:sameAs` inference, see Optimizing owl:sameAs Inference (page 2-11).
- `RAW8=T` uses RAW8 data types for the auxiliary inference tables. This option can improve entailment performance by up to 30% in some cases.
- `PROOF=T` generates proof for inferred triples. Do not specify this option unless you need to; it slows inference performance because it causes more data to be generated. If you specify this option, you cannot specify `OPT_SAMEAS=T`.
- `USER_RULES=T` causes any user-defined rules to be applied. If you specify this option, you cannot specify `PROOF=T` or `DISTANCE=T`, and you must accept the default value for the `passes` parameter.

For the `delta_in` parameter, inference performance is best if the value is small compared to the overall size of those models. In a typical scenario, the best results might be achieved when the delta contains fewer than 10,000 triples; however, some tests have shown significant inference performance improvements with deltas as large as 100,000 triples.

For the `label_gen` parameter, if you want to use the default OLS label generator, specify the appropriate SEM_RDFSA package constant value from Table 11-2 (page 11-23).

Table 11-2 SEM_RDFSA Package Constants for label_gen Parameter

Constant	Description
SEM_RDFSA.LABELGEN_SUBJECT	Label generator that applies the label associated with the inferred triple's subject as the triple's label.
SEM_RDFSA.LABELGEN_PREDICATE	Label generator that applies the label associated with the inferred triple's subject as the triple's label.
SEM_RDFSA.LABELGEN_OBJECT	Label generator that applies the label associated with the inferred triple's subject as the triple's label.
SEM_RDFSA.LABELGEN_RULE	Label generator that applies the label associated with the rule that directly produced the inferred triple as the triple's label. If you specify this option, you must also specify `PROOF=T` in the `options` parameter.
SEM_RDFSA.LABELGEN_DOMINATING	Label generator that computes a dominating label of all the available labels for the triple's components (subject, predicate, object, and rule), and applies it as the label for the inferred triple.

Fine-Grained Access Control (OLS and VPD) Considerations

When fine-grained access control is enabled for a specific model using VPD or for the entire network using OLS, only a user with FULL access privileges to the associated policy may create an entailment. Full access privileges to a VPD policy are granted by setting the context attribute defined by the package constant `sem_rdfsa.VPD_FULL_ACCESS` to 1 for the namespace associated with the VPD policy (see VPD Policy for RDF Data (page C-3)). When OLS is enabled, full access privileges to the OLS policy are granted using the `SA_USER_ADMIN.SET_USER_PRIVS` procedure. VPD policy is automatically enforced for any entailment created using at least one VPD enabled RDF model. As discussed in VPD Policy for RDF Data (page C-3), the RDF metadata associated with the VPD policy may need to be maintained using the newly inferred data to ensure that the predefined VPD policies are not circumvented using inferred data.

Inferred triples accessed through generated labels might not be same as conceptual triples inferred directly from the user accessible triples and rules. The labels generated using a subset of triple components may be weaker than intended. For example, one of the antecedents for the inferred triple may have a higher label than any of the components of the triple. When the label is generated based on just the triple components, end users with no access to one of the antecedents may still have access to the inferred triple. Even when the antecedents are used for custom label generation, the generated label may be stronger than intended. The inference process is not exhaustive, and information pertaining to any alternate ways of inferring the same triple is not available. So, the label generated using a given set of antecedents may be too strong, because the user with access to all the triples in the alternate path could infer the triple with lower access.

Even when generating a label that dominates all its components and antecedents, the label may not be precise. This is the case when labels considered for dominating relationship have non-overlapping group information. For example, consider two labels `L:C:NY` and `L:C:NH` where L is a level, C is a component and NY and NH are two groups. A simple label that dominates these two labels is `L:C:NY,NH`, and a true supremum for the two labels is `L:C:US`, where US is parent group for both NY and

NH. Unfortunately, neither of these two dominating labels is precise for the triple inferred from the triples with first two labels. If `L:C:NY,NH` is used for the inferred triple, a user with membership in either of these groups has access to the inferred triple, whereas the same user does not have access to one of its antecedents. On the other hand, if `L:C:US` is used for the inferred triple, a user with membership in both the groups and not in the US group will not be able to access the inferred triple, whereas that user could infer the triple by directly accessing its components and antecedents.

Because of these unique challenges with inferred triples, extra caution must be taken when choosing or implementing the label generator.

See also the OLS example in the Examples section.

Examples

The following example creates an entailment named `OWLTST_IDX` using the OWLPrime rulebase, and it causes proof to be generated for inferred triples.

```
EXECUTE sem_apis.create_entailment('owltst_idx', sem_models('owltst'),
sem_rulebases('OWLPRIME'), SEM_APIS.REACH_CLOSURE, null, 'PROOF=T');
```

The following example assumes an OLS environment. It creates a rulebase with a rule, and it creates an entailment.

```
-- Create an entailment with a rule. --
exec sdo_rdf_inference.create_entailment('contracts_rb');

insert into mdsys.rdfr_contracts_rb values (
  'projectLedBy', '(?x :drivenBy ?y) (?y :hasVP ?z)',  NULL,
  '(?x :isLedBy ?z)',
  SDO_RDF_Aliases(SDO_RDF_Alias('','http://www.myorg.com/pred/')));

-- Assign sensitivity label for the predicate to be inferred. --
-- Yhe predicate label may be set globally or it can be assign to --
-- the one or the models used to infer the data – e.g: CONTRACTS.
begin
  sem_rdfsa.set_predicate_label(
         model_name   => 'rdf$global',
         predicate    => 'http://www.myorg.com/pred/isLedBy',
         label_string => 'TS:US_SPCL');
end;
/

-- Create index with a specific label generator. --
begin
  sem_apis.create_entailment(
         entailment_name_in => 'contracts_inf',
         models_in          => SDO_RDF_Models('contracts'),
         rulebases_in       => SDO_RDF_Rulebases('contracts_rb'),
         options            => 'USER_RULES=T',
         label_gen          => sem_rdfsa.LABELGEN_PREDICATE);
end;
/

-- Check for any label exceptions and update them accordingly. --
update mdsys.rdfi_contracts_inf set ctxt1 = 1100 where ctxt1 = -1;

-- The new entailment is now ready for use in SEM_MATCH queries. --
```

11.14 SEM_APIS.CREATE_RDFVIEW_MODEL

Format

```
SEM_APIS.CREATE_RDFVIEW_MODEL(
     model_name               IN VARCHAR2,
     tables                          IN SYS.ODCIVarchar2List,
     prefix                           IN VARCHAR2 DEFAULT NULL,
     r2rml_table_owner      IN VARCHAR2 DEFAULT NULL,
     r2rml_table_name       IN VARCHAR2 DEFAULT NULL,
     schema_table_owner  IN VARCHAR2 DEFAULT NULL,
     schema_table_name   IN VARCHAR2 DEFAULT NULL,
     options                         IN VARCHAR2 DEFAULT NULL);
```

Description

Creates an RDF. view using direct mapping for the specified list of tables or views or using R2RML mapping.

Parameters

model_name
Name of the RDF view to be created.

tables
List of tables or views that are the sources of relational data for the RDF view to be created using direct mapping. This parameter must be null if you want to use R2RML mapping.

prefix
Base prefix to be added at the beginning of the URIs in the RDF view.

r2rml_table_owner
For R2ML mapping, this parameter is required and specifies the name of the schema that owns the staging table that holds the R2RML mapping (in N-triple format) to be used for creating the RDF view.

For direct mapping, this parameter is optional and specifies the name of the schema that owns the staging table into which the R2RML mapping (in N-triple format) generated from the direct mapping will be stored.

r2rml_table_name
For R2ML mapping, this parameter is required and specifies the name of the staging table that holds the R2RML mapping (in N-triple format) to be used for creating the RDF view.

For direct mapping, this parameter is optional and specifies the name of the staging table into which the R2RML mapping (in N-triple format) generated from the direct mapping will be stored.

schema_table_owner
Name of the schema that owns the staging table where the RDF schema generated for the RDF view will be stored.

schema_table_name
Name of the staging table where the RDF schema generated for the RDF view will be stored.

options
For direct mapping, you can optionally specify any combination (including none) of the following:

- `CONFORMANCE=T` suppresses some of the information that would otherwise get included by default, including use of database constraint names and schema-qualified table or view names for constructing RDF predicate names.

  For more information, see Example 10-2 (page 10-3) in Creating an RDF View with Direct Mapping (page 10-2).

- `GENERATE_ONLY=T` only generates the R2RML mapping for the specified tables and stores it in the specified `r2rml_table_name`, but the underlying RDF view model is not created. If you specify this option, the `r2rml_table_name` parameter must not be null.

- `KEY_BASED_REF_PROPERTY=T` uses the foreign key column names to construct the RDF predicate name. If this option is not specified, then the database constraint name is used for constructing the RDF predicate name.

  For direct mapping, RDF predicate names are derived from the corresponding database names; therefore, preserving the name for the foreign key constraint is the default behavior.

  For an example that uses `KEY_BASED_REF_PROPERTY=T` , see Example 10-1 (page 10-2) in Creating an RDF View with Direct Mapping (page 10-2).

- `SCALAR_COLUMNS_ONLY=T` generates the R2RML mapping for only the scalar columns in the specified tables or views. Other non-scalar columns in the tables or views are ignored. Without this option, if you attempt to create a direct mapping on a table with user-defined types or LOB columns, an error is raised.

Usage Notes

You must grant the `SELECT` and `INSERT` privileges on `r2rml_table_name` and `schema_table_name` to `MDSYS`.

For more information about RDF views, see RDF Views: Relational Data as RDF (page 10-1).

Examples

The following example creates an RDF view using direct mapping for tables EMP and DEPT. The prefix used for the URIs is `http://empdb/`.

```
BEGIN
  sem_apis.create_rdfview_model(
    model_name => 'empdb_model_direct',
    tables => sem_models('EMP', 'DEPT'),
    prefix => 'http://empdb/'
  );
END;
/
```

The following example creates an RDF view using R2RML mapping as specified by the RDF triples in the staging table SCOTT.R2RTAB.

```
BEGIN
  sem_apis.create_rdfview_model(
    model_name => 'empdb_model_R2RML',
    tables => NULL,
    r2rml_table_owner => 'SCOTT',
    r2rml_table_name => 'R2RTAB'
  );
END;
/
```

11.15 SEM_APIS.CREATE_RULEBASE

Format

```
SEM_APIS.CREATE_RULEBASE(
     rulebase_name  IN VARCHAR2);
```

Description

Creates a rulebase.

Parameters

rulebase_name
Name of the rulebase.

Usage Notes

This procedure creates a user-defined rulebase. After creating the rulebase, you can add rules to it. To cause the rules in the rulebase to be applied in a query of RDF data, you can specify the rulebase in the call to the SEM_MATCH table function.

Rules and rulebases are explained in Inferencing: Rules and Rulebases (page 1-8). The SEM_MATCH table function is described in Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22),

Examples

The following example creates a rulebase named family_rb. (It is an excerpt from Example 1-90 (page 1-96) in Example: Family Information (page 1-95).)

```
EXECUTE SEM_APIS.CREATE_RULEBASE('family_rb');
```

11.16 SEM_APIS.CREATE_SEM_MODEL

Format

```
SEM_APIS.CREATE_SEM_MODEL(
     model_name         IN VARCHAR2,
     table_name           IN VARCHAR2,
     column_name       IN VARCHAR2,
     model_tablespace IN VARCHAR2 DEFAULT NULL);
```

Description

Creates a semantic technology model.

Parameters

model_name
Name of the model.

table_name
Name of the table to hold references to semantic technology data for this model.

column_name
Name of the column of type SDO_RDF_TRIPLE_S in `table_name`.

model_tablespace
Name of the tablespace for the tables and other database objects used by Oracle to support this model. The default value is the tablespace that was specified in the call to the SEM_APIS.CREATE_SEM_NETWORK (page 11-28) procedure.

Usage Notes

You must create the table to hold references to semantic technology data before calling this procedure to create the semantic technology model. For more information, see Quick Start for Using Semantic Data (page 1-92).

This procedure adds the model to the MDSYS.SEM_MODEL$ view, which is described in Metadata for Models (page 1-3).

This procedure is the only supported way to create a model. Do not use SQL INSERT statements with the MDSYS.SEM_MODEL$ view.

To delete a model, use the SEM_APIS.DROP_SEM_MODEL (page 11-39) procedure.

Examples

The following example creates a semantic technology model named `articles`. References to the triple data for the model will be stored in the TRIPLE column of the ARTICLES_RDF_DATA table. (This example is an excerpt from Example 1-89 (page 1-94) in Example: Family Information (page 1-95).)

```
EXECUTE SEM_APIS.CREATE_SEM_MODEL('articles', 'articles_rdf_data', 'triple');
```

The definition of the ARTICLES_RDF_DATA table is as follows:

```
CREATE TABLE articles_rdf_data (id NUMBER, triple SDO_RDF_TRIPLE_S);
```

11.17 SEM_APIS.CREATE_SEM_NETWORK

Format

```
SEM_APIS.CREATE_SEM_NETWORK(
     tablespace_name  IN VARCHAR2
```

Description

Creates structures for persistent storage of semantic data.

Parameters

tablespace_name
Name of the tablespace to be used for tables created by this procedure. This tablespace will be the default for all models that you create, although you can override the default when you create a model by specifying the `model_tablespace` parameter in the call to the SEM_APIS.CREATE_SEM_MODEL (page 11-27) procedure.

Usage Notes

This procedure creates system tables and other database objects used for semantic technology support.

You should create a tablespace for the semantic technology system tables and specify the tablespace name in the call to this procedure. (You should *not* specify the `SYSTEM` tablespace.) The size needed for the tablespace that you create will depend on the amount of semantic technology data you plan to store.

You must connect to the database as a user with DBA privileges in order to call this procedure, and you should call the procedure only once for the database.

To drop these structures for persistent storage of semantic data, you must connect as a user with DBA privileges and call the SEM_APIS.DROP_SEM_NETWORK (page 11-39) procedure.

Examples

The following example creates a tablespace for semantic technology system tables and creates structures for persistent storage of semantic data in this tablespace.

```
CREATE TABLESPACE rdf_tblspace
 DATAFILE '/oradata/orcl/rdf_tblspace.dat' SIZE 1024M REUSE
 AUTOEXTEND ON NEXT 256M MAXSIZE UNLIMITED
 SEGMENT SPACE MANAGEMENT AUTO;
. . .
EXECUTE SEM_APIS.CREATE_SEM_NETWORK('rdf_tblspace');
```

11.18 SEM_APIS.CREATE_SOURCE_EXTERNAL_TABLE

Format

```
SEM_APIS.CREATE_SOURCE_EXTERNAL_TABLE(
     source_table   IN VARCHAR2,
     def_directory  IN VARCHAR2,
     log_directory  IN VARCHAR2 DEFAULT NULL,
     bad_directory  IN VARCHAR2 DEFAULT NULL,
     log_file       IN VARCHAR2 DEFAULT NULL,
     bad_file       IN VARCHAR2 DEFAULT NULL,
     parallel       IN INTEGER DEFAULT NULL,
     source_table_owner IN VARCHAR2 DEFAULT NULL,
     flags          IN VARCHAR2 DEFAULT NULL);
```

Description

Creates an external table to map an N-Triple or N-Quad format file into a table.

Parameters

source_table
Name of the external table to be created.

def_directory
Database directory where the input files are located. To load from this staging table, you must have READ privilege on this directory.

log_directory
Database directory where the log files will be generated when loading from the external table. If not specified, the value of the `def_directory` parameter is used. When loading from the external table, you must have WRITE privilege on this directory.

bad_directory
Database directory where the bad files will be generated when loading from the external table. If not specified, the value of the `def_directory` parameter is used. When loading from the external table, you must have WRITE privilege on this directory.

log_file
Name of the log file. If not specified, the name will be .generated automatically during a load operation.

bad_file
Name of the bad file. If not specified, the name will be .generated automatically during a load operation.

parallel
Degree of parallelism to associate with the external table being created.

source_table_owner
Owner for the external table being created. If not specified, the invoker becomes the owner.

flags
(Reserved for future use)

Usage Notes

For more information and an example, see Loading N-Quad Format Data into a Staging Table Using an External Table (page 1-76).

Examples

The following example creates a source external table. (This example is an excerpt from Example 1-82 (page 1-76) in Loading N-Quad Format Data into a Staging Table Using an External Table (page 1-76).)

```
BEGIN
  sem_apis.create_source_external_table(
    source_table    => 'stage_table_source'
   ,def_directory   => 'DATA_DIR'
   ,bad_file        => 'CLOBrows.bad'
   );
END;
```

11.19 SEM_APIS.CREATE_VIRTUAL_MODEL

Format

```
SEM_APIS.CREATE_VIRTUAL_MODEL(
     vm_name     IN VARCHAR2,
     models      IN SEM_MODELS,
     rulebases   IN SEM_RULEBASES DEFAULT NULL,
     options     IN VARCHAR2 DEFAULT NULL,
     entailments IN SEM_ENTAILMENTS DEFAULT NULL);
```

Description

Creates a virtual model containing the specified semantic models and/or entailments. Entailments can be specified in one of the following ways:

- By specifying one or more models and one or more rulebases. In this case, a virtual model will be created using the single entailment that corresponds to the exact combination of models and rulebases specified. An error is raised if no such entailment exists.
- By specifying zero or more models and one or more entailments. In this case, the contents of the models and entailments will be combined regardless of their relationship.

The first method ensures a sound and complete dataset, whereas the second method relaxes the sound and complete constraints for more flexibility.

Parameters

vm_name
Name of the virtual model to be created.

models
One or more semantic model names. Its data type is SEM_MODELS, which has the following definition: `TABLE OF VARCHAR2(25)`. If this parameter is null, no models are included in the virtual model definition.

rulebases
One or more rulebase names. Its data type is SEM_RULEBASES, which has the following definition: `TABLE OF VARCHAR2(25)`. If this parameter is null, no rulebases are included in the virtual model definition. Rules and rulebases are explained in Inferencing: Rules and Rulebases (page 1-8).

If you specify this parameter, you cannot also specify the `entailments` parameter.

options
`REPLACE=T` lets you to replace a virtual model without dropping it. (Using this option is analogous to using CREATE OR REPLACE VIEW with a view.)

entailments
One or more entailment names. Its data type is SEM_ENTAILMENTS, which has the following definition: `TABLE OF VARCHAR2(25)`. If this parameter is null, no entailments are included in the virtual model definition. Entailments are explained in Using OWL Inferencing (page 2-5).

If you specify this parameter, you cannot also specify the `rulebases` parameter.

Usage Notes

For an explanation of virtual models, including usage information, see Virtual Models (page 1-13).

An entailment must exist for each specified combination of semantic model and rulebase.

To create a virtual model, you must either be (A) the owner of each specified model and any corresponding entailments, or (B) a user with DBA privileges.

To replace a virtual model, you must be the owner of the virtual model or a user with DBA privileges.

This procedure creates views with names in the following format:

- SEMV_*vm_name*, which corresponds to a UNION ALL of the triples in each model and entailment. This view may contain duplicates.
- SEMU_*vm_name*, which corresponds to a UNION of the triples in each model and entailment. This view will not contain duplicates (thus, the *U* in SEMU indicates *unique*).

  However, the SEMU_*vm_name* view is not created if the virtual model contains only one semantic model and no entailment.

To use the example in Virtual Models (page 1-13) of a virtual model vm1 created from models m1, m2, m3, and with an entailment created for m1, m2 ,and m3 using the OWLPrime rulebase, this procedure will create the following two views (assuming that m1, m2, and m3, and the OWLPRIME entailment have internal model_id values 1, 2, 3, 4):

```
CREATE VIEW MDSYS.SEMV_VM1 AS
  SELECT start_node_id, p_value_id, canon_end_node_id, end_node_id
  FROM MDSYS.rdf_link$
  WHERE model_id IN (1, 2, 3, 4);

CREATE VIEW MDSYS.SEMU_VM1 AS
  SELECT start_node_id, p_value_id, canon_end_node_id, MAX(end_node_id)
  FROM MDSYS.rdf_link$
  WHERE model_id IN (1, 2, 3, 4)
  GROUP BY start_node_id, p_value_id, canon_end_node_id;
```

The user that invokes this procedure will be the owner of the virtual model and will have SELECT WITH GRANT privileges on the SEMU_*vm_name* and SEMV_*vm_name* views. To query the corresponding virtual model, a user must have select privileges on these views.

Examples

The following example creates a virtual model named `VM1`.

```
EXECUTE sem_apis.create_virtual_model('VM1', sem_models('model_1', 'model_2'),
sem_rulebases('OWLPRIME'));
```

The following example creates a virtual model named `VM1` using the relaxed entailment specification.

```
EXECUTE sem_apis.create_virtual_model('VM1', models=>sem_models('model_1',
'model_2'), entailments=>sem_entailments('entailment1','entailment2'));
```

The following example effectively redefines virtual model VM1 by using the REPLACE=T option.

```
EXECUTE sem_apis.create_virtual_model('VM1', models=>sem_models('model_1',
'model_2'), entailments=>sem_entailments('entailment1'), options=>'REPLACE=T');
```

11.20 SEM_APIS.DELETE_ENTAILMENT_STATS

Format

```
SEM_APIS.DELETE_ENTAILMENT_STATS (
     entailment_name  IN VARCHAR2,
     cascade_parts    IN BOOLEAN DEFAULT TRUE,
     cascade_columns  IN BOOLEAN DEFAULT TRUE,
     cascade_indexes  IN BOOLEAN DEFAULT TRUE,
     no_invalidate    IN BOOLEAN DEFAULT DBMS_STATS.AUTO_INVALIDATE,
     force            IN BOOLEAN DEFAULT FALSE);
```

Description

Deletes statistics for a specified entailment.

Parameters

entailment_name
Name of the entailment.

(other parameters)
See the parameter explanations for the DBMS_STATS.DELETE_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference,* although `force` here applies to entailment statistics.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example deletes statistics for an entailment named OWLTST_IDX.

```
EXECUTE SEM_APIS.DELETE_ENTAILMENT_STATS('owltst_idx');
```

11.21 SEM_APIS.DELETE_MODEL_STATS

Format

```
SEM_APIS.DELETE_MODEL_STATS (
     model_name      IN VARCHAR2,
     cascade_parts   IN BOOLEAN DEFAULT TRUE,
     cascade_columns IN BOOLEAN DEFAULT TRUE,
     cascade_indexes IN BOOLEAN DEFAULT TRUE,
     no_invalidate   IN BOOLEAN DEFAULT DBMS_STATS.AUTO_INVALIDATE,
     force           IN BOOLEAN DEFAULT FALSE);
```

Description

Deletes statistics for a specified model.

Parameters

model_name
Name of the model.

(other parameters)
See the parameter explanations for the DBMS_STATS.DELETE_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference*, although `force` here applies to model statistics.

Usage Notes

Only the model owner or a users with DBA privileges can execute this procedure.

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example deletes statistics for a model named `FAMILY`.

```
EXECUTE SEM_APIS.DELETE_MODEL_STATS('family');
```

11.22 SEM_APIS.DISABLE_CHANGE_TRACKING

Format

```
SEM_APIS.DISABLE_CHANGE_TRACKING(
     models_in IN SEM_MODELS);
```

Description

Disables change tracking for a specified set of models.

Parameters

models_in
One or more model names. Its data type is SEM_MODELS, which has the following definition: `TABLE OF VARCHAR2(25)`

Usage Notes

Disabling change tracking on a model automatically disables incremental inference on all entailment that use the model.

To use this procedure, you must be the owner of the specified model, and incremental inference must have been previously enabled.

For an explanation of incremental inference, including usage information, see Performing Incremental Inference (page 2-13).

Examples

The following example disables change tracking for the `family` model.

```
EXECUTE sem_apis.disable_change_tracking(sem_models('family'));
```

11.23 SEM_APIS.DISABLE_INC_INFERENCE

Format

```
SEM_APIS.DISABLE_INC_INFERENCE(
     entailment_name IN VARCHAR2);
```

Description

Disables incremental inference for a specified entailment (rules index).

Parameters

entailment_name
Name of the entailment for which to disable incremental inference.

Usage Notes

To use this procedure, you must be the owner of the specified entailment, and incremental inference must have been previously enabled by the SEM_APIS.ENABLE_INC_INFERENCE (page 11-42) procedure.

Calling this procedure automatically disables change tracking for all models owned by the invoking user that were having changes tracked only because of this particular inference.

For an explanation of incremental inference, including usage information, see Performing Incremental Inference (page 2-13).

Examples

The following example enables incremental inference for the entailment named `RDFS_RIX_FAMILY`.

```
EXECUTE sem_apis.disable_inc_inference('rdfs_rix_family');
```

11.24 SEM_APIS.DROP_DATATYPE_INDEX

Format

```
SEM_APIS.DROP_DATATYPE_INDEX(
     datatype    IN VARCHAR2,
     force_drop  IN BOOLEAN default FALSE);
```

Description

Drops (deletes) an existing data type index.

Parameters

datatype
URI of the data type for the index to drop.

force_drop
TRUE forces the index to be dropped if an error occurs during the processing of the statement; FALSE (the default) does not drop the index if an error occurs during the processing of the statement.

Usage Notes

You must have DBA privileges to call this procedure.

For an explanation of data type indexes, see Using Data Type Indexes (page 1-88).

Examples

The following example drops the data type index for `xsd:string` typed literals and plain literals.

```
EXECUTE SEM_APIS.DROP_DATATYPE_INDEX('http://www.w3.org/2001/XMLSchema#string');
```

11.25 SEM_APIS.DROP_ENTAILMENT

Format

```
SEM_APIS.DROP_ENTAILMENT(
     entailment_name_in IN VARCHAR2,
     named_g_in         IN SEM_GRAPHS DEFAULT NULL,
     dop                IN INT DEFAULT 1);
```

Description

Drops (deletes) an entailment (rules index).

Parameters

entailment_name_in
Name of the entailment to be deleted.

named_g_in
Causes only the triples with the specified graph names in the entailment to be deleted. A null value (the default) drops the entire entailment.

For example, `named_g_in => sem_graphs('<urn:G1>','<urn:G2>')` drops only the triples in entailment with graph names `G1` and `G2`; the rest of the entailment graph is not dropped.

dop
Degree of parallelism for a parallel execution of triple deletion. Applies only if the `named_g_in` parameter is not null.

Usage Notes

You can use this procedure to delete an entailment that you created using the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure.

If you drop only a subset of the entailment with specified named graphs (that is, when `named_g_in` is not null) on an entailment with a `VALID` or `INCOMPLETE` status, then the resulting status of the entailment after the drop is set to `INCOMPLETE`.

Examples

The following example deletes a entailment named `OWLTST_IDX`.

```
EXECUTE sem_apis.drop_entailment('owltst_idx');
```

The following example deletes only inferred triples with graph names `G1` and `G2` that belong to the entailment named `OWLNG_IDX`. Any inferred triples in the default graph and other named graphs remain in the entailment.

```
EXECUTE sem_apis.drop_entailment('owlng_idx',sem_graphs('<urn:G1>','<urn:G2>'));
```

11.26 SEM_APIS.DROP_RDFVIEW_MODEL

Format

```
SEM_APIS.DROP_RDFVIEW_MODEL(
     model_name IN VARCHAR2,
     options    IN VARCHAR2 DEFAULT NULL);
```

Description

Drops (deletes) an RDF view.

Parameters

model_name
Name of the RDF view to be dropped.

options
(Reserved for future use.)

Usage Notes

You must be the owner of the RDF view to be dropped.

For more information about RDF views, see RDF Views: Relational Data as RDF (page 10-1).

Examples

The following example drops an RDF view.

```
BEGIN
  sem_apis.drop_rdfview_model(
    model_name => 'empdb_model'
  );
END;
/
```

11.27 SEM_APIS.DROP_RULEBASE

Format

```
SEM_APIS.DROP_RULEBASE(
     rulebase_name  IN VARCHAR2);
```

Description

Deletes a rulebase.

Parameters

rulebase_name
Name of the rulebase.

Usage Notes

This procedure deletes the specified rulebase, making it no longer available for use in calls to the SEM_MATCH table function. For information about rulebases, see Inferencing: Rules and Rulebases (page 1-8).

Only the creator of a rulebase can delete the rulebase.

Examples

The following example drops the rulebase named `family_rb`.

```
EXECUTE SEM_APIS.DROP_RULEBASE('family_rb');
```

11.28 SEM_APIS.DROP_SEM_INDEX

Format

```
SEM_APIS.DROP_SEM_INDEX(
     index_code  IN VARCHAR2);
```

Description

Drops a semantic network index on the models and entailments of the semantic network.

Parameters

index_code
Index code string. Must match the `index_code` value that was specified in an earlier call to the SEM_APIS.ADD_SEM_INDEX (page 11-2) procedure.

Usage Notes

For an explanation of semantic network indexes, see Using Semantic Network Indexes (page 1-86).

Examples

The following example drops a semantic network index with the index code string `pcsm` on the models and entailments of the semantic network.

```
EXECUTE SEM_APIS.DROP_SEM_INDEX('pscm');
```

11.29 SEM_APIS.DROP_SEM_MODEL

Format

```
SEM_APIS.DROP_SEM_MODEL(
     model_name  IN VARCHAR2);
```

Description

Drops (deletes) a semantic technology model.

Parameters

model_name
Name of the model.

Usage Notes

This procedure deletes the model from the MDSYS.SEM_MODEL$ view, which is described in Metadata for Models (page 1-3).

This procedure is the only supported way to delete a model. Do not use SQL DELETE statements with the MDSYS.SEM_MODEL$ view.

Only the creator of a model can delete the model.

Examples

The following example drops the semantic technology model named `articles`.

```
EXECUTE SEM_APIS.DROP_SEM_MODEL('articles');
```

11.30 SEM_APIS.DROP_SEM_NETWORK

Format

```
SEM_APIS.DROP_SEM_NETWORK(
     cascade  IN BOOLEAN DEFAULT FALSE);
```

Description

Removes structures used for persistent storage of semantic data.

Parameters

cascade
TRUE drops any existing semantic technology models and rulebases, and removes structures used for persistent storage of semantic data; FALSE (the default) causes the operation to fail if any semantic technology models or rulebases exist.

Usage Notes

To remove structures used for persistent storage of semantic data, you must connect as a user with DBA privileges and call this procedure.

If any version-enabled models exist, this procedure will fail regardless of the value of the `cascade` parameter.

Examples

The following example removes structures used for persistent storage of semantic data.

```
EXECUTE SEM_APIS.DROP_SEM_NETWORK;
```

11.31 SEM_APIS.DROP_USER_INFERENCE_OBJS

Format

```
SEM_APIS.DROP_USER_INFERENCE_OBJS(
     uname  IN VARCHAR2);
```

Description

Drops (deletes) all rulebases and entailments owned by a specified database user.

Parameters

uname
Name of a database user. (This value is case-sensitive; for example, `HERMAN` and `herman` are considered different users.)

Usage Notes

You must have sufficient privileges to delete rules and rulebases for the specified user.

This procedure does not delete the database user. It deletes only RDF rulebases and entailments owned by that user.

Examples

The following example deletes all rulebases and entailments owned by user `SCOTT`.

```
EXECUTE SEM_APIS.DROP_USER_INFERENCE_OBJS('SCOTT');

PL/SQL procedure successfully completed.
```

11.32 SEM_APIS.DROP_VIRTUAL_MODEL

Format

```
SEM_APIS.DROP_VIRTUAL_MODEL(
     vm_name  IN VARCHAR2);
```

Description

Drops (deletes) a virtual model.

Parameters

vm_name
Name of the virtual model to be deleted.

Usage Notes

You can use this procedure to delete a virtual model that you created using the SEM_APIS.CREATE_VIRTUAL_MODEL (page 11-31) procedure. A virtual model is deleted automatically if any of its component models, rulebases, or entailment are deleted.

To use this procedure, you must be the owner of the specified virtual model.

For an explanation of virtual models, including usage information, see Virtual Models (page 1-13).

Examples

The following example deletes a virtual model named `VM1`.

```
EXECUTE sem_apis.drop_virtual_model('VM1');
```

11.33 SEM_APIS.ENABLE_CHANGE_TRACKING

Format

```
SEM_APIS.ENABLE_CHANGE_TRACKING(
     models_in IN SEM_MODELS);
```

Description

Enables change tracking for a specified set of models.

Parameters

models_in
One or more model names. Its data type is SEM_MODELS, which has the following definition: `TABLE OF VARCHAR2(25)`

Usage Notes

Change tracking must be enabled on a model before incremental inference can be enabled on any entailments that use the model.

To use this procedure, you must be the owner of the specified model or models.

If the owner of an entailment is also an owner of any underlying models, then enabling incremental inference on the entailment (by calling the SEM_APIS.ENABLE_INC_INFERENCE (page 11-42) procedure) automatically enables change tracking on those models owned by that user.

To disable change tracking for a set of models, use the SEM_APIS.DISABLE_CHANGE_TRACKING (page 11-34) procedure.

For an explanation of incremental inference, including usage information, see Performing Incremental Inference (page 2-13).

Examples

The following example enables change tracking for the `family` model.

```
EXECUTE sem_apis.enable_change_tracking(sem_models('family'));
```

11.34 SEM_APIS.ENABLE_INC_INFERENCE

Format

```
SEM_APIS.ENABLE_INC_INFERENCE(
     entailment_name IN VARCHAR2);
```

Description

Enables incremental inference for a specified entailment (rules index).

Parameters

entailment_name
Name of the entailment for which to enable incremental inference.

Usage Notes

To use this procedure, you must be the owner of the specified entailment.

Before this procedure is executed, all underlying models involved in the entailment must have change tracking enabled. If the owner of the entailment is also an owner of any underlying models, calling this procedure automatically enables change tracking on those models. However, if some underlying model are not owned by the owner of the entailment, the appropriate model owners must first call the SEM_APIS.ENABLE_CHANGE_TRACKING (page 11-41) procedure to enable change tracking on those models.

To disable incremental inference for an entailment, use the SEM_APIS.DISABLE_INC_INFERENCE (page 11-35) procedure.

For an explanation of incremental inference, including usage information, see Performing Incremental Inference (page 2-13).

Examples

The following example enables incremental inference for the entailment named `RDFS_RIX_FAMILY`.

```
EXECUTE sem_apis.enable_inc_inference('rdfs_rix_family');
```

11.35 SEM_APIS.ESCAPE_CLOB_TERM

Format

```
SEM_APIS.ESCAPE_CLOB_TERM(
     term        IN  CLOB CHARACTER SET ANY_CS,
     utf_encode  IN NUMBER DEFAULT 1
     ) RETURN CLOB CHARACTER SET val%CHARSET;
```

Description

Returns the input RDF term with special characters and non-ASCII characters escaped as specified by the W3C N-Triples format (http://www.w3.org/TR/rdf-testcases/#ntriples).

Parameters

term
The RDF term to escape.

utf_encode
Set to 1 (the default) if non-ASCII characters and non-printable ASCII characters other than chr(8), chr(9), chr(10), chr(12), and chr(13) should be escaped. Otherwise, such characters will not be escaped.

Usage Notes

For information about using the `DO_UNESCAPE` keyword in the `options` parameter of the SEM_MATCH table function, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).

Examples

The following example escapes an input RDF term containing TAB and NEWLINE characters.

```
SELECT SEM_APIS.ESCAPE_CLOB_TERM('"abc' || chr(9) || 'def' || chr(10) ||
'hij"^^<http://www.w3.org/2001/XMLSchema#string>')
  FROM DUAL;
```

11.36 SEM_APIS.ESCAPE_CLOB_VALUE

Format

```
SEM_APIS.ESCAPE_CLOB_VALUE(
     val          IN  CLOB CHARACTER SET ANY_CS,
     start_offset IN NUMBER DEFAULT 1,
     end_offset   IN NUMBER DEFAULT 0,
     utf_encode   IN NUMBER DEFAULT 1,
      include_start IN NUMBER DEFAULT 0
     ) RETURN VARCHAR2 CHARACTER SET val%CHARSET;
```

Description

Returns the input CLOB value with special characters and non-ASCII characters escaped as specified by the W3C N-Triples format (http://www.w3.org/TR/rdf-testcases/#ntriples).

Parameters

val
The CLOB text to escape.

start_offset
The offset in `val` from which to start character escaping. The default (1) causes escaping to start at the first character of `val`.

end_offset
The offset in `val` from which to end character escaping. The default (0) causes escaping to continue through the end of `val`.

utf_encode
Set to 1 (the default) if non-ASCII characters and non-printable ASCII characters other than chr(8), chr(9), chr(10), chr(12), and chr(13) should be escaped. Otherwise, such characters will not be escaped.

include_start
Set to 1 if the characters in val from 1 to `start_offset` should be prefixed (prepended) to the return value. Otherwise, no such characters will be prefixed to the return value.

Usage Notes

For information about using the `DO_UNESCAPE` keyword in the `options` parameter of the SEM_MATCH table function, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).

Examples

The following example escapes an input character string containing TAB and NEWLINE characters.

```
SELECT SEM_APIS.ESCAPE_CLOB_VALUE('abc' || chr(9) || 'def' || chr(10) || 'hij')
  FROM DUAL;
```

11.37 SEM_APIS.ESCAPE_RDF_TERM

Format

```
SEM_APIS.ESCAPE_RDF_TERM(
     term       IN  VARCHAR2 CHARACTER SET ANY_CS,
     utf_encode IN NUMBER DEFAULT 1
     ) RETURN VARCHAR2 CHARACTER SET val%CHARSET;
```

Description

Returns the input RDF term with special characters and non-ASCII characters escaped as specified by the W3C N-Triples format (`http://www.w3.org/TR/rdf-testcases/#ntriples`).

Parameters

term
The RDF term to escape.

utf_encode
Set to 1 (the default) if non-ASCII characters and non-printable ASCII characters other than chr(8), chr(9), chr(10), chr(12), and chr(13) should be escaped. Otherwise, such characters will not be escaped.

Usage Notes

For information about using the `DO_UNESCAPE` keyword in the `options` parameter of the SEM_MATCH table function, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).

Examples

The following example escapes an input RDF term containing TAB and NEWLINE characters.

```
SELECT SEM_APIS.ESCAPE_RDF_TERM('"abc' || chr(9) || 'def' || chr(10) ||
'hij"^^<http://www.w3.org/2001/XMLSchema#string>')
  FROM DUAL;
```

11.38 SEM_APIS.ESCAPE_RDF_VALUE

Format

```
SEM_APIS.ESCAPE_RDF_VALUE(
     val         IN  VARCHAR2 CHARACTER SET ANY_CS,
     utf_encode  IN NUMBER DEFAULT 1
     ) RETURN VARCHAR2 CHARACTER SET val%CHARSET;
```

Description

Returns the input CLOB value with special characters and non-ASCII characters escaped as specified by the W3C N-Triples format (`http://www.w3.org/TR/rdf-testcases/#ntriples`).

Parameters

val
The text to escape.

utf_encode
Set to 1 (the default) if non-ASCII characters and non-printable ASCII characters other than chr(8), chr(9), chr(10), chr(12), and chr(13) should be escaped. Otherwise, such characters will not be escaped.

Usage Notes

For information about using the `DO_UNESCAPE` keyword in the `options` parameter of the SEM_MATCH table function, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).

Examples

The following example escapes an input character string containing TAB and NEWLINE characters.

```
SELECT SEM_APIS.ESCAPE_RDF_VALUE('abc' || chr(9) || 'def' || chr(10) || 'hij')
  FROM DUAL;
```

11.39 SEM_APIS.EXPORT_ENTAILMENT_STATS

Format

```
SEM_APIS.EXPORT_ENTAILMENT_STATS (
     entailment_name IN VARCHAR2,
     stattab         IN VARCHAR2,
     statid          IN VARCHAR2 DEFAULTNULL,
     cascade         IN BOOLEAN DEFAULT TRUE,
     statown         IN VARCHAR2 DEFAULT NULL,
     stat_category   IN VARCHAR2 DEFAULT 'OBJECT_STATS');
```

Description

Exports statistics for a specified entailment and stores them in the user statistics table.

Parameters

entailment_name
Name of the entailment.

(other parameters)
See the parameter explanations for the DBMS_STATS.EXPORT_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference*, although `force` here applies to entailment statistics.

Specifying `cascade` also exports all index statistics associated with the entailment.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example exports statistics for an entailment named `OWLTST_IDX` and stores them in a table named `STAT_TABLE`.

```
EXECUTE SEM_APIS.EXPORT_ENTAILMENT_STATS('owltst_idx', 'stat_table');
```

11.40 SEM_APIS.EXPORT_MODEL_STATS

Format

```
SEM_APIS.EXPORT_MODEL_STATS (
     model_name   IN VARCHAR2,
     stattab      IN VARCHAR2,
     statid       IN VARCHAR2 DEFAULT NULL,
     cascade      IN BOOLEAN DEFAULT TRUE,
     statown      IN VARCHAR2 DEFAULT NULL,
     stat_category IN VARCHAR2 DEFAULT 'OBJECT_STATS');
```

Description

Exports statistics for a specified model and stores them in the user statistics table.

Parameters

entailment_name
Name of the entailment.

(other parameters)
See the parameter explanations for the DBMS_STATS.EXPORT_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference*.

Specifying `cascade` also exports all index statistics associated with the model.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example exports statistics for a model named `FAMILY` and stores them in a table named `STAT_TABLE`.

```
EXECUTE SEM_APIS.EXPORT_MODEL_STATS('family', 'stat_table');
```

11.41 SEM_APIS.EXPORT_RDFVIEW_MODEL

Format

```
SEM_APIS.EXPORT_RDFVIEW_MODEL(
     model_name       IN VARCHAR2,
     rdf_table_owner  IN VARCHAR2 DEFAULT NULL,
     rdf_table_name   IN VARCHAR2 DEFAULT NULL,
     options          IN VARCHAR2 DEFAULT NULL);
```

Description

Exports (materializes) the virtual RDF triples of an RDF view to a staging table.

Parameters

model_name
Name of the RDF view to be exported.

rdf_table_owner
Name of the schema that owns the staging table where the RDF triples obtained from the RDF view are to be stored.

rdf_table_name
Name of the staging table where the RDF triples obtained from the RDF view are to be stored.

options
(Reserved for future use)

Usage Notes

You must have the SELECT privilege for the database view SEMM_<*model_name*>.

For more information about RDF views, see RDF Views: Relational Data as RDF (page 10-1). For information about exporting RDF views, see Exporting Virtual Content of an RDF View into a Staging Table (page 10-5).

Examples

The following example exports RDF triples from RDF view `empdb_model` to the staging table SCOTT.RDFTAB.

```
BEGIN
  sem_apis.export_rdfview_model(
    model_name => 'empdb_model',
    rdf_table_owner => 'SCOTT',
    rdf_table_name => 'RDFTAB'
  );
END;
/
```

11.42 SEM_APIS.GET_CHANGE_TRACKING_INFO

Format

```
SEM_APIS.GET_CHANGE_TRACKING_INFO(
     model_name           IN VARCHAR2,
     enabled              OUT BOOLEAN,
     tracking_start_time  OUT TIMESTAMP);
```

Description

Returns change tracking information for a model.

Parameters

model_name
Name of the semantic technology model.

enabled
Boolean value returned by the procedure: `TRUE` if change tracking is enabled for the model, or `FALSE` if change tacking is not enabled for the model.

timestamp
Timestamp indicating when change tracking was enabled for the model (if it is enabled).

Usage Notes

The `model_name` value must match a value in the MODEL_NAME column in the MDSYS.SEM_MODEL$ view, which is described in Metadata for Models (page 1-3).

To enable change tracking for a set of models, use the SEM_APIS.ENABLE_CHANGE_TRACKING (page 11-41) procedure.

For an explanation of incremental inference, including usage information, see Performing Incremental Inference (page 2-13).

Examples

The following example displays change tracking information for a model.

```
DECLARE
  bEnabled  boolean;
  tsEnabled  timestamp;

BEGIN
  EXECUTE IMMEDIATE 'create table m1 (t SDO_RDF_TRIPLE_S)';
  sem_apis.create_sem_model('m1','m1','t');

  sem_apis.enable_change_tracking(sem_models('m1'));

  sem_apis.get_change_tracking_info('m1', bEnabled,  tsEnabled);
  dbms_output.put_line('is enabled:' || case when bEnabled then 'true' else 'false'
end);
  dbms_output.put_line('enabled at:' || tsEnabled);
END;
/
```

11.43 SEM_APIS.GET_INC_INF_INFO

Format

```
SEM_APIS.GET_INC_INF_INFO(
     entailment_name     IN VARCHAR2,
     enabled             OUT BOOLEAN,
     prev_inf_start_time OUT TIMESTAMP);
```

Description

Returns incremental inference information for an entailment.

Parameters

entailment_name
Name of the entailment.

enabled
Boolean value returned by the procedure: TRUE if incremental inference is enabled for the entailment, or FALSE if incremental inference is not enabled for the entailment.

timestamp
Timestamp indicating when the entailment was most recently updated (if incremental inference is enabled).

Usage Notes

To enable incremental inference for an entailment, use the SEM_APIS.ENABLE_INC_INFERENCE (page 11-42) procedure.

For an explanation of incremental inference, including usage information, see Performing Incremental Inference (page 2-13).

Examples

The following example displays incremental inference information for an entailment.

```
DECLARE
  bEnabled boolean;
  tsEnabled timestamp;

DECLARE
  EXECUTE IMMEDIATE 'create table m1 (t SDO_RDF_TRIPLE_S)';
  sem_apis.create_sem_model('m1','m1','t');

  sem_apis.create_entailment('m1_inf',sem_models('m1'),
sem_rulebases('owlprime'),null,null,'INC=T');

  sem_apis.get_inc_inf_info('m1_inf', bEnabled,  tsEnabled);
  dbms_output.put_line('is enabled:' || case when bEnabled then 'true' else 'false'
  end);
  dbms_output.put_line('enabled at:' || tsEnabled);
END
/
```

11.44 SEM_APIS.GET_MODEL_ID

Format

```
SEM_APIS.GET_MODEL_ID(
     model_name  IN VARCHAR2
     ) RETURN NUMBER;
```

Description

Returns the model ID number of a semantic technology model.

Parameters

model_name
Name of the semantic technology model.

Usage Notes

The `model_name` value must match a value in the MODEL_NAME column in the MDSYS.SEM_MODEL$ view, which is described in Metadata for Models (page 1-3).

Examples

The following example returns the model ID number for the model named `articles`. (This example is an excerpt from Example 1-89 (page 1-94) in Example: Family Information (page 1-95).)

```
SELECT SEM_APIS.GET_MODEL_ID('articles') AS model_id FROM DUAL;

  MODEL_ID
----------
         1
```

11.45 SEM_APIS.GET_MODEL_NAME

Format

```
SEM_APIS.GET_MODEL_NAME(
     model_id  IN NUMBER
     ) RETURN VARCHAR2;
```

Description

Returns the model name of a semantic technology model.

Parameters

model_id
ID number of the semantic technology model.

Usage Notes

The `model_id` value must match a value in the MODEL_ID column in the MDSYS.SEM_MODEL$ view, which is described in Metadata for Models (page 1-3).

Examples

The following example returns the model ID number for the model with the ID value of 1. This example is an excerpt from Example 1-89 (page 1-94) in Example: Family Information (page 1-95).)

```
SQL> SELECT SEM_APIS.GET_MODEL_NAME(1) AS model_name FROM DUAL;

MODEL_NAME
--------------------------------------------------------------------------------
ARTICLES
```

11.46 SEM_APIS.GET_TRIPLE_ID

Format

```
SEM_APIS.GET_TRIPLE_ID(
     model_id  IN NUMBER,
     subject   IN VARCHAR2,
     property  IN VARCHAR2,
     object    IN VARCHAR2
     ) RETURN VARCHAR2;
```

or

```
SEM_APIS.GET_TRIPLE_ID(
     model_name IN VARCHAR2,
     subject    IN VARCHAR2,
     property   IN VARCHAR2,
     object     IN VARCHAR2
     ) RETURN VARCHAR2;
```

Description

Returns the ID of a triple in the specified semantic technology model, or a null value if the triple does not exist.

Parameters

model_id
ID number of the semantic technology model. Must match a value in the MODEL_ID column of the MDSYS.SEM_MODEL$ view, which is described in Metadata for Models (page 1-3).

model_name
Name of the semantic technology model. Must match a value in the MODEL_NAME column of the MDSYS.SEM_MODEL$ view, which is described in Metadata for Models (page 1-3).

subject
Subject. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

property
Property. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

object
Object. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

Usage Notes

This function has two formats, enabling you to specify the semantic technology model by its model number or its name.

Examples

The following example returns the ID number of a triple. (This example is an excerpt from Example 1-89 (page 1-94) in Example: Family Information (page 1-95).)

```
SELECT SEM_APIS.GET_TRIPLE_ID(
  'articles',
  'http://nature.example.com/Article2',
  'http://purl.org/dc/terms/references',
  'http://nature.example.com/Article3') AS RDF_triple_id FROM DUAL;

RDF_TRIPLE_ID
--------------------------------------------------------------------------------
2_9F2BFF05DA0672E_90D25A8B08C653A_46854582F25E8AC5
```

11.47 SEM_APIS.GETV$DATETIMETZVAL

Format

```
SEM_APIS.GETV$DATETIMETZVAL(
     value_type     IN VARCHAR2,
     vname_prefix   IN VARCHAR2,
     vname_suffix   IN VARCHAR2,
```

```
     literal_type   IN VARCHAR2,
     language_type  IN VARCHAR2,
     ) RETURN NUMBER;
```

Description

Returns a TIMESTAMP WITH TIME ZONE value for xsd:dateTime typed literals, and returns a null value for all other RDF terms. Greenwich Mean Time is used as the default time zone for xsd:dateTime values without time zones.

Parameters

value_type
Type of the RDF term.

vname_prefix
Prefix value of the RDF term.

vname_suffix
Suffix value of the RDF term.

literal_type
Literal type of the RDF term.

language_type
Language type of the RDF term.

Usage Notes

For better performance, consider creating a function-based index on this function. For more information, see Function-Based Indexes for FILTER Constructs Involving Typed Literals (page 1-70).

Examples

The following example returns TIMESTAMP WITH TIME ZONE values for all xsd:dateTime literals in the MDSYS.RDF_VALUE$ table:

```
SELECT SEM_APIS.GETV$DATETIMETZVAL(value_type, vname_prefix, vname_suffix,
  literal_type, language_type)
  FROM MDSYS.RDF_VALUE$;
```

11.48 SEM_APIS.GETV$DATETZVAL

Format

```
SEM_APIS.GETV$DATETZVAL(
     value_type     IN VARCHAR2,
     vname_prefix   IN VARCHAR2,
     vname_suffix   IN VARCHAR2,
     literal_type   IN VARCHAR2,
     language_type  IN VARCHAR2,
     ) RETURN TIMESTAMP WITH TIME ZONE;
```

Description

Returns a TIMESTAMP WITH TIME ZONE value for xsd:date typed literals, and returns a null value for all other RDF terms. Greenwich Mean Time is used as the default time zone for xsd:date values without time zones.

Parameters

value_type
Type of the RDF term.

vname_prefix
Prefix value of the RDF term.

vname_suffix
Suffix value of the RDF term.

literal_type
Literal type of the RDF term.

language_type
Language type of the RDF term.

Usage Notes

For better performance, consider creating a function-based index on this function. For more information, see Function-Based Indexes for FILTER Constructs Involving Typed Literals (page 1-70).

Examples

The following example returns TIMESTAMP WITH TIME ZONE values for all xsd:date literals in the MDSYS.RDF_VALUE$ table:

```
SELECT SEM_APIS.GETV$DATETZVAL(value_type, vname_prefix, vname_suffix,
  literal_type, language_type)
  FROM MDSYS.RDF_VALUE$;
```

11.49 SEM_APIS.GETV$NUMERICVAL

Format

```
SEM_APIS.GETV$NUMERICVAL(
     value_type     IN VARCHAR2,
     vname_prefix   IN VARCHAR2,
     vname_suffix   IN VARCHAR2,
     literal_type   IN VARCHAR2,
     language_type  IN VARCHAR2,
     ) RETURN NUMBER;
```

Description

Returns a numeric value for XML Schema numeric typed literals, and returns a null value for all other RDF terms.

Parameters

value_type
Type of the RDF term.

vname_prefix
Prefix value of the RDF term.

vname_suffix
Suffix value of the RDF term.

literal_type
Literal type of the RDF term.

language_type
Language type of the RDF term.

Usage Notes

For better performance, consider creating a function-based index on this function. For more information, see Function-Based Indexes for FILTER Constructs Involving Typed Literals (page 1-70).

Examples

The following example returns numeric values for all numeric literals in the MDSYS.RDF_VALUE$ table:

```
SELECT SEM_APIS.GETV$NUMERICVAL(value_type, vname_prefix, vname_suffix,
  literal_type, language_type)
  FROM MDSYS.RDF_VALUE$;
```

11.50 SEM_APIS.GETV$STRINGVAL

Format

```
SEM_APIS.GETV$STRINGVAL(
     value_type     IN VARCHAR2,
     vname_prefix   IN VARCHAR2,
     vname_suffix   IN VARCHAR2,
     literal_type   IN VARCHAR2,
     language_type  IN VARCHAR2,
     ) RETURN TIMESTAMP WITH TIME ZONE;
```

Description

Returns a VARCHAR2 string of the lexical form of plain literals and xsd:string typed literals, and returns a null value for all other RDF terms. CHR(0) is returned for empty literals.

Parameters

value_type
Type of the RDF term.

vname_prefix
Prefix value of the RDF term.

vname_suffix
Suffix value of the RDF term.

literal_type
Literal type of the RDF term.

language_type
Language type of the RDF term.

Usage Notes

For better performance, consider creating a function-based index on this function. For more information, see Function-Based Indexes for FILTER Constructs Involving Typed Literals (page 1-70).

Examples

The following example returns lexical values for all plain literals and xsd:string literals in the MDSYS.RDF_VALUE$ table:

```
SELECT SEM_APIS.GETV$STRINGVAL(value_type, vname_prefix, vname_suffix,
  literal_type, language_type)
  FROM MDSYS.RDF_VALUE$;
```

11.51 SEM_APIS.GETV$TIMETZVAL

Format

```
SEM_APIS.GETV$TIMETZVAL(
     value_type     IN VARCHAR2,
     vname_prefix   IN VARCHAR2,
     vname_suffix   IN VARCHAR2,
     literal_type   IN VARCHAR2,
     language_type  IN VARCHAR2,
     ) RETURN TIMESTAMP WITH TIME ZONE;
```

Description

Returns a TIMESTAMP WITH TIME ZONE value for xsd:time typed literals, and returns a null value for all other RDF terms. Greenwich Mean Time is used as the default time zone for xsd:time values without time zones. 2009-06-26 is used as the default date in all the generated TIMESTAMP WITH TIME ZONE values.

Parameters

value_type
Type of the RDF term.

vname_prefix
Prefix value of the RDF term.

vname_suffix
Suffix value of the RDF term.

literal_type
Literal type of the RDF term.

language_type
Language type of the RDF term.

Usage Notes

For better performance, consider creating a function-based index on this function. For more information, see Function-Based Indexes for FILTER Constructs Involving Typed Literals (page 1-70).

Because xsd:time values include only a time but not a date, the returned TIMESTAMP WITH TIME ZONE values (which include a date component) have 2009-06-26 added as the date. This is done so that the returned values can be indexed internally, and so that the date is the same for all of them.

Examples

The following example returns TIMESTAMP WITH TIME ZONE values (using the default 2009-06-26 for the date) for all xsd:time literals in the MDSYS.RDF_VALUE$ table. (

```
SELECT SEM_APIS.GETV$DATETIMETZVAL(value_type, vname_prefix, vname_suffix,
  literal_type, language_type)
  FROM MDSYS.RDF_VALUE$;
```

11.52 SEM_APIS.IMPORT_ENTAILMENT_STATS

Format

```
SEM_APIS.IMPORT_ENTAILMENT_STATS (
     entailment_name  IN VARCHAR2,
     stattab          IN VARCHAR2,
     statid           IN VARCHAR2 DEFAULT NULL,
     cascade          IN BOOLEAN DEFAULT TRUE,
     statown          IN VARCHAR2 DEFAULT NULL,
     no_invalidate    IN BOOLEAN DEFAULT FALSE,
     force            IN BOOLEAN DEFAULT FALSE,
     stat_category    IN VARCHAR2 DEFAULT 'OBJECT_STATS');
```

Description

Retrieves statistics for an entailment from a user statistics table and stores them in the dictionary.

Parameters

entailment_name
Name of the entailment.

(other parameters)
See the parameter explanations for the DBMS_STATS.IMPORT_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference*, although `force` here applies to entailment statistics.

Specifying `cascade` also exports all index statistics associated with the model.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example imports statistics for an entailment named `OWLTST_IDX` from a table named `STAT_TABLE`.

```
EXECUTE SEM_APIS.IMPORT_ENTAILMENT_STATS('owltst_idx', 'stat_table');
```

11.53 SEM_APIS.IMPORT_MODEL_STATS

Format

```
SEM_APIS.IMPORT_MODEL_STATS (
     model_name    IN VARCHAR2,
     stattab       IN VARCHAR2,
     statid        IN VARCHAR2 DEFAULT NULL,
     cascade       IN BOOLEAN DEFAULT TRUE,
     statown       IN VARCHAR2 DEFAULT NULL,
     no_invalidate IN BOOLEAN DEFAULT FALSE,
     force         IN BOOLEAN DEFAULT FALSE,
     stat_category IN VARCHAR2 DEFAULT 'OBJECT_STATS);
```

Description

Retrieves statistics for a specified model from a user statistics table and stores them in the dictionary.

Parameters

model_name
Name of the entailment.

(other parameters)
See the parameter explanations for the DBMS_STATS.IMPORT_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference*.

Specifying `cascade` also imports all index statistics associated with the model.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example imports statistics for a model named `FAMILY` from a table named `STAT_TABLE`, and stores them in the dictionary.

```
EXECUTE SEM_APIS.IMOPRT_MODEL_STATS('family', 'stat_table');
```

11.54 SEM_APIS.IS_TRIPLE

Format

```
SEM_APIS.IS_TRIPLE(
     model_id  IN NUMBER,
     subject   IN VARCHAR2,
     property  IN VARCHAR2,
     object    IN VARCHAR2) RETURN VARCHAR2;
```

or

```
SEM_APIS.IS_TRIPLE(
     model_name IN VARCHAR2,
     subject    IN VARCHAR2,
     property   IN VARCHAR2,
     object     IN VARCHAR2) RETURN VARCHAR2;
```

Description

Checks if a statement is an existing triple in the specified model in the database.

Parameters

model_id
ID number of the semantic technology model. Must match a value in the MODEL_ID column of the MDSYS.SEM_MODEL$ view, which is described in Metadata for Models (page 1-3).

model_name
Name of the semantic technology model. Must match a value in the MODEL_NAME column of the MDSYS.SEM_MODEL$ view, which is described in Metadata for Models (page 1-3).

subject
Subject. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

property
Property. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

object
Object. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

Usage Notes

This function returns the string value FALSE, TRUE, or TRUE (EXACT):

- FALSE means that the statement is not a triple in the specified model the database.
- TRUE means that the statement matches the value of a triple or is the canonical representation of the value of a triple in the specified model the database.

- `TRUE (EXACT)` means that the specified `subject`, `property`, and `object` values have exact matches in a triple in the specified model in the database.

Examples

The following example checks if a statement is a triple in the database. In this case, there is an exact match. (This example is an excerpt from Example 1-89 (page 1-94) in Example: Family Information (page 1-95).)

```
SELECT SEM_APIS.IS_TRIPLE(
  'articles',
  'http://nature.example.com/Article2',
  'http://purl.org/dc/terms/references',
  'http://nature.example.com/Article3') AS is_triple FROM DUAL;

IS_TRIPLE
--------------------------------------------------------------------------------
TRUE (EXACT)
```

11.55 SEM_APIS.LOAD_INTO_STAGING_TABLE

Format

```
SEM_APIS.LOAD_INTO_STAGING_TABLE(
     stagong_table    IN VARCHAR2,
     source_table     IN VARCHAR2,
     input_format     IN VARCHAR2 DEFAULT NULL,
     parallel         IN INTEGER DEFAULT NULL,
     staging_table_owner IN VARCHAR2 DEFAULT NULL,
     source_table_owner  IN VARCHAR DEFAULT NULL,
     flags            IN VARCHAR DEFAULT NULL);
```

Description

Loads data into a staging table from an external table mapped to an N-Triple or N-Quad format input file.

Parameters

staging_table
Name of the staging table.

source_table
Name of the source external table.

input_format
Format of the input file mapped by the source external table: `N-TRIPLE` or `N-QUAD`

parallel
Degree of parallelism to use during the load.

staging_table_owner
Owner for the staging table being created. If not specified, the invoker is assumed to be the owner.

source_table_owner
Owner for the source table. If not specified, the invoker is assumed to be the owner.

flags
(Reserved for future use)

Usage Notes

For more information and an example, see Loading N-Quad Format Data into a Staging Table Using an External Table (page 1-76).

Examples

The following example loads the staging table. (This example is an excerpt from Example 1-82 (page 1-76) in Loading N-Quad Format Data into a Staging Table Using an External Table (page 1-76).)

```
BEGIN
  sem_apis.load_into_staging_table(
    staging_table => 'STAGE_TABLE'
   ,source_table  => 'stage_table_source'
   ,input_format  => 'N-QUAD');
END;
```

11.56 SEM_APIS.LOOKUP_ENTAILMENT

Format

```
SEM_APIS.LOOKUP_ENTAILMENT (
     models     IN SEM_MODELS,
     rulebases  IN SEM_RULEBASES
     ) RETURN VARCHAR2;
```

Description

Returns the name of the entailment (rules index) based on the specified models and rulebases.

Parameters

models
One or more model names. Its data type is SEM_MODELS, which has the following definition: `TABLE OF VARCHAR2(25)`

rulebases
One or more rulebase names. Its data type is SEM_RULEBASES, which has the following definition: `TABLE OF VARCHAR2(25)`Rules and rulebases are explained in Inferencing: Rules and Rulebases (page 1-8).

Usage Notes

For a rulebase index to be returned, it must be based on all specified models and rulebases.

Examples

The following example finds the entailment that is based on the `family` model and the `RDFS` and `family_rb` rulebases. (It is an excerpt from Example 1-90 (page 1-96) in Example: Family Information (page 1-95).)

```
SELECT SEM_APIS.LOOKUP_ENTAILMENT(SEM_MODELS('family'),
  SEM_RULEBASES('RDFS','family_rb')) AS lookup_entailment FROM DUAL;

LOOKUP_ENTAILMENT
--------------------------------------------------------------------------------
RDFS_RIX_FAMILY
```

11.57 SEM_APIS.MERGE_MODELS

Format

```
SEM_APIS.MERGE_MODELS(
     source_model         IN VARCHAR2,
     destination_model    IN VARCHAR2,
     rebuild_apptab_index IN BOOLEAN DEFAULT TRUE,
     drop_source_model    IN BOOLEAN DEFAULT FALSE,
     options              IN VARCHAR2 DEFAULT NULL);
```

Description

Inserts the content from a source model into a destination model, and updates the destination application table.

Parameters

source_model
Name of the source model.

destination_model
Name of the destination model.

rebuild_apptab_index
TRUE causes indexes on the destination application table to be rebuilt after the models are merged; FALSE does not rebuild any indexes.

drop_source_model
TRUE causes the source model (`source_model`) to be deleted after the models are merged; FALSE (the default) does not delete the source model.

options
A comma-delimited string of options that overrides the default behavior of the procedure. Currently, only the DOP (degree of parallelism) option is supported, to enable parallel execution of this procedure and to specify the degree of parallelism to be associated with the operation.

Usage Notes

Before you merge any models, if you are using positional parameters, check to be sure that you are specifying the correct models for the first and second parameters (source model for the first, destination model for the second). This is especially important if you plan to specify `drop_source_model=TRUE`.

If appropriate, make copies of the destination model or both models before performing the merge. To make a copy of a model, use SEM_APIS.CREATE_SEM_MODEL (page 11-27) to create an empty model with the desired name for the copy, and use SEM_APIS.MERGE_MODELS to populate the newly created copy as the destination model.

Some common uses for this procedure include the following:

- If you have read-only access to a model that you want to modify, you can clone that model into an empty model on which you have full access, and then modify this latter model.
- If you want to consolidate multiple models, you can use this procedure as often as necessary to merge the necessary models. Merging all models beforehand and using only the merged model simplifies entailment and can improve entailment performance.

On a multi-core or multi-cpu machine, the `DOP` (degree of parallelism) option can be beneficial. See Examples for an example that uses the `DOP` option.

If the source model is large, you may want to update the optimizer statistics on the destination after the merge operation by calling the SEM_APIS.ANALYZE_MODEL (page 11-9) procedure.

The following considerations apply to the use of this procedure:

- You must be the owner of the destination model and have SELECT privilege on the source model. If `drop_second_model=TRUE`, you must also be owner of the source model.
- This procedure cannot be used on VPD-enabled or version-enabled semantic models.
- This procedure is not supported on virtual models (explained in Virtual Models (page 1-13)).
- No table constraints are allowed on the destination application table.

Examples

The following example inserts the contents of model `M1` into `M2`.

```
EXECUTE SEM_APIS.MERGE_MODELS('M1', 'M2');
```

The following example inserts the contents of model `M1` into `M2`, and it specifies a degree of parallelism of 4 (up to four parallel threads for execution of the merge operation).

```
EXECUTE SEM_APIS.MERGE_MODELS('M1', 'M2', null, null, 'DOP=4');
```

11.58 SEM_APIS.PRIVILEGE_ON_APP_TABLES

Format

SEM_APIS.PRIVILEGE_ON_APP_TABLES(

command IN VARCHAR2 DEFAULT 'GRANT',
privilege IN VARCHAR2 DEFAULT 'SELECT');

Description

Grants (or revokes) SELECT or INSERT privilege to (or from) MDSYS on application tables corresponding to all the RDF models owned by the invoker.

Parameters

command
SQL statement, with possible values `GRANT` (the default) or `REVOKE` (case insensitive).

privilege
Privilege name, with possible values `SELECT` (the default) or `INSERT` (case insensitive).

Usage Notes

(None)

Examples

The following example grants SELECT privilege to MDSYS on application tables corresponding to all the RDF models owned by the invoker.

```
EXECUTE SEM_APIS.PRIVILEGE_ON_APP_TABLES('grant', 'select');
```

11.59 SEM_APIS.PURGE_UNUSED_VALUES

Format

```
SEM_APIS.PURGE_UNUSED_VALUES(
     flags  IN VARCHAR2 DEFAULT NULL);
```

Description

Purges purges invalid geometry literal values from the semantic network.

Parameters

flags
An optional quoted string with one or more of the following keyword specifications:

- `MBV_METHOD=SHADOW` allows the use of a different value loading strategy that may lead to faster processing when a large number of values need to be purged.
- `PARALLEL=<integer>` allows much of the processing to be done in parallel using the specified `integer` degree of parallelism to be associated with the operation. If only `PARALLEL` is specified without a degree, a default degree will be used.
- `PUV_COMPUTE_VIDS_USED` allows use of a different strategy that may lead to faster processing when most of the values are expected to be purged.

Usage Notes

Before calling this procedure, you must grant to MDSYS the SELECT privilege on application tables for all the currently existing RDF models.

For more usage information and an extended example, see Purging Unused Values (page 1-83).

Examples

The following example purges unused values using a degree of parallelism of 4.

```
EXECUTE SEM_APIS.PURGE_UNUSED_VALUES(flags => 'PARALLEL=4');
```

11.60 SEM_APIS.REFRESH_SEM_NETWORK_INDEX_INFO

Format

```
SEM_APIS.REFRESH_SEM_NETWORK_INDEX_INFO(
     options  IN VARCHAR2 DEFAULT NULL);
```

Description

Refreshes the information about semantic network indexes.

Parameters

options
(Reserved for future use)

Usage Notes

This procedure updates the information in the MDSYS.SEM_NETWORK_INDEX_INFO view, which is described in MDSYS.SEM_NETWORK_INDEX_INFO View (page 1-87).

Examples

The following example refreshes the information about semantic network indexes.

```
EXECUTE sem_apis.refresh_sem_network_index_info;
```

11.61 SEM_APIS.REMOVE_DUPLICATES

Format

```
SEM_APIS.REMOVE_DUPLICATES(
     model_name           IN VARCHAR2,
     threshold            IN FLOAT DEFAULT 0.3,
     rebuild_apptab_index IN BOOLEAN DEFAULT TRUE);
```

Description

Removes duplicate triples from a model.

Parameters

model_name
Name of the model.

threshold
A value to determine how numerous triples must be in order for the removal operation to be performed. This procedure removes triples only if the number of triples in the model exceeds the following formula: (total-triples - total-unique-triples + 0.01) / (total-unique-triples + 0.01). For the default value of 0.3 and a model containing 1000 total triples (including duplicates), duplicate triples would be removed only if the number of duplicates exceeds approximately 230.

The lower the threshold value, the fewer duplicates are needed for the procedure to remove duplicates; the higher the threshold value, the more duplicates are needed for the procedure to remove duplicates.

rebuild_apptab_index
TRUE (the default) causes all usable indexes on tables that were affected by this operation to be rebuilt after the duplicate triples are removed; FALSE does not rebuild any indexes.

Usage Notes

When duplicate triples are removed, all information in the removed rows is lost, including information in columns other than the triple column.

This procedure is not supported on virtual models (explained in Virtual Models (page 1-13)).

This procedure is not supported on version-enabled models (explained in Workspace Manager Support for RDF Data (page 6-1)).

If the model is empty, or if it contains no duplicate triples or not enough duplicate triples (as computed using the `threshold` value), this procedure does not perform any removal operations.

If there are not enough duplicates (as computed using the `threshold` value) to perform the operation, an informational message is displayed.

If unusable indexes are involved, be sure that the SKIP_UNUSABLE_INDEXES system parameter is set to TRUE. Although TRUE is the default value for this parameter, some production databases may use the value FALSE; therefore, if you need to change it, enter the following:

```
SQL> alter session set skip_unusable_indexes=true;
```

To use this procedure on an application table with one or more user-defined triggers, you must connect as a DBA user and grant the ALTER ANY TRIGGER privilege to the MDSYS user, as follows:

```
SQL> grant alter any trigger to MDSYS;
```

Examples

The following example removes duplicate triples in the model named `family`. It accepts the default threshold value of 0.3 and (by default) rebuilds indexes after the duplicates are removed.

```
EXECUTE SEM_APIS.REMOVE_DUPLICATES('family');
```

11.62 SEM_APIS.RENAME_ENTAILMENT

Format

```
SEM_APIS.RENAME_ENTAILMENT(
     old_name  IN VARCHAR2,
     new_name  IN VARCHAR2);
```

Description

Renames an entailment (rules index).

Parameters

old_name
Name of the existing entailment to be renamed.

new_name
New name for the entailment.

Usage Notes

This procedure is not supported on version-enabled RDF models, as explained in Special Considerations When Using Workspace Manager Support for RDF Data (page 6-5).

Examples

The following example renames a entailment named `OWLTST_IDX` to `MY_OWLTST_IDX`.

```
EXECUTE sem_apis.rename_entailment('owltst_idx', 'my_owltst_idx');
```

11.63 SEM_APIS.RENAME_MODEL

Format

```
SEM_APIS.RENAME_MODEL(
     old_name  IN VARCHAR2,
     new_name  IN VARCHAR2);
```

Description

Renames a model.

Parameters

old_name
Name of the existing model to be renamed.

new_name
New name for the model.

Usage Notes

The following considerations apply to the use of this procedure:

- You must be the owner of the existing model.
- This procedure cannot be used on VPD-enabled or version-enabled semantic models.
- This procedure is not supported on virtual models (explained in Virtual Models (page 1-13)).

Contrast this procedure with SEM_APIS.SWAP_NAMES (page 11-69), which swaps (exchanges) the names of two existing models.

Examples

The following example renames a model named MODEL1 to MODEL2.

```
EXECUTE sem_apis.rename_model('model1', 'model2');
```

11.64 SEM_APIS.SET_ENTAILMENT_STATS

Format

```
SEM_APIS.SET_ENTAILMENT_STATS (
     entailment_name IN VARCHAR2,
     numrows         IN NUMBER DEFAULT NULL,
     numblks         IN NUMBER DEFAULT NULL,
     avgrlen         IN NUMBER DEFAULT NULL,
     flags           IN NUMBER DEFAULT NULL,
     no_invalidate   IN BOOLEAN DEFAULT DBMS_STATS.AUTO_INVALIDATE,
     cachedblk       IN NUMBER DEFAULT NULL,
     cachehit        IN NUMBER DEFAULT NULL,
     force           IN BOOLEAN DEFAULT FALSE);
```

Description

Sets statistics for a specified entailment.

Parameters

entailment_name
Name of the entailment.

(other parameters)
See the parameter explanations for the DBMS_STATS.SET_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference,* although `force` here applies to entailment statistics.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example sets statistics for an entailment named OWLTST_IDX.

```
EXECUTE SEM_APIS.SET_ENTAILMENT_STATS('owltst_idx', numrows => 100);
```

11.65 SEM_APIS.SET_MODEL_STATS

Format

```
SEM_APIS.SET_MODEL_STATS (
     model_name  IN VARCHAR2,
     numrows       IN NUMBER DEFAULT NULL,
     numblks       IN NUMBER DEFAULT NULL,
     avgrlen       IN NUMBER DEFAULT NULL,
```

```
    flags         IN NUMBER DEFAULT NULL,
    no_invalidate IN BOOLEAN DEFAULT DBMS_STATS.AUTO_INVALIDATE,
    cachedblk     IN NUMBER DEFAULT NULL,
    cachehit      IN NUMBER DEFAULT NULL,
    force         IN BOOLEAN DEFAULT FALSE);
```

Description

Sets statistics for a specified model.

Parameters

model_name
Name of the model.

(other parameters)
See the parameter explanations for the DBMS_STATS.DELETE_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference*, although `force` here applies to model statistics.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example sets statistics for a model named `FAMILY`.

```
EXECUTE SEM_APIS.SET_MODEL_STATS('family', numrows => 100);
```

11.66 SEM_APIS.SWAP_NAMES

Format

```
SEM_APIS.SWAP_NAMES(
    model1  IN VARCHAR2,
    model2  IN VARCHAR2);
```

Description

Swaps (exchanges) the names of two existing models.

Parameters

model1
Name of a model.

model2
Name of another model.

Usage Notes

As a result of this procedure, the name of model `model1` is changed to the (old) name of `model2`, and the name of model `model2` is changed to the (old) name of `model1`.

The order of the names does not affect the result. For example, you could specify `TEST` for `model1` and `PRODUCTION` for `model2`, or `PRODUCTION` for `model1` and `TEST` for `model2`, and the result will be the same.

This procedure is not supported on version-enabled RDF models, as explained in Special Considerations When Using Workspace Manager Support for RDF Data (page 6-5).

Contrast this procedure with SEM_APIS.RENAME_MODEL (page 11-67), which renames an existing model.

Examples

The following example changes the name of the (old) TEST model to PRODUCTION, and the name of the (old) PRODUCTION model to TEST.

```
EXECUTE sem_apis.swap_names('test', 'production');
```

11.67 SEM_APIS.UNESCAPE_CLOB_TERM

Format

```
SEM_APIS.UNESCAPE_CLOB_TERM(
     term  IN CLOB CHARACTER SET ANY_CS
     ) RETURN CLOB CHARACTER SET val%CHARSET;
```

Description

Returns the input RDF term with special characters and non-ASCII characters unescaped as specified by the W3C N-Triples format (`http://www.w3.org/TR/rdf-testcases/#ntriples`).

Parameters

term
The RDF term to unescape.

Usage Notes

For information about using the `DO_UNESCAPE` keyword in the `options` parameter of the SEM_MATCH table function, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).

Examples

The following example unescapes an input RDF term containing TAB and NEWLINE characters.

```
SEM_APIS.UNESCAPE_CLOB_TERM('"abc\tdef\nhij"^^<http://www.w3.org/2001/
XMLSchema#string>')
  FROM DUAL;
```

11.68 SEM_APIS.UNESCAPE_CLOB_VALUE

Format

```
SEM_APIS.UNESCAPE_CLOB_VALUE(
     val           IN  CLOB CHARACTER SET ANY_CS,
     start_offset  IN NUMBER DEFAULT 1,
     end_offset    IN NUMBER DEFAULT 0,
     include_start IN NUMBER DEFAULT 0
     ) RETURN VARCHAR2 CHARACTER SET val%CHARSET;
```

Description

Returns the input CLOB value with special characters and non-ASCII characters unescaped as specified by the W3C N-Triples format (`http://www.w3.org/TR/rdf-testcases/#ntriples`).

Parameters

val
The CLOB text to unescape.

start_offset
The offset in `val` from which to start character unescaping. The default (1) causes escaping to start at the first character of `val`.

end_offset
The offset in `val` from which to end character unescaping. The default (0) causes escaping to continue through the end of `val`.

include_start
Set to 1 if the characters in val from 1 to `start_offset` should be prefixed (prepended) to the return value. Otherwise, no such characters will be prefixed to the return value.

Usage Notes

For information about using the `DO_UNESCAPE` keyword in the `options` parameter of the SEM_MATCH table function, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).

Examples

The following example unescapes an input character string containing TAB and NEWLINE characters.

```
SELECT SEM_APIS.UNESCAPE_CLOB_VALUE('abc\tdef\nhij')
  FROM DUAL;
```

11.69 SEM_APIS.UNESCAPE_RDF_TERM

Format

```
SEM_APIS.UNESCAPE_RDF_TERM(
     term  IN VARCHAR2 CHARACTER SET ANY_CS
     ) RETURN VARCHAR2 CHARACTER SET val%CHARSET;
```

Description

Returns the input RDF term with special characters and non-ASCII characters unescaped as specified by the W3C N-Triples format (http://www.w3.org/TR/rdf-testcases/#ntriples).

Parameters

term
The RDF term to unescape.

Usage Notes

For information about using the `DO_UNESCAPE` keyword in the `options` parameter of the SEM_MATCH table function, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).

Examples

The following example unescapes an input RDF term containing TAB and NEWLINE characters.

```
SELECT SEM_APIS.UNESCAPE_RDF_TERM('"abc\tdef\nhij"^^<http://www.w3.org/2001/
XMLSchema#string>')
  FROM DUAL;
```

11.70 SEM_APIS.UNESCAPE_RDF_VALUE

Format

```
SEM_APIS.UNESCAPE_RDF_VALUE(
     val  IN VARCHAR2 CHARACTER SET ANY_CS
     ) RETURN VARCHAR2 CHARACTER SET val%CHARSET;
```

Description

Returns the input CLOB value with special characters and non-ASCII characters unescaped as specified by the W3C N-Triples format (http://www.w3.org/TR/rdf-testcases/#ntriples).

Parameters

val
The text to unescape.

utf_encode
Set to 1 (the default) if non-ASCII characters and non-printable ASCII characters other than chr(8), chr(9), chr(10), chr(12), and chr(13) should be escaped. Otherwise, such characters will not be escaped.

Usage Notes

For information about using the `DO_UNESCAPE` keyword in the `options` parameter of the SEM_MATCH table function, see Using the SEM_MATCH Table Function to Query Semantic Data (page 1-22).

Examples

The following example unescapes an input character string containing TAB and NEWLINE characters.

```
SELECT SEM_APIS.UNESCAPE_RDF_VALUE('abc\tdef\nhij')
  FROM DUAL;
```

11.71 SEM_APIS.VALIDATE_ENTAILMENT

Format

```
SEM_APIS.VALIDATE_ENTAILMENT(
     models_in    IN SEM_MODELS,
     rulebases_in IN SEM_RULEBASES,
     criteria_in  IN VARCHAR2 DEFAULT NULL,
     max_conflict IN NUMBER DEFAULT 100,
     options      IN VARCHAR2 DEFAULT NULL
     ) RETURN RDF_LONGVARCHARARRAY;
```

Description

Validates entailments (rules indexes) that can be used to perform OWL or RDFS inferencing for one or more models.

Parameters

models_in
One or more model names. Its data type is SEM_MODELS, which has the following definition: `TABLE OF VARCHAR2(25)`

rulebases_in
One or more rulebase names. Its data type is SEM_RULEBASES, which has the following definition: `TABLE OF VARCHAR2(25)`. Rules and rulebases are explained in Inferencing: Rules and Rulebases (page 1-8).

criteria_in
A comma-delimited string of validation checks to run. If you do not specify this parameter, by default all of the following checks are run:

- `UNSAT`: Find unsatisfiable classes.
- `EMPTY`: Find instances that belong to unsatisfiable classes.
- `SYNTAX_S`: Find triples whose subject is neither URI nor blank node.

- SYNTAX_P: Find triples whose predicate is not URI.
- SELF_DIF: Find individuals that are different from themselves.
- INST: Find individuals that simultaneously belong to two disjoint classes.
- SAM_DIF: Find pairs of individuals that are same (owl:sameAs) and different (owl:differentFrom) at the same time.

To specify fewer checks, specify a string with only the checks to be performed. For example, `criteria_in => 'UNSAT'` causes the validation process to search only for unsatisfiable classes.

max_conflict
The maximum number of conflicts to find before the validation process stops. The default value is 100.

options
(Not currently used. Reserved for Oracle use.).

Usage Notes

This procedure can be used to detect inconsistencies in the original entailment. For more information, see Validating OWL Models and Entailments (page 2-9).

This procedure returns a null value if no errors are detected or (if errors are detected) an object of type RDF_LONGVARCHARARRAY, which has the following definition: `VARRAY(32767) OF VARCHAR2(4000)`

To create an entailment, use the SEM_APIS.CREATE_ENTAILMENT (page 11-15) procedure.

Examples

For an example of this procedure, see Example 2-5 (page 2-9) in Validating OWL Models and Entailments (page 2-9).

11.72 SEM_APIS.VALIDATE_MODEL

Format

```
SEM_APIS.VALIDATE_MODEL(
     models_in    IN SEM_MODELS,
     criteria_in  IN VARCHAR2 DEFAULT NULL,
     max_conflict IN NUMBER DEFAULT 100,
     options      IN VARCHAR2 DEFAULT NULL
     ) RETURN RDF_LONGVARCHARARRAY;
```

Description

Validates one or more models.

Parameters

models_in
One or more model names. Its data type is SEM_MODELS, which has the following definition: `TABLE OF VARCHAR2(25)`

criteria_in
A comma-delimited string of validation checks to run. If you do not specify this parameter, by default all of the following checks are run:

- `UNSAT`: Find unsatisfiable classes.
- `EMPTY`: Find instances that belong to unsatisfiable classes.
- `SYNTAX_S`: Find triples whose subject is neither URI nor blank node.
- `SYNTAX_P`: Find triples whose predicate is not URI.
- `SELF_DIF`: Find individuals that are different from themselves.
- `INST`: Find individuals that simultaneously belong to two disjoint classes.
- `SAM_DIF`: Find pairs of individuals that are same (owl:sameAs) and different (owl:differentFrom) at the same time.

To specify fewer checks, specify a string with only the checks to be performed. For example, `criteria_in => 'UNSAT'` causes the validation process to search only for unsatisfiable classes.

max_conflict
The maximum number of conflicts to find before the validation process stops. The default value is 100.

options
(Not currently used. Reserved for Oracle use.).

Usage Notes

This procedure can be used to detect inconsistencies in the original data model. For more information, see Validating OWL Models and Entailments (page 2-9).

This procedure returns a null value if no errors are detected or (if errors are detected) an object of type RDF_LONGVARCHARARRAY, which has the following definition: `VARRAY(32767) OF VARCHAR2(4000)`

Examples

The following example validates the model named `family`.

```
SELECT SEM_APIS.VALIDATE_MODEL(SEM_MODELS('family')) FROM DUAL;
```

11.73 SEM_APIS.VALUE_NAME_PREFIX

Format

```
SEM_APIS.VALUE_NAME_PREFIX (
     value_name  IN VARCHAR2,
     value_type  IN VARCHAR2
     ) RETURN VARCHAR2;
```

Description

Returns the value in the VNAME_PREFIX column for the specified value name and value type pair in the MDSYS.RDF_VALUE$ table.

Parameters

value_name
Value name. Must match a value in the VALUE_NAME column in the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

value_type
Value type. Must match a value in the VALUE_TYPE column in the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

Usage Notes

This function usually causes an index on the MDSYS.RDF_VALUE$ table to be used for processing a lookup for values, and thus can make a query run faster.

Examples

The following query returns value name portions of all the lexical values in MDSYS.RDF_VALUE$ table with a prefix value same as that returned by the VALUE_NAME_PREFIX function. This query uses an index on the MDSYS.RDF_VALUE$ table, thereby providing efficient lookup.

```
SELECT value_name FROM MDSYS.RDF_VALUE$
  WHERE vname_prefix = SEM_APIS.VALUE_NAME_PREFIX(
    'http://www.w3.org/1999/02/22-rdf-syntax-ns#type','UR');

VALUE_NAME
--------------------------------------------------------------------------------
http://www.w3.org/1999/02/22-rdf-syntax-ns#Alt
http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag
http://www.w3.org/1999/02/22-rdf-syntax-ns#List
http://www.w3.org/1999/02/22-rdf-syntax-ns#Property
http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq
http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement
http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral
http://www.w3.org/1999/02/22-rdf-syntax-ns#first
http://www.w3.org/1999/02/22-rdf-syntax-ns#nil
http://www.w3.org/1999/02/22-rdf-syntax-ns#object
http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate
http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
http://www.w3.org/1999/02/22-rdf-syntax-ns#subject
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/1999/02/22-rdf-syntax-ns#value

15 rows selected.
```

11.74 SEM_APIS.VALUE_NAME_SUFFIX

Format

```
SEM_APIS.VALUE_NAME_SUFFIX (
     value_name  IN VARCHAR2,
     value_type  IN VARCHAR2
     ) RETURN VARCHAR2;
```

Description

Returns the value in the VNAME_SUFFIX column for the specified value name and value type pair in the MDSYS.RDF_VALUE$ table.

Parameters

value_name
Value name. Must match a value in the VALUE_NAME column in the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

value_type
Value type. Must match a value in the VALUE_TYPE column in the MDSYS.RDF_VALUE$ table, which is described in Statements (page 1-5).

Usage Notes

This function usually causes an index on the MDSYS.RDF_VALUE$ table to be used for processing a lookup for values, and thus can make a query run faster.

Examples

The following query returns value name portions of all the lexical values in MDSYS.RDF_VALUE$ table with a suffix value same as that returned by the VALUE_NAME_SUFFIX function. This query uses an index on the MDSYS.RDF_VALUE$ table, thereby providing efficient lookup.

```
SELECT value_name FROM MDSYS.RDF_VALUE$
  WHERE vname_suffix = SEM_APIS.VALUE_NAME_SUFFIX(
    'http://www.w3.org/1999/02/22-rdf-syntax-ns#type','UR');

VALUE_NAME
--------------------------------------------------------------------------------
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
```

12

SEM_OLS Package Subprograms

The SEM_OLS package contains subprograms (functions and procedures) related to triple-level security to RDF data, using Oracle Label Security (OLS). To use the subprograms in this chapter, you should understand the conceptual and usage information in RDF Semantic Graph Overview (page 1-1) and Fine-Grained Access Control for RDF Data (page 5-1).

This chapter provides reference information about the subprograms, listed in alphabetical order.

12.1 SEM_OLS.APPLY_POLICY_TO_APP_TAB

Format

```
SEM_OLS.APPLY_POLICY_TO_APP_TAB(
     policy_name  IN VARCHAR2,
     schema_name  IN VARCHAR2,
     table_name   IN VARCHAR2,
     predicate    IN VARCHAR2 DEFAULT NULL);
```

Description

Applies an OLS policy to an application table.

Parameters

policy_name
Name of an existing OLS policy.

schema_name
Name of the schema containing the application table.

table_name
Name of the application table.

predicate
An additional predicate to combine with the label-based predicate.

Usage Notes

When you use triple-level security, OLS is applied to each semantic model in the network. That is, label security is applied to the relevant internal tables and to all the application tables; there is no need to manually apply policies to the application tables of existing semantic models. However, if you need to create additional models after applying the OLS policy, you must use the SEM_OLS.APPLY_POLICY_TO_APP_TAB procedure to apply OLS to the application table before creating the model.

You must have the following to execute this procedure: EXECUTE privilege for the SA_POLICY_ADMIN package, and the policy_DBA role.

Before executing this procedure, you must have executed the SEM_RDFSA.APPLY_OLS_POLICY (page 15-3) procedure specifying SEM_RDFSA.TRIPLE_LEVEL_ONLY for the rdfsa_options parameter.

To remove the OLS policy from the application table, use the SEM_OLS.REMOVE_POLICY_FROM_APP_TAB (page 12-2) procedure.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example applies an OLS policy named defense to the MY_SCHEMA.MY_APP_TABLE application table.

```
begin
  sem_ols.apply_policy_to_app_table(
        policy_name => 'defense',
        schema_name => 'my_schema',
        table_name  => 'my_app_table');
end;
/
```

12.2 SEM_OLS.REMOVE_POLICY_FROM_APP_TAB

Format

```
SEM_OLS.REMOVE_POLICY_FROM_APP_TAB(
     policy_name  IN VARCHAR2,
     schema_name  IN VARCHAR2,
     table_name   IN VARCHAR2);
```

Description

Permanently removes or detaches the OLS policy from an application table.

Parameters

policy_name
Name of the existing OLS policy.

schema_name
Name of the schema containing the application table.

table_name
Name of the application table.

Usage Notes

If you have dropped a semantic model and you no longer need to protect the application table, you can use this procedure.

You must have the following to execute this procedure: EXECUTE privilege for the SA_POLICY_ADMIN package, and the policy_DBA role.

Before executing this procedure, you must have executed the SEM_RDFSA.APPLY_OLS_POLICY (page 15-3) procedure specifying `SEM_RDFSA.TRIPLE_LEVEL_ONLY` for the `rdfsa_options` parameter.

An exception is generated if the associated model exists. In this case, if you want to execute this procedure, you must first drop the model.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example removes the OLS policy named `defense` from the MY_SCHEMA.MY_APP_TABLE application table.

```
begin
  sem_ols.remove_policy_from_app_table(
        policy_name => 'defense',
        schema_name => 'my_schema',
        table_name  => 'my_app_table');
end;
/
```

13

SEM_PERF Package Subprograms

The SEM_PERF package contains subprograms for examining and enhancing the performance of the Resource Description Framework (RDF) and Web Ontology Language (OWL) support in an Oracle database. To use the subprograms in this chapter, you must understand the conceptual and usage information in RDF Semantic Graph Overview (page 1-1) and OWL Concepts (page 2-1).

This chapter provides reference information about the subprograms, listed in alphabetical order.

13.1 SEM_PERF.DELETE_NETWORK_STATS

Format

```
SEM_PERF.DELETE_NETWORK_STATS (
     cascade_parts    IN BOOLEAN DEFAULT TRUE,
     cascade_columns  IN BOOLEAN DEFAULT TRUE,
     cascade_indexes  IN BOOLEAN DEFAULT TRUE,
     no_invalidate    IN BOOLEAN DEFAULT DBMS_STATS.AUTO_INVALIDATE,
     force            IN BOOLEAN DEFAULT FALSE,
     options          IN VARCHAR2 DEFAULT NULL);
```

Description

Deletes statistics for the semantic network.

Parameters

options
Controls the scope of the operation:

- If `MDSYS.SDO_RDF.VALUE_TABLE_ONLY`, the operation applies only to the MDSYS.RDF_VALUE$ table.
- If `MDSYS.SDO_RDF.LINK_TABLE_ONLY`, the operation applies only to the MDSYS.RDF_LINK$ table.
- If null (the default), the operation applies to both the MDSYS.RDF_VALUE$ and MDSYS.RDF_LINK$ tables.

(other parameters)
See the parameter explanations for the DBMS_STATS.DELETE_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference*, although `force` here applies to network statistics.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example deletes statistics for the semantic network:

```
EXECUTE SEM_APIS.DELETE_NETWORK_STATS;
```

13.2 SEM_PERF.EXPORT_NETWORK_STATS

Format

```
SEM_PERF.EXPORT_NETWORK_STATS (
     stattab        IN VARCHAR2,
     statid         IN VARCHAR2 DEFAULT NULL,
     cascade        IN BOOLEAN DEFAULT TRUE,
     statown        IN VARCHAR2 DEFAULT NULL,
     stat_category  IN VARCHAR2 DEFAULT 'OBJECT_STATS',
     options        IN VARCHAR2 DEFAULT NULL);
```

Description

Exports the statistics for the semantic network and stores them in the user statistics table.

Parameters

options
Controls the scope of the operation:

- If `MDSYS.SDO_RDF.VALUE_TABLE_ONLY`, the operation applies only to the MDSYS.RDF_VALUE$ table.
- If `MDSYS.SDO_RDF.LINK_TABLE_ONLY`, the operation applies only to the MDSYS.RDF_LINK$ table.
- If null (the default), the operation applies to both the MDSYS.RDF_VALUE$ and MDSYS.RDF_LINK$ tables.

(other parameters)
See the parameter explanations for the DBMS_STATS.EXPORT_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference*.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example exports the statistics for the semantic network and stores them in a table named STAT_TABLE.

```
EXECUTE SEM_APIS.EXPORT_NETWORK_STATS('stat_table');
```

13.3 SEM_PERF.GATHER_STATS

Format

```
SEM_PERF.GATHER_STATS(
     just_on_values_table  IN BOOLEAN DEFAULT FALSE,
     degree                IN NUMBER(38) DEFAULT NULL);
```

Description

Gathers statistics about RDF and OWL tables and their indexes.

Parameters

just_on_values_table
TRUE collects statistics only on the table containing the lexical values of triples; FALSE (the default) collects statistics on all major tables related to the storage of RDF and OWL data.

A value of TRUE reduces the execution time for the procedure; and it may be sufficient if you need only to collect statistics on the values table (for example, if you use other interfaces to collect any other statistics that you might need).

degree
Degree of parallelism. For more information about parallel execution, see *Oracle Database VLDB and Partitioning Guide*.

Usage Notes

To use this procedure, you must connect as a user with permission to execute it. By default, when Spatial and Graph is installed as part of Oracle Database, only the MDSYS user can execute this procedure; however execution permission on this procedure can be granted to users as needed.

This procedure collects statistical information that can help you to improve inferencing performance, as explained in Enhancing Inference Performance (page 2-10). This procedure internally calls the DBMS_STATS.GATHER_TABLE_STATS procedure to collect statistics on RDF- and OWL-related tables and their indexes, and stores the statistics in the Oracle Database data dictionary. For information about using the DBMS_STATS package, see *Oracle Database PL/SQL Packages and Types Reference*.

Gathering statistics uses significant system resources, so execute this procedure when it cannot adversely affect essential applications and operations.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example gathers statistics about RDF and OWL related tables and their indexes.

```
EXECUTE SEM_PERF.GATHER_STATS;
```

13.4 SEM_PERF.IMPORT_NETWORK_STATS

Format

```
SEM_PERF.IMPORT_NETWORK_STATS (
     stattab       IN VARCHAR2,
     statid        IN VARCHAR2 DEFAULT NULL,
     cascade       IN BOOLEAN DEFAULT TRUE,
     statown       IN VARCHAR2 DEFAULT NULL,
     no_invalidate IN BOOLEAN DEFAULT FALSE,
     force         IN BOOLEAN DEFAULT FALSE,
     stat_category IN VARCHAR2 DEFAULT 'OBJECT_STATS',
     options       IN VARCHAR2 DEFAULT NULL);
```

Description

Retrieves the statistics for the semantic network from a user statistics table and stores them in the dictionary.

Parameters

options
Controls the scope of the operation:

- If `MDSYS.SDO_RDF.VALUE_TABLE_ONLY`, the operation applies only to the MDSYS.RDF_VALUE$ table.
- If `MDSYS.SDO_RDF.LINK_TABLE_ONLY`, the operation applies only to the MDSYS.RDF_LINK$ table.
- If null (the default), the operation applies to both the MDSYS.RDF_VALUE$ and MDSYS.RDF_LINK$ tables.

(other parameters)
See the parameter explanations for the DBMS_STATS.IMPORT_TABLE_STATS procedure in *Oracle Database PL/SQL Packages and Types Reference*, although `force` here applies to network statistics.

Usage Notes

See the information about the DBMS_STATS package in*Oracle Database PL/SQL Packages and Types Reference*.

See also Managing Statistics for Semantic Models and the Semantic Network (page 1-89).

Examples

The following example imports the statistics for the semantic network in a table named `STAT_TABLE`, and stores them in the dictionary.

```
EXECUTE SEM_APIS.IMPORT_NETWORK_STATS('stat_table');
```

14

SEM_RDFCTX Package Subprograms

The SEM_RDFCTX package contains subprograms (functions and procedures) to manage extractor policies and semantic indexes created for documents. To use the subprograms in this chapter, you should understand the conceptual and usage information in Semantic Indexing for Documents (page 4-1).

This chapter provides reference information about the subprograms, listed in alphabetical order.

14.1 SEM_RDFCTX.ADD_DEPENDENT_POLICY

Format

```
SEM_RDFCTX.ADD_DEPENDENT_POLICY(
     index_name     IN VARCHAR2,
     policy_name    IN VARCHAR2,
     partition_name IN VARCHAR2 DEFAULT NULL);
```

Description

Adds a dependent policy to an (already created) index or index partition.

Parameters

index_name
Name of the index.

policy_name
Name of the dependent policy.

partition_name
If the specified index is local, the name of the target partition. (Otherwise, must be null.)

Usage Notes

The base policy corresponding to the new dependent policy must already be a part of the index.

Examples

The following example adds a new dependent policy `SEM_EXTR_PLUS_GEOONT` to the index `ArticleIndex`.

```
begin
  sem_rdfctx.add_dependent_policy (index_name  => 'ArticleIndex',
                                   policy_name => 'SEM_EXTR_PLUS_GEOONT');
```

```
end;
/
```

14.2 SEM_RDFCTX.CREATE_POLICY

Format

```
SEM_RDFCTX.CREATE_POLICY(
     policy_name  IN VARCHAR2,
     extractor    IN mdsys.rdfctx_extractor,
     preferences  IN sys.XMLType DEFAULT NULL);
```

or

```
SEM_RDFCTX.CREATE_POLICY(
     policy_name       IN VARCHAR2,
     base_policy       IN VARCHAR2,
     user_models       IN SEM_MODELS DEFAULT NULL,
     user_entailments  IN SEM_MODELS DEFAULT NULL);
```

Description

Creates an extractor policy. (The first format is for a base policy; the second format is for a policy that is dependent on a base policy.)

Parameters

policy_name
Name of the extractor policy.

extractor
An instance of a subtype of the RDFCTX_EXTRACTOR type that encapsulates the extraction logic for the information extractor.

preferences
Any preferences associated with the policy.

base_policy
Base extractor policy for a dependent policy.

user_models
List of user models for a dependent policy.

user_entailments
List of user entailments for a dependent policy.

Usage Notes

An extractor policy created using this procedure determines the characteristics of a semantic index that is created using the policy. Each extractor policy refers to an instance of an extractor type, either directly or indirectly. An extractor policy with a direct reference to an extractor type instance can be used to compose other extractor policies that include additional RDF models for ontologies.

An instance of the extractor type assigned to the extractor parameter must be an instance of a direct or indirect subtype of type `mdsys.rdfctx_extractor`.

The RDF models specified in the `user_models` parameter must be accessible to the user that is creating the policy.

The RDF entailments specified in the `user_entailments` parameter must be accessible to the user that is creating the policy. Note that the RDF models underlying the entailments do not get automatically included in the dependent policy. To include one or more of those underlying RDF models, you need to include the models in the `user_models` parameter.

The preferences specified for extractor policy determine the type of repository used for the documents to be indexed and other relevant information. For more information, see Indexing External Documents (page 4-10).

Examples

The following example creates an extractor policy using the gatenlp_extractor extractor type, which is included with the Oracle Database support for semantic indexing.

```
begin
  sem_rdfctx.create_policy (policy_name => 'SEM_EXTR',
                            extractor   => mdsys.gatenlp_extractor());
end;
/
```

The following example creates a dependent policy for the previously created extractor policy, and it adds the user-defined RDF model `geo_ontology` to the dependent policy.

```
begin
  sem_rdfctx.create_policy (policy_name => 'SEM_EXTR_PLUS_GEOONT',
                            base_policy => 'SEM_EXTR',
                            user_models => SEM_MODELS ('geo_ontology'));
end;
/
```

14.3 SEM_RDFCTX.DROP_POLICY

Format

```
SEM_RDFCTX.DROP_POLICY(
     policy_name  IN VARCHAR2);
```

Description

Deletes (drops) an unused extractor policy.

Parameters

policy_name
Name of the extractor policy.

Usage Notes

An exception is generated if the specified policy being is used for a semantic index for documents or if a dependent extractor policy exists for the specified policy.

Examples

The following example drops the SEM_EXTR_PLUS_GEOONT extractor policy.

```
begin
  sem_rdfctx.drop_policy (policy_name => 'SSEM_EXTR_PLUS_GEOONT');
end;
/
```

14.4 SEM_RDFCTX.MAINTAIN_TRIPLES

Format

```
SEM_RDFCTX.MAINTAIN_TRIPLES(
     index_name     IN VARCHAR2,
     where_clause   IN VARCHAR2,
     rdfxml_content IN sys.XMLType,
     policy_name    IN VARCHAR2 DEFAULT NULL,
     action         IN VARCHAR2 DEFAULT 'ADD');
```

Description

Adds one or more triples to graphs that contain information extracted from specific documents.

Parameters

index_name
Name of the semantic index for documents.

where_clause
A SQL predicate (WHERE clause text without the WHERE keyword) on the table in which the documents are stored, to identify the rows for which to maintain the index.

rdfxml_content
Triples, in the form of an RDF/XML document, to be added to the individual graphs corresponding to the documents.

policy_name
Name of the extractor policy. If policy_name is null (the default), the triples are added to the information extracted by the default (or the only) extractor policy for the index; if you specify a policy name, the triples are added to the information extracted by that policy.

action
Type of maintenance operation to perform on the triples. The only value currently supported in ADD (the default), which adds the triples that are specified in the rdfxml_content parameter.

Usage Notes

The information extracted from the semantically indexed documents may be incomplete and lacking in proper context. This procedure enables a domain expect to add triples to individual graphs pertaining to specific semantically indexed documents, so that all subsequent SEM_CONTAINS queries can consider these triples in their document search criteria.

This procedure accepts the index name and WHERE clause text to identify the specific documents to be annotated with the additional triples. For example, the where_clause might be specified as a simple predicate involving numeric data, such as `'docId IN (1,2,3)'`.

Examples

The following example annotates a specific document with the semantic index `ArticleIndex` by adding triples to the corresponding individual graph.

```
begin
  sem_rdfctx.maintain_triples(
     index_name      => 'ArticleIndex',
     where_clause    => 'docid = 15',
     rdfxml_content => sys.xmltype(
      '<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
                xmlns:pred="http://myorg.com/pred/">
       <rdf:Description rdf:about=" http://newscorp.com/Org/ExampleCorp">
         <pred:hasShortName
               rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
             Example
         </pred:hasShortName>
     </rdf:Description>
    </rdf:RDF>'));
end;
/
```

14.5 SEM_RDFCTX.SET_DEFAULT_POLICY

Format

```
SEM_RDFCTX.SET_DEFAULT_POLICY(
     index_name   IN VARCHAR2,
     policy_name  IN VARCHAR2);
```

Description

Sets the default extractor policy for a semantic index that is configured with multiple extractor policies.

Parameters

index_name
Name of the semantic index for documents.

policy_name
Name of the extractor policy to be used as the default extractor policy for the specified semantic index. Must be one of the extractor policies listed in the PARAMETERS clause of the CREATE INDEX statement that created `index_name`.

Usage Notes

When you create a semantic index for documents, you can specify multiple extractor policies as a space-separated list of names in the PARAMETERS clause of the CREATE INDEX statement. As explained in Semantically Indexing Documents (page 4-5), the first policy from this list is used as the default extractor policy for all SEM_CONTAINS

queries that do not identify an extractor policy by name. You can use the SEM_RDFCTX.SET_DEFAULT_POLICY procedure to set a different default policy for the index.

Examples

The following example sets `CITY_EXTR` as the default extractor policy for the `ArticleIndex` index.

```
begin
  sem_rdfctx.set_default_policy (index_name => 'ArticleIndex',
                                 policy_name => 'CITY_EXTR');
end;
/
```

14.6 SEM_RDFCTX.SET_EXTRACTOR_PARAM

Format

```
SEM_RDFCTX.SET_EXTRACTOR_PARAM(
     param_key    IN VARCHAR2,
     patam_value  IN VARCHAR2,
     param_desc   IN VARCHAR2);
```

Description

Configures the Oracle Database semantic indexing support to work with external information extractors, such as Calais and GATE.

Parameters

param_key
Key for the parameter to be set.

param_value
Value for the parameter to be set.

param_desc
Short description for the parameter to be set.

Usage Notes

You must have the SYSDBA role to use this procedure.

To work with the Calais extractor type (see Configuring the Calais Extractor type (page 4-12)), you must specify values for the following parameters:

- `CALAIS_WS_ENDPOINT`: Web service end point for Calais.
- `CALAIS_KEY`: License key for Calais.
- `CALAIS_WS_SOAPACTION`: SOAP action for the Calais Web service.

To work with the General Architecture for Text Engineering (GATE) extractor type (see Working with General Architecture for Text Engineering (GATE) (page 4-12)), you must specify values for the following parameters:

- `GATE_NLP_HOST`: Host for the GATE NLP Listener.

- GATE_NLP_PORT: Port for the GATE NLP Listener.

In addition to these parameters, you may need to specify a value for the HTTP_PROXY parameter to work with information extractors or index documents that are outside the firewall.

A database instance only has one set of values for these parameters, and they are used for all instances of semantic indexes using the corresponding information extractor. You can use this procedure if you need to change the existing values of any of the parameters.

Examples

For examples, see the following sections:

- Configuring the Calais Extractor type (page 4-12)
- Working with General Architecture for Text Engineering (GATE) (page 4-12)

15

SEM_RDFSA Package Subprograms

The SEM_RDFSA package contains subprograms (functions and procedures) for providing fine-grained access control to RDF data, using either a virtual private database (VPD) or Oracle Label Security (OLS). To use the subprograms in this chapter, you should understand the conceptual and usage information in RDF Semantic Graph Overview (page 1-1) and Fine-Grained Access Control for RDF Data (page 5-1).

This chapter provides reference information about the subprograms, listed in alphabetical order.

15.1 SEM_RDFSA.ADD_VPD_CONSTRAINT

Format

```
SEM_RDFSA.ADD_VPD_CONSTRAINT(
     policy_name    IN VARCHAR2,
     constr_name    IN VARCHAR2,
     match_pattern  IN VARCHAR2,
     apply_pattern  IN VARCHAR2,
     constr_group   IN VARCHAR2 DEFAULT NULL);
```

Description

Adds a data access constraint to a VPD policy.

Parameters

policy_name
Name of an existing VPD policy.

constr_name
Unique case-insensitive name for the constraint to be defined in the VPD policy.

match_pattern
Match pattern that determines the target of the constraint.

apply_pattern
The constraint's apply pattern that enforces the access restriction.

constr_group
Case-insensitive name for the group to which the constraint belongs.

Usage Notes

This procedure adds a data access constraint to a VPD policy. Only the owner of a VPD policy can add a new constraint to the policy. The newly added constraint comes into effect immediately and it is enforced for queries on all RDF models associated with the VPD policy.

A constraint group may be specified for each policy such that they can be selectively activated and deactivated at runtime based on the application context (as explained in VPD Policy for RDF Data (page C-3)). A constraint with no constraint group is always active. See Data Access Constraints (page C-7) for details about the match and apply patterns passed to this procedure.

See also the MDSYS.RDFVPD_POLICY_CONSTRAINTS view, which is described in RDFVPD_POLICY_CONSTRAINTS View (page C-11).

For information about support for VPD, .see Virtual Private Database Support in RDF Semantic Graph (page C-1).

Examples

The following example adds data access constraint for the RDF Class Contracts to the VPD policy and assigns it a constraint group named `user`:

```
begin
  sem_rdfsa.add_vpd_constraint(
          policy_name   => 'contracts_policy',
          constr_name   => 'andy_constraint_1',
          match_pattern =>'{?contract  rdf:type
                              <http://www.myorg.com/classes/Contract>}',
          apply_pattern => '{?contract  pred:hasMember
              "sys_context('sa$appctx','app_user_uri'}"^^orardf:instruction }',
          constr_group  => 'user');
end;
/
```

The preceding example makes use of secure application context to identify the user issuing the query and binds its value in the apply pattern to ensure that the user has access to a specific contract. The secure application context with the namespace `sa $appctx` is expected to initialize the value for the attribute `app_user_uri` to the URI identifying the user logged in.

The following example adds a data access constraint for the RDF property `hasContractValue` to the same VPD policy and assigns it to a constraint group named `vp`.

```
begin
  sem_rdfsa.add_vpd_constraint(
          policy_name   => 'contracts_policy',
          constr_name   => 'vp_access_to_contr_value',
          match_pattern => '{?contract  pred:hasContractValue ?cvalue }',
          apply_pattern => '{?contract  pred:drivenBy         ?dept .
                            ?dept       pred:hasVP
              "sys_context('sa$appctx','app_user_uri'}"^^orardf:instruction }',
          constr_group  => 'vp');
end;
/
```

15.2 SEM_RDFSA.APPLY_OLS_POLICY

Format

```
SEM_RDFSA.APPLY_OLS_POLICY(
     policy_name    IN VARCHAR2,
     rdfsa_options  IN NUMBER  DEFAULT SEM_RDFSA.SECURE_SUBJECT,
     table_options  IN VARCHAR2 DEFAULT 'ALL_CONTROL',
     label_function IN VARCHAR2 DEFAULT NULL,
     predicate      IN VARCHAR2 DEFAULT NULL);
```

Description

Applies an OLS policy to the semantic data store.

Parameters

policy_name
Name of an existing OLS policy.

rdfsa_options
Options specifying the mode of fine-grained access control to be enabled for RDF data. The default option for securing RDF data involves assigning sensitivity labels for the resources appearing the triples' subject position. You can override the defaults by using the `rdfsa_options` parameter and specifying one of the constants defined in Table 15-1 (page 15-4) in the Usage Notes.

table_options
Policy enforcement options. The default value (`ALL_CONTROL`) is the only supported value for this procedure.

label_function
A string invoking a function to return a label value to use as the default.

predicate
An additional predicate to combine with the label-based predicate.

Usage Notes

The OLS policy specified with this procedure must be created with CTXT1 as the column name, and it should use default policy options. For information about policy options, see *Oracle Label Security Administrator's Guide*.

This procedure invokes the `sa_policy_admin.apply_table_policy` procedure on multiple tables defined in the MDSYS schema. The parameters `table_options`, `label_function`, and `predicate` for the SEM_RDFSA.APPLY_OLS_POLICY procedure have same semantics as the parameters with same names in the sa_policy_admin.apply_table_policy procedure.

For the `rdfsa_options` parameter, you can specify the package constant for the desired option. Table 15-1 (page 15-4) lists these constants and their descriptions.

Table 15-1 SEM_RDFSA Package Constants for rdfsa_options Parameter

Constant	Description
SEM_RDFSA.SECURE_SUBJECT	Assigns sensitivity labels for the resources appearing the triples' subject position.
SEM_RDFSA.SECURE_PREDICATE	Assigns sensitivity labels for the resources appearing the triples' predicate position.
SEM_RDFSA.SECURE_OBJECT	Assigns sensitivity labels for the resources appearing the triples' object position.
SEM_RDFSA.TRIPLE_LEVEL_ONLY	Applies triple-level security. Provides good performance, and eliminates the need to assign labels to individual resources. (Requires that Patch 9819833, available from My Oracle Support, be installed.)
SEM_RDFSA.OPT_DEFINE_BEFORE_USE	Restricts the use of an RDF resource in a triple before the sensitivity label is defined for the resource. If this option is not specified, the user's initial row label is used as the default label for the resource upon first use.
SEM_RDFSA.OPT_RELAX_TRIPLE_LABEL	Relaxes the dominating relationship that exists between the triple label and the labels associated with all its components. With this option, a triple can be defined if the user has READ access to all the triple components and the triple label may not bear any relationship with the component labels. Without this option, the triple label should at least cover the label for all its components.

You can specify a function in the `label_function` parameter to generate custom labels for newly inserted triples. The label function is associated with the MDSYS.RDF_LINK$ table, and the columns in this table may be configured as parameters to the label function as shown in the following example:

```
fgac_admin.new_triple_label(:new.model_id,
                            :new.start_node_id,
                            :new.p_value_id,
                            :new.canon_end_node_id)'
```

Because the OLS policy is applied to more than one table with different structures, the only valid column reference in any predicates assigned to the `predicate` parameter is that of the label column: CTXT1. If OLS is enabled for a semantic data store with existing data, you can specify a predicate of the form `'OR CTXT1 is null'` to be able to continue using this data with no access restrictions.

An OLS-enabled semantic data store uses sensitivity labels for all the RDF triples organized in multiple models. User access to such triples, through model views and SEM_MATCH queries, is restricted by the OLS policy. Additionally, independent of a user owning the application table, access to the triple column (of type SDO_RDF_TRIPLE_S) in the table is restricted to users with FULL access privileges with the OLS policy.

The triples are inserted into a specific RDF model using the INSERT privileges on the corresponding application table. A sensitivity label for the new triple is generated using the user's session context (initial row label) or the label function. The triple is validated for any RDF policy violations using labels associated with the triple

components. Although the triple information may not be accessed trough the application table, the model view may be queried to access the triples, while enforcing the OLS policy restrictions. If you have the necessary policy privileges (such as writeup, writeacross), you can update the CTXT1 column in the model view to reset the label assigned to the triple. The new label is automatically validated for any RDF policy violations involving the triple components. Update privilege on the CTXT1 column of the model view is granted to the owner of the model, and this user may selectively grant this privilege to other users.

If the RDF models are created in schemas other than the user with FULL access, necessary privileges on the model objects -- specifically, read/write access on the application table, read access to the model view, and write access to the CTXT1 column in the model view -- can be granted to such users for maintenance operations. These operations include bulk loading into the model, resetting any sensitivity labels assigned to the triples, and creating entailments using the model.

To disable the OLS policy, use the SEM_RDFSA.DISABLE_OLS_POLICY (page 15-8) procedure.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example enable secure access to RDF data with secure subject and secure predicate options.

```
begin
  sem_rdfsa.apply_ols_policy(
        policy_name   => 'defense',
        rdfsa_options => sem_rdfsa.SECURE_SUBJECT+
                         sem_rdfsa.SECURE_PREDICATE);
end;
/
```

The following example extends the preceding example by specifying a Define Before Use option, which allows a user to define a triple only if the triple components secured (Subject, Predicate or Object) are predefined with an associated sensitivity label. This configuration is effective if the user inserting the triple does not have execute privileges on the SEM_RDFSA package.

```
begin
  sem_rdfsa.apply_ols_policy(
        policy_name   => 'defense',
        rdfsa_options => sem_rdfsa.SECURE_SUBJECT+
                         sem_rdfsa.SECURE_PREDICATE+
                         sem_rdfsa.OPT_DEFINE_BEFORE_USE);
end;
/
```

15.3 SEM_RDFSA.APPLY_VPD_POLICY

Format

```
SEM_RDFSA.APPLY_VPD_POLICY(
     policy_name  IN VARCHAR2,
     model_name   IN VARCHAR2);
```

Description

Applies a VPD policy to an RDF model.

Parameters

policy_name
Name of an existing VPD policy.

model_name
Name of the model to which to apply the VPD policy.

Usage Notes

This procedure applies a VPD policy to an RDF model. The owner of a VPD policy can apply it to any model in the database instance. The owner must also have EXECUTE privileges on the SYS.DBMS_RLS package.

After a policy is applied to an RDF model, the data stored in the model and the data inferred from this model can be accessed only using SPARQL query patterns within a SEM_MATCH operation. All other forms of data access (such as through model views or using classic graph pattern syntax in a SEM_MATCH query) are not permitted.

A VPD policy may not applied to a model participating in a virtual model. Similarly, a virtual model may not be defined to include any VPD-enabled models

See also the MDSYS.RDFVPD_POLICIES view, which is described in RDFVPD_POLICIES View (page C-10).

For information about support for VPD, see Virtual Private Database Support in RDF Semantic Graph (page C-1).

Examples

The following example applies a VPD policy to an RDF model.

```
begin
  sem_rdfsa.apply_vpd_policy(
      policy_name => 'contracts_policy',
      model_name  => 'contracts');
end;
/
```

15.4 SEM_RDFSA.CREATE_VPD_POLICY

Format

```
SEM_RDFSA.CREATE_VPD_POLICY(
     policy_name    IN VARCHAR2,
     namespace_map  IN RDF_ALIASES DEFAULT NULL,
    policy_context  IN VARCHAR2 DEFAULT NULL);
```

Description

Creates a new VPD policy in a user schema.

Parameters

policy_name
Name for the VPD policy.

namespace_map
A mapping of namespaces and their prefixes to be used in the VPD constraints.

policy_context
Name of the context created specifically to manage constraint groups defined in the VPD policy.

Usage Notes

You must have EXECUTE privileges on the SYS.DBMS_RLS package.

The namespace map associated with the policy may be used to define namespace prefixes that are subsequently used in the policy's metadata statements added using the SEM_RDFSA.MAINT_VPD_METADATA (page 15-10) procedure and the policy's data access constraints defined using the SEM_RDFSA.ADD_VPD_CONSTRAINT (page 15-1) procedure.

The name of the application context assigned to the `policy_context` parameter is used to manage the constraint groups dynamically at runtime. Such context must be created using the CREATE CONTEXT command before associating it with the VPD policy. Each data access constraint defined in the VPD policy may specify the name of a constraint group to which it belongs. At runtime, the name of the constraint group is matched with the name of the `policy_context` value to selectively activate the group. (For more information, see VPD Policy for RDF Data (page C-3)).

See also the MDSYS.RDFVPD_POLICIES view, which is described in RDFVPD_POLICIES View (page C-10).

For information about support for VPD, see Virtual Private Database Support in RDF Semantic Graph (page C-1).

Examples

The following example creates a VPD policy with a namespace map and also associates a policy context to manage constraint groups.

```
create context contracts_appctx using sec_admin.contracts_appctx;
begin
  sem_rdfsa.create_vpd_policy(
        policy_name  => 'contracts_policy',
        namespace_map =>
          mdsys.rdf_aliases(
            mdsys.rdf_alias('','http://www.myorg.com/
classes/'),
            mdsys.rdf_alias('pred','http://www.myorg.com/pred/'),
            mdsys.rdf_alias('emp','http://www.myorg.com/employee/')),
        policy_context => 'contracts_appctx');
end;
/
```

15.5 SEM_RDFSA.DELETE_VPD_CONSTRAINT

Format

```
SEM_RDFSA.DELETE_VPD_CONSTRAINT(
     policy_name    IN VARCHAR2,
     constr_name    IN VARCHAR2);
```

Description

Deletes a data access constraint from a VPD policy.

Parameters

policy_name
Name of an existing VPD policy.

constr_name
Unique case-insensitive name for the constraint to be deleted from the VPD policy.

Usage Notes

This procedure deletes a data access constraint from a VPD policy. Only the owner of a VPD policy can delete a constraint from the policy.

See also the MDSYS.RDFVPD_POLICY_CONSTRAINTS view, which is described in RDFVPD_POLICY_CONSTRAINTS View (page C-11).

For information about support for VPD, see Virtual Private Database Support in RDF Semantic Graph (page C-1).

Examples

The following example adds data access constraint named `vp_access_to_contr_value` from the VPD policy named `contracts_policy`:

```
begin
  sem_rdfsa.delete_vpd_constraint(
      policy_name   => 'contracts_policy',
      constr_name   => 'vp_access_to_contr_value');
end;
/
```

15.6 SEM_RDFSA.DISABLE_OLS_POLICY

Format

```
SEM_RDFSA.DISABLE_OLS_POLICY;
```

Description

Disables the OLS policy that has been previously applied to or enabled on the semantic data store.

Parameters

(None.)

Usage Notes

You can use this procedure to disable temporarily the OLS policy that had been applied to or enabled for the semantic data store. The user disabling the policy should have the necessary privileges to administer OLS policies and should also have access to the OLS policy applied to RDF data.

The sensitivity labels assigned to various RDF resources and triples are preserved and the OLS policy may be re-enabled to enforce them. New resources with specific labels can be added, or labels for existing triples and resources can be updated when the OLS policy is disabled.

To apply an OLS policy, use the SEM_RDFSA.APPLY_OLS_POLICY (page 15-3) procedure; to enable an OLS policy that had been disabled, use the SEM_RDFSA.ENABLE_OLS_POLICY (page 15-10) procedure.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example disables the OLS policy for the semantic data store.

```
begin
  sem_rdfsa.disable_ols_policy;
end;
/
```

15.7 SEM_RDFSA.DROP_VPD_POLICY

Format

```
SEM_RDFSA.DROP_VPD_POLICY(
     policy_name   IN VARCHAR2);
```

Description

Drops an unused VPD policy.

Parameters

policy_name
Name of an existing VPD policy.

Usage Notes

This procedure permanently drops an unused VPD policy along with its metadata and constraints. You must be the owner of a VPD policy or a user with a DBA role to perform this operation.

For information about support for VPD, see Virtual Private Database Support in RDF Semantic Graph (page C-1).

Examples

The following example adds data access constraint named `vp_access_to_contr_value` from the VPD policy named `contracts_policy`:

```
begin
  sem_rdfsa.drop_vpd_policy(
      policy_name   => 'contracts_policy',
      constr_name   => 'vp_access_to_contr_value');
end;
/
```

15.8 SEM_RDFSA.ENABLE_OLS_POLICY

Format

```
SEM_RDFSA.ENABLE_OLS_POLICY;
```

Description

Enables the OLS policy that has been previously disabled.

Parameters

(None.)

Usage Notes

You can use this procedure to enable the OLS policy that had been disabled for the semantic data store. The user enabling the policy should have the necessary privileges to administer OLS policies and should also have access to the OLS policy applied to RDF data.

To disable an OLS policy, use the SEM_RDFSA.DISABLE_OLS_POLICY (page 15-8) procedure.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example enables the OLS policy for the semantic data store.

```
begin
  sem_rdfsa.enable_ols_policy;
end;
/
```

15.9 SEM_RDFSA.MAINT_VPD_METADATA

Format

```
SEM_RDFSA.MAINT_VPD_METADATA(
     policy_name  IN VARCHAR2,
     t_subject    IN VARCHAR2,
     t_predicate  IN VARCHAR2,
     t_object     IN VARCHAR2,
     action       IN VARCHAR2 DEFAULT 'ADD');
```

Description

Maintains the VPD metadata by accepting RDF schema statements with their subject, predicate, and object terms.

Parameters

policy_name
Name of an existing VPD policy.

t_subject
Subject of the metadata triple.

t_predicate
Predicate of the metadata triple.

t_object
Object of the metadata triple.

action
Maintenance operation to be performed: `ADD` (add the specified triple to the metadata) or `DELETE` (delete the specified triple from the metadata).

Usage Notes

This procedure maintains the metadata associated with the VPD policy by accepting RDF Schema statements and performing appropriate actions on its dictionary. Only the owner of the VPD policy or a user with FULL access to the VPD policy can maintain its metadata.

The types metadata statements that you can add using this procedure are determined by the terms assigned to the `t_predicate` parameter. The accepted values for this parameter are:

```
http://www.w3.org/2000/01/rdf-schema#domain
http://www.w3.org/2000/01/rdf-schema#range
http://www.w3.org/2000/01/rdf-schema#subClassOf
http://www.w3.org/2000/01/rdf-schema#subPropertyOf
http://www.w3.org/2002/07/owl#equivalentProperty
```

MDSYS.RDFVPD_PREDICATE_MDATA and MDSYS.RDFVPD_RESOURCE_REL views, which are described in RDFVPD_PREDICATE_MDATA View (page C-11) and RDFVPD_RESOURCE_REL View (page C-12), respectively.

For information about support for VPD, see Virtual Private Database Support in RDF Semantic Graph (page C-1).

Examples

The following example specifies `Contract` as a subclass of `Project` and adds domain information for the `hasContractValue` property:

```
begin
  sem_rdfsa.maint_vpd_metadata(
      policy_name => 'contracts_policy',
      t_subject   => '<http://www.myorg.com/classes/Project>',
      t_predicate => '<http://www.w3.org/2000/01/rdf-schema#subClassOf>',
      t_object    => '<http://www.myorg.com/classes/Contract>',
      action      => 'add');
```

```
end;

begin
  sem_rdfsa.maint_vpd_metadata(
      policy_name => 'contracts_policy',
      t_subject   => '<http://www.myorg.com/property/hasConfValue>',
      t_predicate => 'rdfs:domain',
      t_object    => '<http://www.myorg.com/classes/Contract>');
end;
```

15.10 SEM_RDFSA.REMOVE_OLS_POLICY

Format

```
SEM_RDFSA.REMOVE_OLS_POLICY;
```

Description

Permanently removes or detaches the OLS policy from the semantic data store.

Parameters

(None.)

Usage Notes

You should have the necessary privileges to administer OLS policies, and you should also have access to the OLS policy applied to RDF data. Once the OLS policy is detached from the semantic data store, all the sensitivity labels previously assigned to the triples and resources are lost.

This operation drops objects that are specifically created to maintain the RDF security policies.

To apply an OLS policy, use the SEM_RDFSA.APPLY_OLS_POLICY (page 15-3) procedure.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example removes the OLS policy that had been previously applied to the semantic data store.

```
begin
  sem_rdfsa.remove_ols_policy;
end;
/
```

15.11 SEM_RDFSA.REMOVE_VPD_POLICY

Format

```
SEM_RDFSA.REMOVE_VPD_POLICY(
     policy_name IN VARCHAR2,
     model_name  IN VARCHAR2);
```

Description

Removes (detaches) a VPD policy from an RDF model.

Parameters

policy_name
Name of an existing VPD policy.

model_name
Name of the model to which to apply the VPD policy.

Usage Notes

This procedure permanently removes or detaches a VPD policy from an RDF Model. You must be the owner of the VPD policy or a user with a DBA role to perform this operation.

After a VPD policy is removed, fine-grained access control for the specific model is turned off and the users with SELECT privileges on the model view can access all the data in the model.

For information about support for VPD, see Virtual Private Database Support in RDF Semantic Graph (page C-1).

Examples

The following example removes a VPD policy from an RDF model.

```
begin
  sem_rdfsa.remove_vpd_policy(
      policy_name => 'contracts_policy',
      model_name  => 'contracts');
end;
/
```

15.12 SEM_RDFSA.RESET_MODEL_LABELS

Format

```
SEM_RDFSA.RESET_MODEL_LABELS(
     model_name  IN VARCHAR2);
```

Description

Resets the labels associated with a model or with global resources; requires that the associated model or models be empty.

Parameters

model_name
Name of the model for which the labels should be reset, or the string `RDF$GLOBAL` to reset the labels associated with all global resources.

Usage Notes

If you specify a model name, the model must be empty. If you specify `RDF$GLOBAL`, all the models must be empty (that is, no triples in the RDF repository).

You must have FULL access privilege with the OLS policy applied to the semantic data store.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example removes all resources and their labels associated with the `Contracts` model.

```
begin
   sem_rdfsa.reset_model_labels(model_name => 'Contracts');
end;
/
```

15.13 SEM_RDFSA.SET_PREDICATE_LABEL

Format

```
SEM_RDFSA.SET_PREDICATE_LABEL(
     model_name   IN VARCHAR2,
     predicate    IN VARCHAR2,
     label_string IN VARCHAR2);
```

Description

Sets a sensitivity label for a predicate at the model level or for the whole repository.

Parameters

model_name
Name of the model to which the predicate belongs, or the string `RDF$GLOBAL` if the same label should applied for the use of the predicate in all models.

predicate
Predicate for which the label should be assigned.

label_string
OLS row label in string representation.

Usage Notes

If you specify a model name, you must have read access to the model and execute privileges on the SEM_RDFSA package to perform this operation. If you specify `RDF$GLOBAL`, you must have FULL access privilege with the OLS policy applied to RDF data.

You must have access to the specified label and OLS policy privilege to overwrite an existing label if a label already exists for the predicate. The SECURE_PREDICATE option must be enabled for RDF data.

If an existing predicate label is updated with this operation, the labels for the triples using this predicate must all dominate the new predicate label. The only exception is when the OPT_RELAX_TRIPLE_LABEL option is chosen for the OLS-enabled RDF data.

If you specify RDF$GLOBAL, a global predicate with a unique sensitivity label across models is created. If the same predicate is previously defined in one or more models, the global label dominates all such labels and the model-specific labels are replaced for the given predicate.

After a label for a predicate is set, new triples with the predicate can be added only if the triple label (which may be initialized from user's initial row label or using a label function) dominates the predicate's sensitivity label. This dominance relationship can be relaxed with the OPT_RELAX_TRIPLE_LABEL option, in which case the user should at least have read access to the predicate to be able to define a new triple using the predicate.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example sets a predicate label for Contracts model and another predicate label for all models in the database instance.

```
begin
  sem_rdfsa.set_predicate_label(
         model_name   => 'contracts',
         predicate    => '<http://www.myorg.com/pred/hasContractValue>',
         label_string => 'TS:US_SPCL');
end;
/

begin
  sem_rdfsa.set_predicate_label(
         model_name   => 'rdf$global',
         predicate    => '<http://www.myorg.com/pred/hasStatus>',
         label_string => 'SE:US_SPCL:US');
end;
/
```

15.14 SEM_RDFSA.SET_RDFS_LABEL

Format

```
SEM_RDFSA.SET_RDFS_LABEL(
     label_string  IN VARCHAR2,
     inf_override  IN VARCHAR2);
```

Description

Sets a sensitivity label for RDFS schema elements.

Parameters

label_string
OLS row label in string representation, to be used as the sensitivity label for all RDF schema constructs.

inf_override
OLS row label to be used as the override for generating labels for inferred triples.

Usage Notes

This procedure sets or resets the sensitivity label associated with the RDF schema resources, often recognized by `http://www.w3.org/1999/02/22-rdf-syntax-ns#` and `http://www.w3.org/2000/01/rdf-schema#` prefixes for their URIs. You can assign a sensitivity label with restricted access to these resources, so that operations such as creating new RDF classes and adding new properties can be restricted to users with higher privileges.

You must have FULL access privilege with policy applied to RDF data.

RDF schema elements implicitly use the relaxed triple label option, so that the triples using RDFS and OWL constructs for subject, predicate, or object are not forced to have a sensitivity label that dominates the labels associated with the schema constructs. Therefore, a user capable of defining new RDF classes and properties must least have read access to the schema elements.

When RDF schema elements are referred to in the inferred triples, the system-defined and custom label generators consider the inference override label in determining the appropriate label for the inferred triples. If a custom label generator is used, this override label is passed instead of the actual label when an RDF schema element is involved.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example sets a label with a unique compartment for all RDF schema elements. A user capable of defining new RDF classes and properties is expected to have an exclusive membership to the compartment.

```
begin
  sem_rdfsa.set_rdfs_label(
         label_string  => 'SE:RDFS:',
         inf_override  => 'SE:US_SPCL:US');
end;
/
```

15.15 SEM_RDFSA.SET_RESOURCE_LABEL

Format

```
SEM_RDFSA.SET_RESOURCE_LABEL(
     model_name   IN VARCHAR2,
     resource_uri IN VARCHAR2,
     label_string IN VARCHAR2,
     resource_pos IN VARCHAR2 DEFAULT 'S');
```

Description

Sets a sensitivity label for a resource that may be used in the subject and/or object position of a triple.

Parameters

model_name
Name of the model to which the resource belongs, or the string `RDF$GLOBAL` if the same label should applied for using the resource in all models.

resource_uri
URI for the resource that may be used as subject or object in one or more triples.

label_string
OLS row label in string representation.

resource_pos
Position of the resource within a triple: `S`, `O`, or `S,O`. You can specify up to two separate labels for the same resource, one to be considered when the resource is used in the subject position of a triple and the other to be considered when it appears in the object position. The values '`S`', '`O`' or '`S,O`' set a label for the resource in subject, object or both subject and object positions, respectively.

Usage Notes

If you specify a model name, you must have read access to the model and execute privileges on the SEM_RDFSA package to perform this operation. If you specify `RDF $GLOBAL`, you must have FULL access privilege with the OLS policy applied to RDF data.

You must have access to the specified label and OLS policy privilege to overwrite an existing label if a label already exists for the predicate. The SECURE_PREDICATE option must be enabled for RDF data.

If an existing resource label is updated with this operation, the labels for the triples using this resource in the specified position must all dominate the new resource label. The only exception is when the OPT_RELAX_TRIPLE_LABEL option is chosen for the OLS-enabled RDF data.

If you specify `RDF$GLOBAL`, a global resource with a unique sensitivity label across models is created. If the same resource is previously defined in one or more models with the same triple position, the global label dominates all such labels and the model-specific labels are replaced for the given resource in that position.

After a label for a predicate is set, new triples using the resource in the specified position can be added only if the triple label dominates the resource's sensitivity label. This dominance relationship can be relaxed with OPT_RELAX_TRIPLE_LABEL option, in which case, the user should at least have read access to the resource.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example sets sensitivity labels for multiple resources based on their position.

```
begin
  sem_rdfsa.set_resource_label(
         model_name   => 'contracts',
         resource_uri => '<http://www.myorg.com/contract/projectHLS>',
         label_string => 'SE:US_SPCL:US',
         resource_pos => 'S,O');
end;
/

begin
  sem_rdfsa.set_resource_label(
       model_name   => 'rdf$global',
       resource_uri => '<http://www.myorg.com/contract/status/Complete>',
```

```
       label_string => 'SE:US_SPCL:US',
       resource_pos => 'O');
end;
/
```

15.16 SEM_RDFSA.SET_RULE_LABEL

Format

```
SEM_RDFSA.SET_RULE_LABEL(
     rule_base    IN VARCHAR2,
     rule_name    IN VARCHAR2,
     label_string IN VARCHAR2);
```

Description

Sets sensitivity label for a rule belonging to a rulebase.

Parameters

rule_base
Name of an existing RDF rulebase.

rule_name
Name of the rule belonging to the rulebase.

label_string
OLS row label in string representation.

Usage Notes

The sensitivity label assigned to the rule is used to generate the label for the inferred triples when an appropriate label generator option is chosen.

You must have access have access to the rulebase, and you must have FULL access privilege with the OLS policy can assign labels for system-defined rules in the RDFS rulebase.

There is no support for labels assigned to user-defined rules.

For information about support for OLS, see Fine-Grained Access Control for RDF Data (page 5-1).

Examples

The following example assigns a sensitivity label for an RDFS rule.

```
begin
sem_rdfsa.set_rule_label (rule_base    => 'RDFS',
                          rule_name    => 'RDF-AXIOMS',
                          label_string => 'SE:US_SPCL:');
end;
/
```

A

Enabling, Downgrading, or Removing RDF Semantic Graph Support

This appendix describes the required steps that you must perform before you can use any types, synonyms, or PL/SQL packages related to RDF Semantic Graph support in the current Oracle Database release. You must run one or more scripts, and you must ensure that Spatial and Graph is installed and the Partitioning option is enabled. These requirements are explained in Enabling RDF Semantic Graph Support (page A-1).

This appendix also describes the steps if, after enabling RDF Semantic Graph support, you need to do any of the following:

- Downgrade the RDF Semantic Graph support to that provided with a previous Oracle Database release, as explained in Downgrading RDF Semantic Graph Support to a Previous Release (page A-6).
- Remove all support for RDF Semantic Graph from the database, as explained in Removing RDF Semantic Graph Support (page A-7).

A.1 Enabling RDF Semantic Graph Support

Before you can use any types, synonyms, or PL/SQL packages related to RDF Semantic Graph support in the current Oracle Database release, you must either install the capabilities in a new Oracle Database installation or upgrade the capabilities from a previous release, following the steps in whichever of the following sections applies to your situation:

- Enabling RDF Semantic Graph Support in a New Database Installation (page A-1)
- Upgrading RDF Semantic Graph Support from Release 11.1 or 11.2 (page A-4)

In addition, you must ensure that Spatial and Graph is installed and the Partitioning option is enabled, as explained in Spatial and Partitioning Requirements (page A-6).

A.1.1 Enabling RDF Semantic Graph Support in a New Database Installation

To enable RDF Semantic Graph support in a new installation of Oracle Database Release 11.2 or later, follow these steps:

1. Connect to the database as the SYS user with SYSDBA privileges (SYS AS SYSDBA, and enter the SYS account password when prompted).
2. For Release 12.1.0.2 only (not Release 12.1.0.1 or earlier, and not later than Release 12.1.0.2), if you have not already enabled the Oracle Spatial and Graph GeoRaster feature, enter the following:

```
EXECUTE mdsys.enableGeoRaster;
```

If the Oracle Spatial and Graph GeoRaster feature has not been enabled, Oracle Spatial and Graph RDF Semantic Graph may be misconfigured in Release 12.1.0.2.0 because a system trigger for DDL operations on application tables associated with semantic models will not exist. That system trigger is created when GeoRaster is enabled; and so for users who chose not to enable GeoRaster, the trigger will not exist.

or more details, including how to check for data corruption and what to do if corruption has occurred, see the Release 12.1.0.2–specific material later in this topic.

3. Start SQL*Plus, and enter the following statement:
   - Linux: `@$ORACLE_HOME/md/admin/catsem.sql`
   - Windows: `@%ORACLE_HOME%\md\admin\catsem.sql`

If a semantic network already exists, or if any RDF Semantic Graph types, tables, or PL/SQL packages already exist, the `catsem.sql` script exits with an error.

If the script completes successfully, a row with the following column values is inserted into the MDSYS.RDF_PARAMETER table:

- NAMESPACE: `MDSYS`
- ATTRIBUTE: `SEM_VERSION`
- VALUE: (string starting with `12.1`)
- DESCRIPTION: `VALID`

If you use Workspace Manager with RDF data, you must also run the appropriate script *after* you run `catsem.sql`, as explained in Enabling Workspace Manager Support for RDF Data (page 6-2).

In Release 12.1.0.2 only, if the Oracle Spatial and Graph GeoRaster feature has not been enabled, Oracle Spatial and Graph RDF Semantic Graph may be misconfigured because a system trigger for DDL operations on application tables associated with semantic models will not exist. To create this trigger, you must execute `mdsys.enableGeoRaster` as SYS.

This problem affects both new Oracle Database version 12.1.0.2.0 installations and databases that have been upgraded to version 12.1.0.2.0.

If this DDL trigger is not present, TRUNCATE operations on an application table will remove all rows from the application table, but the associated triples stored in MDSYS.RDF_LINK$ will not be removed. In this case, there will be a mismatch between the application table and its associated semantic model, and you must correct this mismatch.

Use SQL queries in the following form to check for a data mismatch (for example, on an application table named ATAB associated with a semantic model named M):

```
SELECT COUNT(*) FROM ATAB;
SELECT SUM(COST) FROM MDSYS.RDFM_M;
```

If the two queries return identical results, then there is no data mismatch, and no further action beyond executing `mdsys.enableGeoRaster` is required.

However, if the two queries return different results, in addition to executing `mdsys.enableGeoRaster`,you should take further action with each corrupted

model. If the model is security enabled with an OLS policy, contact Oracle Support. For other models, you should take one of the following actions:

- Re-create the model, as follows:

  1. Build a staging table from the contents of the application table. For example, build staging table STAB from the contents of application table ATAB with SDO_RDF_TRIPLE_S column TRI, as follows:

```
create table stab as
select
  a.tri.get_subject() rdf$stc_sub,
  a.tri.get_property() rdf$stc_pred,
  a.tri.get_object() rdf$stc_obj,
  (case when(instr(a.tri.get_model(),':') > 0) then
          substr(a.tri.get_model(),instr(a.tri.get_model(),':')+1)
        else null
   end) rdf$stc_graph
from atab a;
```

     Alternatively, for very large models, you can use queries similar to the following to fill two staging tables (one for VARCHAR object terms and one for CLOB object terms). Note that a degree of parallelism of 4 (`/*+ parallel(4) */`) is used in the CREATE TABLE statements in the following example; the degree of parallelism should be customized for your particular hardware setup.

```
-- for varchar literals
create /*+ parallel(4) */ table stab_vc as
select /*+ parallel */
  (case when vs.value_type = 'UR' then '<' || vs.value_name || '>'
        else vs.value_name
   end) rdf$stc_sub,
  ('<' || vp.value_name || '>') rdf$stc_pred,
  (case when vo.value_type = 'UR' then '<' || vo.value_name || '>'
        when vo.value_type = 'BN' then vo.value_name
        else '"' || vo.value_name || '"' ||
             nvl2(vo.language_type,'@' || vo.language_type,'') ||
             nvl2(vo.literal_type,'^^<' || vo.literal_type || '>', '')
   end) rdf$stc_obj,
  (nvl2(vg.value_name,('<' || vg.value_name || '>'),'')) rdf$stc_graph
from atab a, mdsys.rdf_value$ vs, mdsys.rdf_value$ vp, mdsys.rdf_value$ vo,
mdsys.rdf_value$ vg
where
  (a.tri.rdf_s_id = vs.value_id) and
  (a.tri.rdf_p_id = vp.value_id) and
  (a.tri.rdf_c_id = vo.value_id) and
  ((bitand(a.tri.rdf_m_id, 79228162514264337593543950335-4294967295)/
4294967296) = vg.value_id(+)) and
  (vo.long_value is null);

-- for clob literals
create /*+ parallel(4) */ table stab_clob as
select /*+ parallel */
  (case when vs.value_type = 'UR' then '<' || vs.value_name || '>'
        else vs.value_name
   end) rdf$stc_sub,
  ('<' || vp.value_name || '>') rdf$stc_pred,
  (sem_apis.compose_rdf_term(vo.value_name, vo.value_type, vo.literal_type,
vo.language_type, vo.long_value)) as rdf$stc_obj,
  (nvl2(vg.value_name,('<' || vg.value_name || '>'),'')) rdf$stc_graph
```

```
from atab a, mdsys.rdf_value$ vs, mdsys.rdf_value$ vp, mdsys.rdf_value$ vo,
mdsys.rdf_value$ vg
where
  (a.tri.rdf_s_id = vs.value_id) and
  (a.tri.rdf_p_id = vp.value_id) and
  (a.tri.rdf_c_id = vo.value_id) and
  ((bitand(a.tri.rdf_m_id, 79228162514264337593543950335-4294967295)/
4294967296) = vg.value_id(+)) and
  (vo.long_value is not null);
```

2. Drop the model. For example, drop model M, as follows:

```
exec sem_apis.drop_sem_model('m');
```

3. Truncate the application table. For example, truncate ATAB, as follows:

```
truncate table atab;
```

4. Create the model with the application table. For example, create model M from application table ATAB, as follows:

```
exec sem_apis.create_sem_model('m','atab','tri');
```

5. Load the triples or quads from the staging table into the model. For example, load the triples or quads from STAB into model M, as follows. (For larger models, you can instead use the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE (page 11-11) procedure.)

```
insert into atab(tri)
select sdo_rdf_triple_s('m1' || nvl2(s.rdf$stc_graph,':'||s.rdf
$stc_graph,''), s.rdf$stc_sub, s.rdf$stc_pred, s.rdf$stc_obj)
from stab s;
```

   Or, referring to the alternative example in step 1, when using two staging tables for very large models, use the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE (page 11-11) procedure to load data from the VARCHAR table (`stab_vc`) and use INSERT AS SELECT to load data from the CLOB table (`stab_clob`).

- Or, contact Oracle Support.

A.1.2 Upgrading RDF Semantic Graph Support from Release 11.1 or 11.2

If you are upgrading from Oracle Database Release 11.1 or 11.2 that includes the semantic technologies support, the semantic technologies support is automatically upgraded to Release 12.1 or later when the database is upgraded.

However, you may also need to migrate RDF data if you have an existing Release 11.1 or 11.2 RDF network containing triples that include typed literal values of type xsd:float, xsd:double, xsd:boolean, or xsd:time.

To check if you need to migrate RDF data, connect to the database as a user with DBA privileges and query the MDSYS.RDF_PARAMETER table, as follows:

```
SELECT namespace, attribute, value FROM mdsys.rdf_parameter
  WHERE namespace='MDSYS'
  AND attribute IN ('FLOAT_DOUBLE_DECIMAL',
                    'XSD_TIME', 'XSD_BOOLEAN',
                    'DATA_CONVERSION_CHECK');
```

If the FLOAT_DOUBLE_DECIMAL, XSD_TIME, or XSD_BOOLEAN attributes have the string value `INVALID` or if the DATA_CONVERSION_CHECK attribute has the string value `FAILED_UNABLE_TO_LOCK_APPLICATION_TABLES`, `FAILED_INSUFFICIENT_WORKSPACE_PRIVILEGES`, or `FAILED_OLS_POLICIES_ARE_ENABLED`, you need to migrate RDF data.

However, if the FLOAT_DOUBLE_DECIMAL, XSD_TIME, and XSD_BOOLEAN attributes do not exist or have the string value `VALID` and if the DATA_CONVERSION_CHECK attribute does not exist, you do *not* need to migrate RDF data. However, if your semantic network may have any empty RDF literals, see Handling of Empty RDF Literals (page A-6); and if you choose to migrate existing empty literals to the new format, follow the steps in this section.

To migrate RDF data, follow these steps:

1. Connect to the database as the SYS user with SYSDBA privileges (SYS AS SYSDBA, and enter the SYS account password when prompted), and enter: `SET CURRENT_SCHEMA=MDSYS`

2. Ensure that the user MDSYS has the following privileges:

   - INSERT privilege on all application tables in the semantic network
   - ALTER ANY INDEX privilege (optional: only necessary if Semantic Indexing for Documents is being used)
   - ACCESS privilege for any workspace in which version-enabled application tables have been modified (optional: only necessary if Workspace Manager is being used for RDF data)

3. Ensure that any OLS policies for RDF data are temporarily disabled (optional: only necessary if OLS for RDF Data is being used). OLS policies can be re-enabled after running convert_old_rdf_data.

4. Start SQL*Plus. If you want to migrate the RDF data without converting existing empty literals to the new format (see Handling of Empty RDF Literals (page A-6)), enter the following statement:

   ```
   EXECUTE sdo_rdf_internal.convert_old_rdf_data;
   ```

   If you want to migrate the RDF data and also convert existing empty literals to the new format, call `convert_old_rdf_data` with the `flags` parameter set to `'CONVERT_ORARDF_NULL'`. In addition, you can use an optional `tablespace_name` parameter to specify the tablespace to use when creating intermediate tables during data migration. For example, the following statement migrates old semantic data, converts existing `"orardf:null "` values to "", and uses the MY_TBS tablespace for any intermediate tables:

   ```
   EXECUTE sdo_rdf_internal.convert_old_rdf_data(
     flags=>'CONVERT_ORARDF_NULL',
     tablespace_name=>'MY_TBS');
   ```

   The `sdo_rdf_internal.convert_old_rdf_data` procedure may take a significant amount of time to run if the semantic network contains many triples that are using (or affected by use of) xsd:float, xsd:double, xsd:time, or xsd:boolean typed literals.

5. Enter the following statement:

- Linux: `@$ORACLE_HOME/md/admin/semrelod.sql`
- Windows: `@%ORACLE_HOME%\md\admin\semrelod.sql`

Note:

You may encounter the ORA-00904 (invalid identifier) error when executing a SEM_MATCH query if the `sdo_rdf_internal.convert_old_rdf_data` procedure and the `semrelod.sql` script were not run after the upgrade to Release 12.1 or later.

A.1.2.1 Handling of Empty RDF Literals

The way empty-valued RDF literals are handled was changed in Release 11.2. Before this release, the values of empty-valued literals were converted to `"orardf:null"`. In Release 11.2 and later, such values are stored without modification (that is, as ""). However, whether you migrate existing `"orardf:null"` values to "" is optional.

To check if `"orardf:null"` values exist in your semantic network, connect to the database as a user with DBA privileges and query the MDSYS.RDF_PARAMETER table, as follows:

```
SELECT namespace, attribute, value FROM mdsys.rdf_parameter
  WHERE namespace='MDSYS'
  AND attribute = 'NULL_LITERAL';
```

If the NULL_LITERAL attribute has the value `EXISTS`, then `"orardf:null"` values are present in your semantic network.

A.1.3 Spatial and Partitioning Requirements

Oracle Spatial and Graph must be installed before you can use any of the RDF and OWL capabilities. Oracle Locator is not sufficient. For information about Spatial and Graph and Locator, see *Oracle Spatial and Graph Developer's Guide*.

The Partitioning option must be enabled before you can use any of the RDF and OWL capabilities. For licensing information about the Partitioning option, see *Oracle Database Licensing Information*. For usage information about partitioning, see *Oracle Database VLDB and Partitioning Guide*.

A.2 Downgrading RDF Semantic Graph Support to a Previous Release

This section explains how to downgrade the RDF Semantic Graph support, in conjunction with an Oracle Database downgrade to Release 11.2.

Downgrading is ***strongly discouraged***, except for rare cases where it is necessary. If you downgrade to a previous release, you will not benefit from bug fixes and enhancements that have been made in intervening releases.

If you have enabled Workspace Manager support for RDF data, you must remove that support *before* you downgrade RDF Semantic Graph support in the database, as explained in Removing Workspace Manager Support for RDF Data (page 6-2).

A.2.1 Downgrading to Release 11.2 Semantic Technologies Support

If you need to downgrade to Oracle Database Release 11.2, the semantic technologies component will be downgraded automatically when you downgrade the database.

However, any RDF or OWL data that is specific to Release 12 (that is, Release 12 or later RDF/OWL persistent structures that are not supported in previous versions) must be dropped *before* you perform the downgrade, so that the database is compatible with Release 11.2.

To check if any Release 12.1 or later RDF data is incompatible with Release 11.2, perform the following steps:

1. Connect to the database (Release 12.1 or later) as the SYS user with SYSDBA privileges (SYS AS SYSDBA, and enter the SYS account password when prompted).
2. Start SQL*Plus, and enter the following statements:

```
SET SERVEROUT ON
EXECUTE SDO_SEM_DOWNGRADE.CHECK_112_COMPATIBLE;
```

If any RDF data is incompatible with Release 11.2, the procedure generates an error and displays a list of the incompatible data. In this case, you must perform the following steps:

1. Remove any Release 12.1 or later release-specific RDF or OWL data if you have not already done so, as explained earlier in this section.
2. Perform the database downgrade.
3. Connect to the Release 11.2 database as the SYS user with SYSDBA privileges (SYS AS SYSDBA, and enter the SYS account password when prompted).
4. Start SQL*Plus, and enter the following statement:

   - Linux: `@$ORACLE_HOME/md/admin/semrelod.sql`
   - Windows: `@%ORACLE_HOME%\md\admin\semrelod.sql`

   If the script completes successfully, a row with the following column values is inserted into the MDSYS.RDF_PARAMETER table:

   - NAMESPACE: `MDSYS`
   - ATTRIBUTE: `SEM_VERSION`
   - VALUE: (string starting with `112`)
   - DESCRIPTION: `VALID`

   After the `semrelod.sql` script completes successfully, Oracle semantic technologies support for Release 11.2 is enabled and ready to use, and all Release 11.2-compatible data is preserved.

A.3 Removing RDF Semantic Graph Support

This section explains how to remove the RDF Semantic Graph support from the database. Removing this support is ***strongly discouraged***, unless you have a strong reason for doing it. After you remove this support, no applications or database users will be able to use any types, synonyms, or PL/SQL packages related to RDF Semantic Graph support.

If you have enabled Workspace Manager support for RDF data, you must remove that support *before* you remove RDF Semantic Graph support from the database, as explained in Removing Workspace Manager Support for RDF Data (page 6-2).

To remove the RDF Semantic Graph support from the database, perform the following steps:

1. Connect to the database as the SYS user with SYSDBA privileges (SYS AS SYSDBA, and enter the SYS account password when prompted).
2. Start SQL*Plus, and enter the following statement:
   - Linux: `@$ORACLE_HOME/md/admin/semremov.sql`
   - Windows: `@%ORACLE_HOME%\md\admin\semremov.sql`

The `semremov.sql` script drops the semantic network and removes any RDF Semantic Graph types, tables, and PL/SQL packages:

B

SEM_MATCH Support for Spatial Queries

This appendix provides reference information for SPARQL extension functions for performing spatial queries in SEM_MATCH. To use these functions, you must understand the concepts explained in Spatial Support (page 1-66).

Note:

Throughout this appendix `geomLiteral` is used as a placeholder for both `orageo:WKTLiteral` and `ogc:wktLiteral`, which can be used interchangeably, in format representations and parameter descriptions. (However, `orageo:WKTLiteral` or `ogc:wktLiteral` is used in actual examples.)

This appendix includes the following GeoSPARQL and Oracle-specific functions:

GeoSPARQL functions:

- ogcf:boundary (page B-2)
- ogcf:buffer (page B-3)
- ogcf:convexHull (page B-4)
- ogcf:difference (page B-5)
- ogcf:distance (page B-6)
- ogcf:envelope (page B-7)
- ogcf:getSRID (page B-7)
- ogcf:intersection (page B-8)
- ogcf:relate (page B-9)
- ogcf:sfContains (page B-10)
- ogcf:sfCrosses (page B-11)
- ogcf:sfDisjoint (page B-12)
- ogcf:sfEquals (page B-13)
- ogcf:sfIntersects (page B-14)
- ogcf:sfOverlaps (page B-15)
- ogcf:sfTouches (page B-16)

- ogcf:sfWithin (page B-17)
- ogcf:symDifference (page B-18)

Oracle-specific functions:

- orageo:area (page B-20)
- orageo:buffer (page B-21)
- orageo:centroid (page B-22)
- orageo:convexHull (page B-22)
- orageo:difference (page B-23)
- orageo:distance (page B-24)
- orageo:intersection (page B-25)
- orageo:length (page B-26)
- orageo:mbr (page B-27)
- orageo:nearestNeighbor (page B-28)
- orageo:relate (page B-29)
- orageo:union (page B-31)
- orageo:withinDistance (page B-32)
- orageo:xor (page B-33)

B.1 ogcf:boundary

Format

ogcf:boundary(geom : geomLiteral) : ogc:wktLiteral

Description

Returns a geometry object that is the closure of the boundary of `geom`.

Parameters

geom
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds the boundaries of U.S. Congressional district polygons.

```
SELECT cb
FROM table(sem_match(
'SELECT (ogcf:boundary(?cgeom) AS ?cb)
 WHERE
 { ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom }'
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.2 ogcf:buffer

Format

ogcf:buffer(geom : geomLiteral, radius : xsd:decimal, units : xsd:anyURI) : ogc:wktLiteral

Description

Returns a buffer polygon the specified radius (measured in units) around a geometry.

Parameters

geom
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

radius
Radius value used to define the buffer.

units
Unit of measurement: a URI of the form `<http://xmlns.oracle.com/rdf/geo/uom/{SDO_UNIT}>` (for example, `<http://xmlns.oracle.com/rdf/geo/uom/KM>`). Any SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table will be recognized. See the section about unit of measurement support in *Oracle Spatial and Graph Developer's Guide* for more information about unit of measurement specification.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds the U.S. Congressional district polygons that are within a 100 kilometer buffer around a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
```

```
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (
     ogcf:sfWithin(?cgeom,
       ogcf:buffer("POINT(-71.46444 42.7575)"^^ogc:wktLiteral,
                   100,
                   <http://xmlns.oracle.com/rdf/geo/uom/KM>)) }'
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.3 ogcf:convexHull

Format

ogcf:convexHull(geom : geomLiteral) : ogc:wktLiteral

Description

Returns a polygon geometry that represents the convex hull of geom. (The convex hull is a simple convex polygon that completely encloses the geometry object, using as few straight-line sides as possible to create the smallest polygon that completely encloses the geometry object.)

Parameters

geom
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds the U.S. Congressional district polygons whose convex hull contains a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfContains(ogcf:convexHull(?cgeom),
      "POINT(-71.46444 42.7575)"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
```

```
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.4 ogcf:difference

Format

ogcf:difference(geom1 : geomLiteral, geom2 : geomLiteral) : ogc:wktLiteral

Description

Returns a geometry object that is the topological difference (MINUS operation) of `geom1` and `geom2`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds the U.S. Congressional district polygons whose centroid is within the difference of two specified polygons.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfWithin(orageo:centroid(?cgeom),
        ogcf:difference("Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                         -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral,
                        "Polygon((-83.2 34.3, -83.0 34.3, -83.0 34.5,
                         -83.2 34.5, -83.2 34.3))"^^ogc:wktLiteral))) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.5 ogcf:distance

Format

ogcf:distance(geom1 : geomLiteral, geom2 : geomLiteral, units : xsd:anyURI) : xsd:decimal

Description

Returns the distance in units between the two closest points of `geom1` and `geom2`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

units
Unit of measurement: a URI of the form `<http://xmlns.oracle.com/rdf/geo/uom/{SDO_UNIT}>` (for example, `<http://xmlns.oracle.com/rdf/geo/uom/KM>`). Any SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table will be recognized. See the section about unit of measurement support in *Oracle Spatial and Graph Developer's Guide* for more information about unit of measurement specification.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example orders U.S. Congressional districts based on distance from a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'SELECT ?name ?cdist
 WHERE
 { # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
 }
 ORDER BY ASC(ogcf:distance(?cgeom,
               "POINT(-71.46444 42.7575)"^^ogc:wktLiteral,
               <http://xmlns.oracle.com/rdf/geo/uom/KM>))'
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
```

```
   )
,null, null, null, ' ALLOW_DUP=T '))
ORDER BY sem$rownum;
```

B.6 ogcf:envelope

Format

ogcf:envelope(geom : geomLiteral) : ogc:wktLiteral

Description

Returns the minimum bounding rectangle (MBR) of `geom`, that is, the single rectangle that minimally encloses `geom`.

Parameters

geom
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds the U.S. Congressional district polygons whose minimum bounding rectangle contains a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfContains(ogcf:envelope(?cgeom),
      "POINT(-71.46444 42.7575)"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.7 ogcf:getSRID

Format

ogcf:getSRID(geom : geomLiteral) : xsd:anyURI

Description

Returns the spatial reference system URI for `geom`.

Parameters

geom
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds spatial reference system URIs for U.S. Congressional district polygons.

```
SELECT csrid
FROM table(sem_match(
'SELECT (ogcf:getSRID(?cgeom) AS ?csrid)
 WHERE
 { ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom }'
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.8 ogcf:intersection

Format

ogcf:intersection (geom1 : geomLiteral, geom2 : geomLiteral) : ogc:wktLiteral

Description

Returns a geometry object that is the topological intersection (AND operation) of `geom1` and `geom2`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds the U.S. Congressional district polygons whose centroid is within the intersection of two specified polygons.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfWithin(orageo:centroid(?cgeom),
        ogcf:intersection("Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                           -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral,
                          "Polygon((-83.2 34.3, -83.0 34.3, -83.0 34.5,
                           -83.2 34.5, -83.2 34.3))"^^ogc:wktLiteral))) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.9 ogcf:relate

Format

ogcf:relate(geom1 : geomLiteral, geom2 : geomLiteral, pattern-matrix : xsd:string) : xsd:boolean

Description

Returns `true` if the topological relationship between `geom1` and `geom2` satisfies the specified DE-9IM pattern-matrix. Returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

pattern-matrix
A dimensionally extended 9-intersection model (DE-9IM) intersection pattern string consisting of `T` (true) and `F` (false) values. A DE-9IM pattern string describes the intersections between the interiors, boundaries, and exteriors of two geometries.

Usage Notes

When invoking `ogcf:relate` with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `ogcf:relate` spatial filter).

It is recommended to use a `LEADING(?var)` HINT0 hint when the query involves a restrictive `ogcf:relate` spatial filter on `?var`.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See the OGC Simple Features Specification (OGC 06-103r3) for a detailed description of DE-9IM intersection patterns. See also the OGC GeoSPARQL specification.

Example

The following example finds the U.S. Congressional district that contains a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:relate(?cgeom,
      "POINT(-71.46444 42.7575)"^^ogc:wktLiteral,
      "TTTFFTFFT")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '
));
```

B.10 ogcf:sfContains

Format

ogcf:sfContains(geom1 : geomLiteral, geom2 : geomLiteral) : xsd:boolean

Description

Returns `true` if `geom1` spatially contains `geom2` as defined by the OGC Simple Features specification (OGC 06-103r3). Returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

When invoking this function with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `ogcf:sfContains` spatial filter).

It is recommended to use a `LEADING(?var)` HINT0 hint when the query involves a restrictive `ogcf:sfContains` spatial filter on `?var`.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds U.S. Congressional district polygons that spatially contain a constant polygon.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfContains(?cgeom,
             "Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                       -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.11 ogcf:sfCrosses

Format

ogcf:sfCrosses(geom1 : geomLiteral, geom2 : geomLiteral) : xsd:boolean

Description

Returns `true` if `geom1` spatially crosses `geom2` as defined by the OGC Simple Features specification (OGC 06-103r3). Returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

When invoking this function with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `ogcf:sfCrosses` spatial filter).

It is recommended to use a `LEADING(?var)` HINT0 hint when the query involves a restrictive `ogcf:sfCrosses` spatial filter on `?var`.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds U.S. Congressional district polygons that spatially cross a constant polygon.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfCrosses(?cgeom,
             "Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                       -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.12 ogcf:sfDisjoint

Format

ogcf:fDisjoint(geom1 : geomLiteral, geom2 : geomLiteral) : xsd:boolean

Description

Returns `true` if the two geometries are spatially disjoint as defined by the OGC Simple Features specification (OGC 06-103r3). Returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

The `ogcf:sfDisjoint` filter cannot use a spatial index for evaluation, so performance will probably be much worse than with other simple features spatial functions.

See also the OGC GeoSPARQL specification.

Example

The following example finds U.S. Congressional district polygons that are spatially disjoint from a constant polygon.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfDisjoint(?cgeom,
             "Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                       -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.13 ogcf:sfEquals

Format

ogcf:sfEquals(geom1 : geomLiteral, geom2 : geomLiteral) : xsd:boolean

Description

Returns `true` if the two geometries are spatially equal as defined by the OGC Simple Features specification (OGC 06-103r3). Returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

When invoking this function with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `ogcf:sfEquals` spatial filter).

It is recommended to use a `LEADING(?var)` HINT0 hint when the query involves a restrictive `ogcf:sfEquals` spatial filter on `?var`.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds U.S. Congressional district polygons that are spatially equal to a constant polygon.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfEquals(?cgeom,
             "Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                       -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.14 ogcf:sfIntersects

Format

ogcf:sfIntersects(geom1 : geomLiteral, geom2 : geomLiteral) : xsd:boolean

Description

Returns `true` if the two geometries are *not* disjoint as defined by the OGC Simple Features specification (OGC 06-103r3). Returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

When invoking this function with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `ogcf:sfIntersects` spatial filter).

It is recommended to use a `LEADING(?var)` HINT0 hint when the query involves a restrictive `ogcf:sfIntersects` spatial filter on `?var`.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds U.S. Congressional district polygons that intersect a constant polygon.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfIntersects(?cgeom,
             "Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                       -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.15 ogcf:sfOverlaps

Format

ogcf:sfOverlaps(geom1 : geomLiteral, geom2 : geomLiteral) : xsd:boolean

Description

Returns `true` if `geom1` spatially overlaps `geom2` as defined by the OGC Simple Features specification (OGC 06-103r3). Returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

When invoking this function with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `ogcf:sfOverlaps` spatial filter).

It is recommended to use a `LEADING(?var)` HINT0 hint when the query involves a restrictive `ogcf:sfOverlaps` spatial filter on `?var`.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds U.S. Congressional district polygons that spatially overlap a constant polygon.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfOverlaps(?cgeom,
             "Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                       -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.16 ogcf:sfTouches

Format

ogcf:sfTouches(geom1 : geomLiteral, geom2 : geomLiteral) : xsd:boolean

Description

Returns `true` if the two geometries spatially touch as defined by the OGC Simple Features specification (OGC 06-103r3). Returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

When invoking this function with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `ogcf:sfTouches` spatial filter).

It is recommended to use a `LEADING(?var)` HINT0 hint when the query involves a restrictive `ogcf:sfTouches` spatial filter on `?var`.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds U.S. Congressional district polygons that spatially touch a constant polygon.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfTouches(?cgeom,
             "Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                       -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.17 ogcf:sfWithin

Format

ogcf:sfWithin(geom1 : geomLiteral, geom2 : geomLiteral) : xsd:boolean

Description

Returns `true` if `geom1` is spatially within `geom2` as defined by the OGC Simple Features specification (OGC 06-103r3). Returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

When invoking this function with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `ogcf:sfWithin` spatial filter).

It is recommended to use a `LEADING(?var)` HINT0 hint when the query involves a restrictive `ogcf:sfWithin` spatial filter on `?var`.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds U.S. Congressional district polygons that are spatially within a constant polygon.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfWithin(?cgeom,
             "Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                       -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral)) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.18 ogcf:symDifference

Format

ogcf:symDifference(geom1 : geomLiteral, geom2 : geomLiteral) : ogc:wktLiteral

Description

Returns a geometry object that is the topological symmetric difference (XOR operation) of `geom1` and `geom2`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds the U.S. Congressional district polygons that are within a 100 kilometer buffer around a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
```

```
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfWithin(orageo:centroid(?cgeom),
      ogcf:symDifference("Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                          -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral,
                          "Polygon((-83.2 34.3, -83.0 34.3, -83.0 34.5,
                          -83.2 34.5, -83.2 34.3))"^^ogc:wktLiteral))) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.19 ogcf:union

Format

ogcf:union(geom1 : geomLiteral, geom2 : geomLiteral) : ogc:wktLiteral

Description

Returns a geometry object that is the topological union (OR operation) of `geom1` and `geom2`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the OGC GeoSPARQL specification.

Example

The following example finds the U.S. Congressional district polygons whose centroid is within the union of two specified polygons.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (ogcf:sfWithin(orageo:centroid(?cgeom),
        ogcf:union("Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                    -83.6 34.5, -83.6 34.1))"^^ogc:wktLiteral,
                   "Polygon((-83.2 34.3, -83.0 34.3, -83.0 34.5,
                    -83.2 34.5, -83.2 34.3))"^^ogc:wktLiteral))) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
```

```
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/')
   )
,null, null, null, ' ALLOW_DUP=T '));
```

B.20 orageo:area

Format

orageo:area(geom1 : geomLiteral, unit : Literal) : xsd:decimal

Description

Returns the area of `geom1` in terms of the specified unit of measure.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

unit
Unit of measurement: a quoted string with an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table (for example, `"unit=SQ_KM"`). See the section about unit of measurement support in *Oracle Spatial and Graph Developer's Guide* for more information about unit of measurement specification.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_AREA function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons with areas greater than 10,000 square kilometers.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:area(?cgeom, "unit=SQ_KM") > 10000) }'
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.21 orageo:buffer

Format

orageo:buffer(geom1 : geomLiteral, distance : xsd:decimal, unit : Literal) : geomLiteral

Description

Returns a buffer polygon at a specified distance around or inside a geometry.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

distance
Distance value. Distance value. If the value is positive, the buffer is generated around `geom1`; if the value is negative (valid only for polygons), the buffer is generated inside `geom1`.

unit
Unit of measurement: a quoted string with an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table (for example, `"unit=KM"`). See the section about unit of measurement support in *Oracle Spatial and Graph Developer's Guide* for more information about unit of measurement specification.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_BUFFER function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons that are completely inside a 100 kilometer buffer around a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (
     orageo:relate(?cgeom,
       orageo:buffer("POINT(-71.46444 42.7575)"^^orageo:WKTLiteral,
                     100, "unit=KM"),
      "mask=inside")) }'
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.22 orageo:centroid

Format

orageo:centroid(geom1 : geomLiteral) : geomLiteral

Description

Returns a point geometry that is the centroid of `geom1`. (The centroid is also known as the "center of gravity.")

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

For an input geometry consisting of multiple objects, the result is weighted by the area of each polygon in the geometry objects. If the geometry objects are a mixture of polygons and points, the points are not used in the calculation of the centroid. If the geometry objects are all points, the points have equal weight.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_CENTROID function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons with centroids within 200 kilometers of a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:withinDistance(orageo:centroid(?cgeom),
      "POINT(-71.46444 42.7575)"^^orageo:WKTLiteral,
      "distance=200 unit=KM")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.23 orageo:convexHull

Format

orageo:convexHull(geom1 : geomLiteral) : geomLiteral

Description

Returns a polygon-type object that represents the convex hull of `geom1`. (The **convex hull** is a simple convex polygon that completely encloses the geometry object, using as few straight-line sides as possible to create the smallest polygon that completely encloses the geometry object.)

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

A convex hull is a convenient way to get an approximation of a complex geometry object.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_CONVEX_HULL function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons whose convex hull contains a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:relate(orageo:convexHull(?cgeom),
      "POINT(-71.46444 42.7575)"^^orageo:WKTLiteral,
      "mask=contains")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.24 orageo:difference

Format

orageo:difference(geom1 : geomLiteral, geom2 : geomLiteral) : geomLiteral

Description

Returns a geometry object that is the topological difference (MINUS operation) of `geom1` and `geom2`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_UNION function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons whose centroid is inside the difference of two specified polygons.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:relate(orageo:centroid(?cgeom),
      orageo:difference("Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                         -83.6 34.5, -83.6 34.1))"^^orageo:WKTLiteral,
                        "Polygon((-83.2 34.3, -83.0 34.3, -83.0 34.5,
                         -83.2 34.5, -83.2 34.3))"^^orageo:WKTLiteral),
      "mask=inside")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.25 orageo:distance

Format

orageo:distance(geom1 : geomLiteral, geom2 : geomLiteral, unit : Literal) : xsd:decimal

Description

Returns the distance between the nearest pair of points or segments of `geom1` and `geom2` in terms of the specified unit of measure.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

unit
Unit of measurement: a quoted string with an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table (for example, `"unit=KM"`). See the section about unit of measurement support in *Oracle Spatial and Graph Developer's Guide* for more information about unit of measurement specification.

Usage Notes

Use orageo:withinDistance `(page B-32)` instead of `orageo:distance` whenever possible, because orageo:withinDistance (page B-32) has a more efficient index-based implementation.

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_DISTANCE function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the ten nearest U.S. Congressional districts to a specified point and orders them by distance from the point.

```
SELECT name, cdist
FROM table(sem_match(
'SELECT ?name ?cdist
 WHERE
 { # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:nearestNeighbor(?cgeom,
      "POINT(-71.46444 42.7575)"^^orageo:WKTLiteral,
      "sdo_num_res=10")) }
 ORDER BY ASC(orageo:distance(?cgeom,
                "POINT(-71.46444 42.7575)"^^orageo:WKTLiteral,
                "unit=KM"))'
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '))
ORDER BY sem$rownum;
```

B.26 orageo:intersection

Format

orageo:intersection(geom1 : geomLiteral, geom2 : geomLiteral) : geomLiteral

Description

Returns a geometry object that is the topological intersection (AND operation) of `geom1` and `geom2`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_INTERSECTION function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons whose centroid is inside the intersection of two specified polygons.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:relate(orageo:centroid(?cgeom),
      orageo:intersection("Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                           -83.6 34.5, -83.6 34.1))"^^orageo:WKTLiteral,
                          "Polygon((-83.2 34.3, -83.0 34.3, -83.0 34.5,
                           -83.2 34.5, -83.2 34.3))"^^orageo:WKTLiteral),
      "mask=inside")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.27 orageo:length

Format

orageo:length(geom1 : geomLiteral, unit : Literal) : xsd:decimal

Description

Returns the length or perimeter of `geom1` in terms of the specified unit of measure.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

unit
Unit of measurement: a quoted string with an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table (for example, `"unit=KM"`). See the section about

unit of measurement support in *Oracle Spatial and Graph Developer's Guide* for more information about unit of measurement specification.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_LENGTH function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons with lengths (perimeters) greater than 1000 kilometers.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:legnth(?cgeom, "unit=KM") > 1000) }'
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.28 orageo:mbr

Format

orageo:mbr(geom1 : geomLiteral) : geomLiteral

Description

Returns the minimum bounding rectangle of `geom1`, that is, the single rectangle that minimally encloses `geom1`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_MBR function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons whose minimum bounding rectangle contains a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:relate(orageo:mbr(?cgeom),
      "POINT(-71.46444 42.7575)"^^orageo:WKTLiteral,
      "mask=contains")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.29 orageo:nearestNeighbor

Format

orageo:nearestNeighbor(geom1: geomLiteral, geom2 : geomLiteral, param : Literal) : xsd:boolean

Description

Returns `true` if `geom1` is a nearest neighbor of `geom2`, where the size of the nearest neighbors set is specified by `param`; returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

param
Determines the behavior of the operator. See the Usage Notes for the available keyword-value pairs.

Usage Notes

In the param parameter, the available keyword-value pairs are:

- `distance=`*n* specifies the maximum allowable distance for the nearest neighbor search.
- `sdo_num_res=`*n* specifies the size of the set for the nearest neighbor search.
- `unit=`*unit* specifies the unit of measurement to use with `distance` value. If you do not specify a value, the unit of measurement associated with the data is used.

`geom1` must be a local variable (that is, a variable that appears in the basic graph pattern that contains the `orageo:nearestNeighbor` spatial filter).

It is a good idea to use a `'LEADING(?var)' HINT0` hint when your query involves a restrictive `orageo:relate` spatial filter on `?var.`

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_NN operator in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the ten nearest U.S. Congressional districts to a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:nearestNeighbor(?cgeom,
      "POINT(-71.46444 42.7575)"^^orageo:WKTLiteral,
      "sdo_num_res=10")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.30 orageo:relate

Format

orageo:relate(geom1: geomLiteral, geom2 : geomLiteral, param : Literal) : xsd:boolean

Description

Returns `true` if `geom1` and `geom2` satisfy the topological spatial relation specified by the `param` parameter; returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

param
Specifies a list of mask relationships to check. See the list of keywords in the Usage Notes.

Usage Notes

The following `param` values (mask relationships) can be tested:

- ANYINTERACT: Returns TRUE if the objects are not disjoint.
- CONTAINS: Returns CONTAINS if the second object is entirely within the first object and the object boundaries do not touch; otherwise, returns FALSE.

- COVEREDBY: Returns COVEREDBY if the first object is entirely within the second object and the object boundaries touch at one or more points; otherwise, returns FALSE.
- COVERS: Returns COVERS if the second object is entirely within the first object and the boundaries touch in one or more places; otherwise, returns FALSE.
- DISJOINT: Returns DISJOINT if the objects have no common boundary or interior points; otherwise, returns FALSE.
- EQUAL: Returns EQUAL if the objects share every point of their boundaries and interior, including any holes in the objects; otherwise, returns FALSE.
- INSIDE: Returns INSIDE if the first object is entirely within the second object and the object boundaries do not touch; otherwise, returns FALSE.
- ON: Returns ON if the boundary and interior of a line (the first object) is completely on the boundary of a polygon (the second object); otherwise, returns FALSE.
- OVERLAPBDYDISJOINT: Returns OVERLAPBDYDISJOINT if the objects overlap, but their boundaries do not interact; otherwise, returns FALSE.
- OVERLAPBDYINTERSECT: Returns OVERLAPBDYINTERSECT if the objects overlap, and their boundaries intersect in one or more places; otherwise, returns FALSE.
- TOUCH: Returns TOUCH if the two objects share a common boundary point, but no interior points; otherwise, returns FALSE.

Values for `param` can be combined using the logical Boolean operator OR. For example, 'INSIDE + TOUCH' returns `INSIDE+TOUCH` if the relationship between the geometries is INSIDE or TOUCH or both INSIDE and TOUCH; it returns `FALSE` if the relationship between the geometries is neither INSIDE nor TOUCH.

When invoking `orageo:relate` with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `orageo:relate` spatial filter).

It is a good idea to use a `'LEADING(?var)' HINT0` hint when your query involves a restrictive `orageo:relate` spatial filter on `?var.`

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_RELATE operator in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district that contains a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
```

```
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:relate(?cgeom,
      "POINT(-71.46444 42.7575)"^^orageo:WKTLiteral,
      "mask=contains")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '
));
```

B.31 orageo:union

Format

orageo:union(geom1 : geomLiteral, geom2 : geomLiteral) : geomLiteral

Description

Returns a geometry object that is the topological union (OR operation) of `geom1` and `geom2`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_UN ION function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons whose centroid is inside the union of two specified polygons.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:relate(orageo:centroid(?cgeom),
      orageo:union("Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
                    -83.6 34.5, -83.6 34.1))"^^orageo:WKTLiteral,
                   "Polygon((-83.2 34.3, -83.0 34.3, -83.0 34.5,
                    -83.2 34.5, -83.2 34.3))"^^orageo:WKTLiteral),
      "mask=inside")) } '
,sem_models('gov_all_vm'), null,
```

```
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.32 orageo:withinDistance

Format

orageo:withinDistance(geom1 : geomLiteral, geom2 : geomLiteral, distance : xsd:decimal, unit : Literal) : xsd:boolean

Description

Returns `true` if the distance between `geom1` and `geom2` is less than or equal to `distance` when measured in `unit`; returns `false` otherwise.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

distance
Distance value.

unit
Unit of measurement: a quoted string with an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table (for example, `"unit=KM"`). See the section about unit of measurement support in *Oracle Spatial and Graph Developer's Guide* for more information about unit of measurement specification.

Usage Notes

When invoking this function with a query variable and a constant geometry, always use the query variable as the first parameter and the constant geometry as the second parameter.

For best performance, `geom1` should be a local variable (that is, a variable that appears in the basic graph pattern that contains the `orageo:withinDistance` spatial filter).

It is a good idea to use a `'LEADING(?var)' HINT0` hint when your query involves a restrictive `orageo:withinDistance` spatial filter on `?var.`

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_WITHIN_DISTANCE operator in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional districts that are within 100 kilometers of a specified point.

```
SELECT name, cdist
FROM table(sem_match(
'{ # HINT0={LEADING(?cgeom)}
   ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:withinDistance(?cgeom,
      "POINT(-71.46444 42.7575)"^^orageo:WKTLiteral,
      100, "KM")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

B.33 orageo:xor

Format

orageo:xor(geom1 : geomLiteral, geom2 : geomLiteral) : geomLiteral

Description

Returns a geometry object that is the topological symmetric difference (XOR operation) of `geom1` and `geom2`.

Parameters

geom1
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

geom2
Geometry object. Specified as a query variable or a constant `geomLiteral` value.

Usage Notes

See Spatial Support (page 1-66) for information about representing , indexing, and querying spatial data in RDF.

See also the SDO_GEOM.SDO_XOR function in *Oracle Spatial and Graph Developer's Guide*.

Example

The following example finds the U.S. Congressional district polygons whose centroid is inside the symmetric difference of two specified polygons.

```
SELECT name, cdist
FROM table(sem_match(
'{ ?person usgovt:name ?name .
   ?person pol:hasRole ?role .
   ?role pol:forOffice ?office .
   ?office pol:represents ?cdist .
   ?cdist orageo:hasExactGeometry ?cgeom
   FILTER (orageo:relate(orageo:centroid(?cgeom),
      orageo:xor("Polygon((-83.6 34.1, -83.2 34.1, -83.2 34.5,
```

```
                  -83.6 34.5, -83.6 34.1))"^^orageo:WKTLiteral,
                 "Polygon((-83.2 34.3, -83.0 34.3, -83.0 34.5,
                  -83.2 34.5, -83.2 34.3))"^^orageo:WKTLiteral),
      "mask=inside")) } '
,sem_models('gov_all_vm'), null,
,sem_aliases(
   sem_alias('usgovt','http://www.rdfabout.com/rdf/schema/usgovt/'),
   sem_alias('pol','http://www.rdfabout.com/rdf/schema/politico/'))
,null, null, null, ' ALLOW_DUP=T '));
```

C

Virtual Private Database Support in RDF Semantic Graph

Note:

Effective with Oracle Database Release 12*c* (12.1), Virtual Private Database (VPD) support in RDF Semantic Graph is deprecated for providing fine-grained access control, and will be removed in an upcoming major release.

You should not develop new RDF Semantic Graph applications that depend on VPD, and you should transition existing RDF Semantic Graph applications that depend on VPD to use Oracle Label Security (OLS) instead. (Support for OLS is described in Fine-Grained Access Control for RDF Data (page 5-1).)

For more information, see My Oracle Support Note 1468273.1.

Virtual Private Database (VPD) for RDF data allows security administrators to define policies that conditionally restrict a user's access to triples that involve instances of a specific RDF class or property. Using VPD, the data stored in the RDF models is classified using its metadata and each user query is rewritten to include context-dependent data access constraints that enforce access restrictions.

For information about using VPD, see *Oracle Database Security Guide*.

Some factors to consider in choosing whether use VPD or OLS with RDF data include the following:

- OLS, when enabled for RDF data, is enforced at the network level, while VPD can be enforced for individual RDF models.
- You cannot use both VPD and OLS for RDF data at the same time in an Oracle instance.

The application programming interface (API) for implementing VPD or OLS with semantic data is provided in the SEM_RDFSA PL/SQL package. SEM_RDFSA Package Subprograms (page 15-1) provides reference information about the programs in the SEM_RDFSA package.

VPD and OLS support for RDF data is included in the RDF Semantic Graph support for Oracle Database. (For information about enabling, downgrading, or removing RDF Semantic Graph support, see Enabling_ Downgrading_ or Removing RDF Semantic Graph Support (page A-1).)

VPD for RDF cannot be used in a CDB (multitenant container database) environment.

C.1 Virtual Private Database (VPD) for RDF Data

The Virtual Private Database (VPD) feature is a row-level security mechanism that restricts access to specific rows in a relational table or view using a security policy and an application context. The security policy includes a policy function that dynamically generates predicates that are enforced for each row returned for the user query. The security predicates returned by the policy function associated with a table are typically expressed using the columns in the table and are thus dependent on the table metadata. Effectively, the security predicates ensure that the rows returned for a user query satisfy additional conditions that are applied on the contents of the row.

When the relational data is mapped to RDF, the data stored in a specific relational table represent triples describing instances of a specific RDF class. In this representation, the columns in the relational table map to RDF properties that are used to describe a resource. This mapping may be further extended to the application of VPD policies.

A VPD policy applied to RDF data restricts users' access to instances of a specific RDF class or property by applying predicates, in the form of graph patterns and filter conditions, on the instance data. For example, a VPD policy may be defined to restrict access to instances of a `Contract` RDF class only to the users belonging to a specific department. Furthermore, access to the `hasContractValue` property for a resource identified as instance of the `Contract` RDF class may be restricted only to the manager of the contract. VPD support for RDF data allows security conditions or data access constraints to be associated with RDF classes and properties, so that access to corresponding instance data is restricted.

A data access constraint associated with an RDF class or property specifies a graph query pattern that must be enforced for all corresponding data instances that are returned as the query result. For example, a SPARQL query pattern to find the due dates for all instances of a `Contract` class, `{?contract :hasDueDate ?due}`, may activate a data access constraint that ensures that the information returned pertains to contracts belonging to a specific department. This is achieved by logically rewriting the user's graph query pattern to include additional graph patterns, as shown in the following example:

```
{ ?contract   :hasDueDate  ?due .
  ?contract   :drivenBy    dept:Dept1 }
```

Furthermore, the values bound into the rewritten graph query pattern may make use of session context to enforce dynamic access restrictions. In the following example, the `sys_context` function in the object position of the triple pattern binds the appropriate department value based on the session context:

```
{ ?contract   :hasDueDate   ?due .
  ?contract   :drivenBy
            "sys_context('sa$appctx','user_dept'}"^^orardf:instruction }
```

In a relational data model, the metadata, in the form of table definition, always exists along with the data (the rows stored in the table); thus, the VPD policies defined using the metadata are well formed and the security conditions are generated using a procedural logic in the policy function.

However, the RDF data model allows data with no accompanying metadata, and therefore the class information for instance data may not always be available for a given RDF graph. For example, in an RDF graph a resource known to be a contract might not accompany a triple that asserts that the resource is an instance of `Contract`

class. Usually such triples can be inferred using available domain and range specifications for the properties describing the resource.

Similarly, a VPD policy relies on the properties' domain and range specifications for deriving the class information for the instance data and for enforcing appropriate data access constraints. However, to avoid runtime dependencies on the user data, a VPD policy maintains the minimal metadata required to derive the class information in its dictionary, separate from the asserted and inferred triples. This also ensures that the metadata maintained by a VPD policy is complete even when some necessary information is missing from the asserted triples and that a VPD policy, with its data access constraints and the metadata, is self-contained and portable with no external dependencies.

A VPD policy with specific data access constraints and RDF metadata specifications can be used to enforce access restrictions for the data stored in an RDF model. Each SPARQL query issued on the model is analyzed to deduce the class information for the resources accessed in the query, and appropriate data access constraints are applied. To facilitate the compile-time analysis and derivation of class information for instance data, a graph query pattern with an unbound predicate is restricted when a VPD policy is in effect. For example, a graph pattern of the following form, anywhere in a SPARQL query pattern, raises an exception when any underlying model has a VPD policy:

```
{ <contract:projectHLS>  ?pred  ?obj }
```

VPD policies are only enforced for SEM_MATCH queries expressed in SPARQL syntax. All other forms of data access (such as classic syntax for graph pattern or direct query on the model view) are not permitted.

C.1.1 VPD Policy for RDF Data

A VPD policy for RDF data is a named dictionary entity that can be used to enforce access restrictions for the data stored in one or more RDF models. A VPD policy defined for RDF data has unique characteristics, and it cannot be reused to enforce security policies for relational data. An RDF-VPD policy defined in the database includes the following:

- The RDF Schema statements or metadata necessary for deriving class information for the data referenced in a SPARQL user query
- The data access constraints that enforce access restrictions for the instance data
- Application context that allows conditional inclusions of groups of data access constraints based on the runtime environment

An RDF-VPD policy is defined, owned, and maintained by a user with a security administrator role in an organization. This user must have at least EXECUTE privileges on the SYS.DBMS_RLS package. The owner of an RDF-VPD policy can maintain the metadata associated with the policy, define new data access constraints, and apply the policy to one or more RDF models.

A SPARQL query issued on an RDF model with a VPD policy is analyzed, and zero or more data access constraints defined in the policy are enforced such that the data instances that are returned as the query result also satisfy these constraints. The exact data access constraints enforced for a user query vary, based on the resources referenced in the query and the application context. For example, a policy that restricts a manager's access to the `hasContractValue` property may be relaxed for a user with the Vice President role.

Based on the role of the user, as captured in the application context, specific constraints to be applied are determined at runtime. To facilitate this dynamic inclusion of subsets of constraints defined in a VPD policy, the data access constraints are arranged into named groups that can be activated and deactivated based on the application context. During query analysis, only the constraints defined in the active groups are considered for enforcement.

The constraint groups within a VPD policy are managed using an application context and its package implementation. Each VPD policy can specify the namespace for a context created with the CREATE CONTEXT command. Each attribute associated with the context is treated as the name of a constraint group that can be activated by initializing its value to 1. For example, setting the value for MANAGER attribute of the context associated with a VPD policy to 1 will activate the constraints associated with MANAGER group for the user session. The logic that initializes specific constraint groups based on the user context is typically embedded in the package associated with the context type. The following example shows an excerpt from a sample implementation for one such package:

```
CREATE CONTEXT contracts_constr_ctx using sec_admin.contracts_ctx_pack;

begin
  -- create the VPD policy with a context --
  sem_rdfsa.create_vpd_policy(policy_name    => 'CONTRACTS_POLICY',
                              policy_context => 'contracts_constr_ctx');
end;
/

create or replace package sec_admin.contracts_ctx_pack as
  procedure init_constr_groups;
end;
/

create or replace package body sec_admin.contracts_ctx_pack as
  procedure init_contr_groups is
    hrdata EmpRole%rowtype;
  begin
    -- specific users with FULL access to the data associated with
    -- the policy --
    if (sys_context('userenv', 'session_user') = 'RDF_ADMIN') then
      dbms_session.set_context('contracts_constr_ctx',
                               sem_rdfsa.VPD_FULL_ACCESS, 1);
      return;
    end if;

    SELECT * into hrdata FROM EmpRole WHERE guid =
                          sys_context('userenv','session_user');

    if (hrdata.emprole = 'VP') then
      -- if the user logged in has VP role, activate the constraint
      -- group named VP and keep all other groups inactive.
      dbms_session.set_context('contracts_constr_ctx','VP', '1');
    elsif (hrdata.emprole = 'MANAGER') then
      dbms_session.set_context('contracts_constr_ctx', 'MANAGER', '1');
    elsif ...
      ...
    else
      raise_application_error(-20010, 'unknown user role');
    end if;

  exception when others then
```

```
    -- enforce constraints for all groups --
    dbms_session.clear_all_context('contracts_constr_ctx');
  end init_contr_groups;
end;
/
```

By default, when a namespace is not associated with an RDF-VPD policy or when a specific constraint group is not activated in a session, all the constraints defined in the policy are active and they are enforced for each user query. However, when a specific constraint group is activated by setting the corresponding namespace-attribute value to 1, only the constraints belonging to the group and any other constraints that are not associated with any group are enforced. For a given session, one or more constraint groups may be activated, in which case all the applicable constraints are enforced conjunctively.

At the time of creation, the data access constraints defined in a RDF-VPD policy may specify the name of a constraint group (explained in Data Access Constraints (page C-7)). Within a database session, appropriate groups of constraints are activated based on the session context set by the context package. For all subsequent SPARQL queries in the database session, the constraints belonging to the active groups are consulted for enforcing appropriate security policies.

Maintenance operations on an RDF model with a VPD policy require unconditional access to data in the model. These operations include creation of an entailment using at least one VPD protected model, and load or data manipulation operations. You can grant unconditional access to the data stored in an RDF model by initializing a reserved attribute for the namespace associated with the VPD policy. The reserved attribute is defined by the package constant `sem_rdfsa.VPD_FULL_ACCESS`, and the context package implementation shown in the preceding example grants FULL access to the RDF_ADMIN user.

DML operations on the application table are not validated for VPD constraint violations, so only a user with FULL access to the corresponding model can add or modify existing triples.

You can use the SEM_MATCH operator to query an RDF model with a VPD policy in a standard SQL query, and to perform a multi-model query on a combination of VPD-enabled models and models with no VPD policy. However, when more than one model in a multi-model query is VPD-enabled, they must all be associated with the same VPD policy. A VPD policy associated with an RDF model is automatically extended to any data inferred from the model. When multiple RDF models are specified during inference, all VPD-enabled models within the set should use the same VPD policy.

C.1.2 RDF Metadata for Enforcing VPD Policies

The types of RDF metadata used to enforce VPD policies include the following:

- Domain and range information for the properties used in the graph
- Subclass relationships in the graph
- Subproperty relationships in the graph
- Equivalent properties in the graph

The RDF metadata associated with a VPD policy is specified as one or more RDF Schema statements using one of the following property URIs:

- `http://www.w3.org/2000/01/rdf-schema#domain`
- `http://www.w3.org/2000/01/rdf-schema#range`
- `http://www.w3.org/2000/01/rdf-schema#subClassOf`
- `http://www.w3.org/2000/01/rdf-schema#subPropertyOf`
- `http://www.w3.org/2002/07/owl#equivalentProperty`

For example, the following RDF Schema statement associated with `contracts_policy` asserts that the domain of the `hasContractValue` property is a `Contract` class. Note that range specification for the predicates can be skipped if they are not relevant or if they are of literal type

```
begin
  sem_rdfsa.maint_vpd_metadata(
        policy_name => 'contracts_policy',
        t_subject   => '<http://www.myorg.com/pred/hasContractValue>',
        t_predicate => '<http://www.w3.org/2000/01/rdf-schema#domain>',
        t_object    => '<http://www.myorg.com/classes/Contract>');
end;
/
```

An RDF-VPD policy maintains its metadata separate from the asserted and inferred triples. You can derive this metadata programmatically from the RDF models and the corresponding entailments. For example, if the domain and range information for the properties and subclass and subproperty relationships are already established in the asserted or inferred triples, you can use a SQL query on the underlying model views to populate the metadata for an RDF-VPD policy.

The domain and range information for the properties aid the query analysis in determining the RDF class type for the terms and unbound variables referenced in the query. This information is further used to enforce appropriate data access constraints on the data accessed by the query. The metadata relating to the subclass property is used to ensure that a data access constraint defined for a specific class in a class hierarchy is automatically enforced for all its subclasses. Similarly, the subproperty specification in a VPD policy is used to enforce any constraints associated with a property to all its subproperties.

The RDF Schema statements associated with a VPD policy are not used to infer additional statements, and the security administrator should ensure that the metadata captured in a VPD policy is complete by cross checking it with inferred data. For example, a subproperty schema statement does not automatically infer the domain and range information for the property based on the domain and range specified for the super-property.

Certain owl and rdfs properties in the asserted triples, when left unchecked, may be used to infer data that may be used to circumvent the VPD policies. For example, when the new property is defined as a super-property of a property that has a specific data access constraint, the inferred data may duplicate all instances of the subproperty using the super-property. Unless the VPD policy explicitly defines access constraints for the super-property, the inferred data may be used to circumvent the access restrictions.

The ability to infer new data is only granted to users with FULL access, and such users should ensure that the metadata associated with the VPD policy is complete in light of newly inferred data. Specifically, the metadata associated with the VPD policy should be maintained if some new `rdfs:subClassOf`, `rdfs:superClassOf`, `rdfs:subPropertyOf`, `rdfs:superPropertyOf`, or `owl:equivalentProperty`

assertions are generated during inference. Also, any new properties introduced by the rulebases used for inference may need domain and range specifications, as well as data access constraints, if they are associated with some sensitive information.

In a VPD policy, a property can be declared to be equivalent to another property so that the domain and range information, as well as any constraints defined for the original property, are automatically duplicated for the equivalent property. However, within a VPD policy, additional metadata or data access constraints cannot be directly assigned to the property declared to be an equivalent of another property.

C.1.3 Data Access Constraints

The data access constraints associated with a VPD policy fall into two general categories, based on the types of access restrictions that they enforce:

- Those that restrict access to instances of specific RDF classes
- Those that restrict to assertions using specific RDF properties

The access restrictions are enforced conditionally, based on the application context and the characteristics of the resources being accessed in a SPARQL query. Data access constraints restrict access to instances of an RDF class or property using some properties associated with the resource. For example, access to a resource that is a member of the `Contract` class may be restricted only to the users who work on the contract, identified using the `hasMember` property associated with the resource. Similarly, access to the `hasContractValue` property for a resource may be restricted to a user identified as the manager of the contract using `hasManager` property associated with the same resource.

Each data access constraint is expressed using two graph patterns identified as a match pattern and an apply pattern. The **match pattern** of a constraint determines the type of access restriction it enforces and binds one or more variables to the corresponding data instances accessed in the user query. For example, the following match pattern is defined for instances of the `Contract` class, and it binds a variable to all such instances accessed through a SPARQL query:

```
{ ?contract  rdf:type  <http://www.myorg.com/classes/Contract> }
```

Similarly, a match pattern for a constraint involving an RDF property matches the instances of the property accessed in a SPARQL query, and binds two variables to the resources in the subject and object position of such instances. For example, the match pattern for a constraint on the `hasContractValue` property is defined as follows:

```
{ ?contract  <http://www.myorg.com/pred/hasContractValue>  ?cvalue }
```

The **apply pattern** of a data access constraint defines additional graph patterns to be applied on the resources that match the match pattern before they can be used to construct the query results. One or more variables defined in the match pattern of a data access constraint are used in the corresponding apply pattern to enforce the access restrictions on the identified resources. For example, the following match pattern and apply pattern combination ensures that the `hasContractValue` of a contract can be accessed only if `Andy` is the manager of the contract being accessed.:

```
Match:  { ?contract  pred:hasContractValue  ?cvalue  }
Apply:  { ?contract  pred:hasManager        emp:Andy }
```

A data access constraint with its match and apply patterns expressed in SPARQL syntax can be added to a VPD policy to enforce access restrictions on the data stored in RDF models that are associated with the VPD policy. The following example, which

adds a constraint to the VPD policy, assumes that the VPD policy is defined with appropriate namespace map for the `pred` and `emp` namespace prefixes. (To associate a namespace map with a VPD policy, use the SEM_RDFSA.CREATE_VPD_POLICY (page 15-6) procedure.)

```
begin
  sem_rdfsa.add_vpd_constraint(
          policy_name   => 'contracts_policy',
          constr_name   => 'andy_constraint_1',
          match_pattern => '{?contract  pred:hasContractValue ?cvalue }',
          apply_pattern => '{?contract  pred:hasManager       emp:Andy }',
          constr_group  => 'andy');
end;
/
```

The ability to arrange data access constraints into groups could ensure that the previous constraint is applied only for the sessions associated with `Andy`. However, to avoid proliferation of structurally similar constraints for each user, you can define a common constraint that uses the application context in the object position of the apply graph patterns, as shown in the following example:

```
begin
  sem_rdfsa.add_vpd_constraint(
          policy_name   => 'contracts_policy',
          constr_name   => 'manager_constraint_1',
          match_pattern => '{?contract  pred:hasContractValue ?cvalue }',
          apply_pattern => '{?contract  pred:hasManager
             "sys_context(''sa$appctx'',''app_user_uri'')"^^orardf:instruction }',
          constr_group  => 'manager');
end;
/
```

In the preceding example. the data access constraint, defined within the `manager` constraint group, can be activated for all sessions involving users with a manager role. In this case, the secure application context can be programmed to initialize the attribute `app_user_uri` of the `sa$appctx` namespace with the URI for the user logged in. For example, when user `Andy` logs into the application, the `app_user_uri` attribute can be initialized to <http://www.myorg.com/employee/Andy>, in which case the constraint will ensure that user `Andy` can view the value for a contract only if user `Andy` manages the contract. Generally, the `sys_context` function can be used in the object position of any graph pattern to allow dynamic URIs or literal values to be bound at the time of query execution. Note that if the context is not initialized properly, the preceding constraint will fail for all data instances and effectively restrict the user from accessing any data.

A SPARQL query issued on an RDF model with a VPD policy is analyzed using the match patterns of all the active data access constraints that are defined in the policy. In the next example, the SPARQL query refers to the `hasContractValue` property, thereby enforcing the constraint if the group is active. Logically, the enforcement of a constraint is equivalent to rewriting the original SPARQL graph pattern to include the apply patterns for all the relevant constraints, using appropriate variables and terms from the user query. With the previous access restriction on the `hasContractValue` property, the following SPARQL graph pattern passed to a SEM_MATCH operator is logically rewritten as shown in the following example:

```
Query:
{ ?contr  pred:drivenBy         ?dept .
  ?contr  pred:hasContractValue ?val }
```

```
Rewritten query:
{ ?contr  pred:drivenBy         ?dept .
  ?contr  pred:hasContractValue ?val .
  ?contr  pred:hasManager
               "sys_context('sa$appctx','app_user_uri'}"^^orardf:instruction }
```

When the match pattern of a data access constraint on an RDF property matches the pattern being accessed in a user query, the equivalent VPD-enforced query appends the corresponding apply patterns to the SPARQL query using the variables and terms appearing in the matched pattern. When a SPARQL query has nested graph patterns, the data access constraints are applied to appropriate basic query graph pattern block. In the following example, the `hasContractValue` property is referenced in the `OPTIONAL` graph pattern, and therefore the corresponding apply pattern is enforced just for this block of the graph pattern.

```
Query:
{ ?contr  pred:drivenBy         ?dept .
   OPTIONAL { ?contr  pred:hasContractValue ?val } }

Rewritten query:
{ ?contr  pred:drivenBy         ?dept .
   OPTIONAL { ?contr  pred:hasContractValue ?val .
              ?contr  pred:hasManager
                "sys_context('sa$appctx','app_user_uri'}"^^orardf:instruction }
```

The apply pattern for a data access constraint can be any valid basic graph pattern with multiple triple patterns and a FILTER clause. For example, the access constraint on the `hasContractValue` property for a user with `VP` role may stipulate that the user can access the property only if he or she is the Vice President of the department driving the contract. The match and apply patterns for such constraint can be defined as follows:

```
Match:  { ?contract  pred:hasContractValue  ?cvalue  }
Apply:  { ?contract  pred:drivenBy          ?dept .
          ?dept      pred:hasVP
               "sys_context('sa$appctx','app_user_uri'}"^^orardf:instruction }
```

A match pattern defined for a data access constraint associated with an RDF class identifies all variables and terms that are known to be instances of the class. The RDF metadata defined in the VPD policy is used to determine the type for each variable and the term in a SPARQL query, and the appropriate access constraints are applied on these variables and terms. For example, the following VPD constraint ensures that a resource that is a member of the `Contract` class can only be accessed by a user who has a `hasMember` relationship with the resource:

```
Match:  { ?contract  rdf:type  <http://www.myorg.com/classes/Contract> }
Apply:  { ?contract  pred:hasMember
               "sys_context('sa$appctx','app_user_uri'}"^^orardf:instruction }
```

The class information for a variable or term appearing in a SPARQL query is derived using the domain and range information for the properties appearing in the query. In the SPARQL query in the next example, if the VPD policy has an RDF Schema statement that asserts that the domain of the `drivenBy` property is the `Contract` class, the variable `?contr` is known to hold instances of the `Contract` class. Therefore, with the previously defined access restriction for the `Contract` class, the user query is rewritten to include an appropriate apply pattern, as shown in the following example:

```
Query:
{ ?contr  pred:drivenBy    ?dept .
  ?contr  pred:hasDueDate  ?due }

Rewritten query:
{ ?contr  pred:drivenBy    ?dept .
  ?contr  pred:hasDueDate  ?due  .
  ?contr  pred:hasMember
               "sys_context('sa$appctx','app_user_uri'}"^^orardf:instruction }
```

When a basic graph pattern in a SPARQL query matches multiple data access constraints, the corresponding apply patterns are combined to form a conjunctive graph pattern, which is subsequently enforced for the specific graph pattern by logically rewriting the SPARQL query. While considering the data access constraints to be enforced for a given SPARQL query, the class and property hierarchy associated with the VPD policy is consulted to automatically enforce all applicable constraints.

- A variable or term identified as an instance of a specific RDF class enforces constraints associated with the class and all its superclasses.
- A constraint associated with a property is enforced when the user query references the property or any property defined as its subproperty or an equivalent property.

You can use the `sys_context` function in a data access constraint to enforce context-dependent access restrictions with structurally similar graph patterns. You can dynamically activate and deactivate constraint groups, based on the application context, to enforce alternate access restrictions using structurally different graph patterns.

C.1.4 RDFVPD_POLICIES View

The MDSYS.RDFVPD_POLICIES view contains information about all VPD policies defined in the schema or the policies to which the user has FULL access. If the same policy is associated with multiple models, this view has one entry for each such association. This view exists only after the semantic network and a VPD policy have been created.

The MDSYS.RDFVPD_POLICIES view contains the columns shown in Table C-1 (page C-10).

Table C-1 MDSYS.RDFVPD_POLICIES View Columns

Column Name	Data Type	Description
POLICY_OWNER	VARCHAR2(32)	Owner of the VPD policy.
POLICY_NAME	VARCHAR2(32)	Name of the VPD policy.
NAMESPACE_MAP	RDF_ALIASES	Mapping for namespace entries that are used in the VPD constraint definitions.
CONTEXT_NAME	VARCHAR2(32)	Name of the context used to manage constraint groups.

C.1.5 RDFVPD_MODELS View

The MDSYS.RDFVPD_MODELS view contains information about RDF models and their associated VPD policies. This view exists only after the semantic network and a VPD policy have been created.

The MDSYS.RDFVPD_MODELS view contains the columns shown in Table C-2 (page C-11).

Table C-2 MDSYS.RDFVPD_MODELS View Columns

Column Name	Data Type	Description
MODEL_NAME	VARCHAR2(25)	Name of the model.
POLICY_OWNER	VARCHAR2(32)	Owner of the VPD policy.
POLICY_NAME	VARCHAR2(32)	Name of the VPD policy.
OPERATION_TYPE	VARCHAR2(9)	Type of operation for which the VPD policy is enforced: `QUERY` or `DML`.

C.1.6 RDFVPD_POLICY_CONSTRAINTS View

The MDSYS.RDFVPD_POLICY_CONSTRAINTS view contains information about the constraints defined in the VPD policy that are accessible to the current user. This view exists only after the semantic network and a VPD policy have been created.

The MDSYS.RDFVPD_POLICY_CONSTRAINTS view contains the columns shown in Table C-3 (page C-11).

Table C-3 MDSYS.RDFVPD_POLICY_CONSTRAINTS View Columns

Column Name	Data Type	Description
POLICY_OWNER	VARCHAR2(32)	Owner of the VPD policy.
POLICY_NAME	VARCHAR2(32)	Name of the VPD policy.
CONSTRAINT_NAME	VARCHAR2(32)	Name of the constraint.
MATCH_PATTERN	VARCHAR2(1000)	Match pattern for the constraint.
APPLY_PATTERN	VARCHAR2(4000)	Apply pattern for the constraint.
CONSTRAINT_GROUP	VARCHAR2(32)	Name of the constraint group to which the constraint belongs. (Not case-sensitive.).

C.1.7 RDFVPD_PREDICATE_MDATA View

The MDSYS.RDFVPD_PREDICATE_MDATA view contains information about the predicate metadata associated with a VPD policy. This view exists only after the semantic network and a VPD policy have been created.

The MDSYS.RDFVPD_PREDICATE_MDATA view contains the columns shown in Table C-4 (page C-12).

Table C-4 MDSYS.RDFVPD_PREDICATE_MDATA View Columns

Column Name	Data Type	Description
POLICY_OWNER	VARCHAR2(32)	Owner of the VPD policy.
POLICY_NAME	VARCHAR2(32)	Name of the VPD policy.
PREDICATE	VARCHAR2(4000)	URI for the predicate for which the domain and range information is defined.
HASDOMAIN	VARCHAR2(4000)	URI representing the domain of the predicate.
HASRANGE	VARCHAR2(4000)	URI representing the range of the predicate.

C.1.8 RDFVPD_RESOURCE_REL View

The MDSYS.RDFVPD_RESOURCE_REL view contains information about the subclass, subproperty, and equivalence property relationships that are defined between resources in a VPD policy. This view exists only after the semantic network and a VPD policy have been created.

The MDSYS.RDFVPD_RESOURCE_REL view contains the columns shown in Table C-5 (page C-12).

Table C-5 MDSYS.RDFVPD_RESOURCE_REL View Columns

Column Name	Data Type	Description
POLICY_OWNER	VARCHAR2(32)	Owner of the VPD policy.
POLICY_NAME	VARCHAR2(32)	Name of the VPD policy.
SUBJECT_RESOURCE	VARCHAR2(4000)	Subject resource.
OBJECT_RESOURCE	VARCHAR2(4000)	Object resource.
RELATIONSHIP_TYPE	VARCHAR2(4000)	Relationship that exists between the subject resource and the object resource.

Glossary

The following are selected terms relevant to the Oracle Database implementation of RDF Semantic Graph support. This is not a comprehensive RDF and OWL glossary.

apply pattern

Part of a data access constraint defines additional graph patterns to be applied on the resources that match the match pattern before they can be used to construct the query results. *See also*: **match pattern**

basic graph pattern (BGP)

A set of triple patterns. From the W3C SPARQL Query Language for RDF Recommendation: "SPARQL graph pattern matching is defined in terms of combining the results from matching basic graph patterns. A sequence of triple patterns interrupted by a filter comprises a single basic graph pattern. Any graph pattern terminates a basic graph pattern."

clique

A graph in which every node of it is connected to, bidirectionally, every other node in the same graph.

Cytoscape

An open source bioinformatics software platform for visualizing molecular interaction networks and integrating these interactions with gene expression profiles and other state data. (See `http://www.cytoscape.org/`.) An RDF viewer (available for download) is provided as a Cytoscape plug-in.

entailment

An object containing precomputed triples that can be inferred from applying a specified set of rulebases to a specified set of models. *See also*: **rulebase**

extractor policy

A named dictionary entity that determines the characteristics of a semantic index that is created using the policy. Each extractor policy refers, directly or indirectly, to an instance of an extractor type.

graph pattern

A combination of triples constructed by combining triple patterns in various ways, including conjunction of triple patterns into groups, optionally using filter conditions, and then combining such groups using connectors similar to disjunctions, outer-joins, and so on. SPARQL querying is based around graph pattern matching.

inferencing

The ability to make logical deductions based on rules. Inferencing enables you to construct queries that perform semantic matching based on meaningful relationships among pieces of data, as opposed to just syntactic matching based on string or other values. Inferencing involves the use of rules, either supplied by Oracle or user-defined, placed in rulebases.

information extractor

An application that processes unstructured documents and extract meaningful information from them, often using natural-language processing engines with the aid of ontologies.

match pattern

Part of a constraint that determines the type of access restriction it enforces and binds one or more variables to the corresponding data instances accessed in the user query. *See also*: **apply pattern**

model

A user-created semantic structure that has a model name, and refers to triples stored in a specified table column. Examples in this manual are the Articles and Family models.

ontology

A shared conceptualization of knowledge in a particular domain. It consists of a collection of classes, properties, and optionally instances. Classes are typically related by class hierarchy (subclass/ superclass relationship). Similarly, the properties can be related by property hierarchy (subproperty/ superproperty relationship). Properties can be symmetric or transitive, or both. Properties can also have domain, ranges, and cardinality constraints specified for them.

OWLPrime

An Oracle-defined subset of OWL capabilities; refers to the elements of the OWL standard supported by the RDF Semantic Graph native inferencing engine.

RDF Semantic Graph support for Apache Jena

An Oracle-supplied adapter (available for download) for Apache Jena, which is a Java framework for building Semantic Web applications.

RDF Semantic Graph support for OpenRDF Sesame

An Oracle-supplied adapter (available for download) that integrates the popular Sesame Java APIs with RDF Semantic Graph support.

reasoning

See **inferencing**

rule

An object that can be applied to draw inferences from semantic data.

rulebase

An object that can contain rules. *See also*: **rule**

rules index

See: **entailment**

semantic index

An index of type MDSYS.SEMCONTEXT, created on textual documents stored in a column of a table, and used with information extractors to locate and extract meaningful information from unstructured documents. *See also*: **information extractor**

Simple Knowledge Organization System (SKOS)

A data model that is especially useful for representing thesauri, classification schemes, taxonomies, and other types of controlled vocabulary. SKOS is based on standard semantic web technologies including RDF and OWL, which makes it easy to define the formal semantics for those knowledge organization systems and to share the semantics across applications.

triple pattern

Similar to an RDF triple, but allows use of a variable in place of any of the three components (subject, predicate, or object). Triple patterns are basic elements in graph patterns used in SPARQL queries. A triple pattern used in a query against an RDF graph is said to match if, substitution of RDF terms for the variables present in the triple pattern, creates a triple that is present in the RDF graph. *See also*: **graph pattern**

Index

Symbols

.gv files (DOT files)
    outputting, *7-37*

A

ADD_DATATYPE_INDEX procedure, *11-1*
ADD_DEPENDENT_POLICY procedure, *14-1*
ADD_SEM_INDEX procedure, *11-2*
ADD_VPD_CONSTRAINT procedure, *15-1*
Advanced Compression, *7-9*
aggregates
    user-defined, *9-28*
aliases
    SEM_ALIASES and SEM_ALIAS data types, *1-23*, *4-6*
ALL_BGP_HASH
    query option for SEM_MATCH, *1-24*
ALL_BGP_NL
    query option for SEM_MATCH, *1-24*
ALL_LINK_HASH
    query option for SEM_MATCH, *1-24*
ALL_LINK_NL
    query option for SEM_MATCH, *1-24*
ALL_MAX_PP_DEPTH(n)
    query option for SEM_MATCH, *1-24*
ALLOW_DUP=T
    query option for SEM_MATCH, *1-24*
ALTER_DATATYPE_INDEX procedure, *11-3*
ALTER_ENTAILMENT procedure, *11-3*
ALTER_MODEL procedure, *11-4*
ALTER_SEM_INDEX_ON_ENTAILMENT procedure, *11-5*
ALTER_SEM_INDEX_ON_MODEL procedure, *11-6*
ANALYZE_ENTAILMENT procedure, *11-7*
ANALYZE_MODEL procedure, *11-9*
apply pattern, *C-7*
APPLY_OLS_POLICY procedure, *15-3*
APPLY_POLICY_TO_APP_TAB procedure, *12-1*
APPLY_VPD_POLICY procedure, *15-5*

B

BASE keyword
    global prefix, *1-49*
basic graph pattern (BGP), *1-24*
batch (bulk) loading, *11-11*, *11-60*
batch loading semantic data, *1-77*
best effort
    specifying for SPARQL query, *7-81*
BGP (basic graph pattern), *1-24*
blank nodes, *1-8*
bulk loading, *11-11*, *11-60*
bulk loading semantic data, *1-74*
BULK_LOAD_FROM_STAGING_TABLE procedure, *11-11*

C

Calais
    configuring the Calais extractor type, *4-12*
canonical forms, *1-6*
catsem.sql script, *A-1*
change tracking
    disabling, *11-34*
    enabling, *11-41*
    getting information, *11-48*
CLEANUP_FAILED procedure, *11-13*
client identifiers, *7-9*
cliques (sameAs), *2-11*
COMPOSE_RDF_TERM function, *11-13*
connection pooling
    support in RDF Semantic Graph support for Apache Jena, *7-40*
    support in RDF Semantic Graph support for OpenRDF Sesame, *8-20*
CONSTRUCT_STRICT=T
    query option for SEM_MATCH, *1-25*
CONSTRUCT_UNIQUE=T
    query option for SEM_MATCH, *1-25*
constructors for semantic data, *1-20*
convert_old_rdf_data procedure, *A-5*
corpus-centric inference, *4-16*
CREATE_ENTAILMENT procedure, *11-15*

CREATE_POLICY procedure, *14-2*
CREATE_RDFVIEW_MODEL procedure, *11-25*
CREATE_RULEBASE procedure, *11-27*
CREATE_SEM_MODEL procedure, *11-27*
CREATE_SEM_NETWORK procedure, *11-28*
CREATE_SOURCE_EXTERNAL_TABLE procedure, *11-29*
CREATE_VIRTUAL_MODEL procedure, *11-31*
CREATE_VPD_POLICY procedure, *15-6*

D

data type indexes
  adding, *11-1*
  altering, *11-3*
  dropping, *11-35*
  SEM_DTYPE_INDEX_INFO view, *1-89*
  using, *1-88*
data types
  for literals, *1-6*
data types for semantic data, *1-20*
default.xslt file
  customizing, *7-82*
DELETE_ENTAILMENT_STATS procedure, *11-33*
DELETE_MODEL_STATS procedure, *11-33*
DELETE_NETWORK_STATS procedure, *13-1*
DELETE_VPD_CONSTRAINT procedure, *15-8*
demo files
  semantic data, *1-93*
DISABLE_CHANGE_TRACKING procedure, *11-34*
DISABLE_INC_INFERENCE procedure, *11-35*
DISABLE_OLS_POLICY procedure, *15-8*
discussion forum
  RDF Semantic Graph, *1-103*
document-centric inference, *4-16*
documents
  semantic indexing for, *4-1*
DOT files
  outputting, *7-37*
downgrading
  semantic technologies support to Release 11.1, *A-6*
downloads
  RDF Semantic Graph, *1-103*
DROP_DATATYPE_INDEX procedure, *11-35*
DROP_ENTAILMENT procedure, *11-36*
DROP_POLICY procedure, *14-3*
DROP_RDFVIEW_MODEL procedure, *11-37*
DROP_RULEBASE procedure, *11-38*
DROP_SEM_INDEX procedure, *11-38*
DROP_SEM_MODEL procedure, *11-39*
DROP_SEM_NETWORK procedure, *11-39*
DROP_USER_INFERENCE_OBJS procedure, *11-40*
DROP_VIRTUAL_MODEL procedure, *11-40*
DROP_VPD_POLICY procedure, *15-9*
duplicate triples
  checking for, *1-6*
duplicate triples *(continued)*
  removing from model, *11-63*, *11-65*

E

ENABLE_CHANGE_TRACKING procedure, *11-41*
ENABLE_INC_INFERENCE procedure, *11-42*
ENABLE_OLS_POLICY procedure, *15-10*
ENABLE_SYNTAX_CHECKING optimizer hint, *2-7*
entailment
  invalid status, *1-24*
entailment rules, *1-9*
entailments
  altering, *11-3*
  deleting if in failed state, *11-13*
  incomplete status, *1-24*, *2-20*
  invalid status, *2-20*
  SEM_RULES_INDEX_DATASETS view, *1-12*
  SEM_RULES_INDEX_INFO view, *1-12*
ESCAPE_CLOB_TERM procedure, *11-42*
ESCAPE_CLOB_VALUE procedure, *11-43*
ESCAPE_RDF_TERM procedure, *11-44*
ESCAPE_RDF_VALUE procedure, *11-45*
examples
  Java (on Oracle Technology Network), *1-93*
  PL/SQL, *1-93*
EXPORT_ENTAILMENT_STATS procedure, *11-46*
EXPORT_MODEL_STATS procedure, *11-46*
EXPORT_NETWORK_STATS procedure, *13-2*
EXPORT_RDFVIEW_MODEL procedure, *11-47*
exporting semantic data, *1-74*
external documents
  indexing, *4-10*
external table
  creating, *11-29*
extractor policies
  RDFCTX_POLICIES view, *4-17*, *4-18*
extractors
  information
    policies, *4-4*

F

failed state
  rulebase or entailment, *11-13*
federated queries, *1-61*, *7-22*
filter
  attribute of SEM_MATCH, *1-23*, *4-6*
FINAL_VALUE_HASH
  query option for SEM_MATCH, *1-25*
FINAL_VALUE_NL
  query option for SEM_MATCH, *1-25*
functions
  user-defined, *9-28*

G

GATE (General Architecture for Text Engineering)
  sample Java implementation, *4-13*
  using, *4-12*
GATHER_STATS procedure, *13-3*
General Architecture for Text Engineering (GATE)
  sample Java implementation, *4-13*
  using, *4-12*
geometry literal, *1-67*
GET_CHANGE_TRACKING_INFO procedure, *11-48*
GET_INC_INF_INFO procedure, *11-49*
GET_MODEL_ID function, *11-50*
GET_MODEL_NAME function, *11-51*
GET_TRIPLE_ID function, *11-51*
GETV$DATETIMETZVAL function, *11-52*
GETV$DATETZVAL function, *11-53*
GETV$NUMERICVAL function, *11-54*
GETV$STRINGVAL function, *11-55*
GETV$TIMETZVAL function, *11-56*
global prefix (BASE keyword), *1-49*
GRAPH_MATCH_UNNAMED=T
  query option for SEM_MATCH, *1-25*
graphs
  attribute of SEM_MATCH, *1-27*

H

HINT0
  query option for SEM_MATCH, *1-25*
HTTP_METHOD=POST_PAR
  query option for SEM_MATCH, *1-26*

I

IMPORT_ENTAILMENT_STATS procedure, *11-57*
IMPORT_MODEL_STATS procedure, *11-58*
IMPORT_NETWORK_STATS procedure, *13-4*
incremental inference
  disabling, *11-35*
  enabling, *11-42*
incremental inferencing
  getting information, *11-49*
index_status
  attribute of SEM_MATCH, *1-24*, *2-20*
inf_ext_user_func_name parameter, *9-2*
inferencing
  user-defined, *9-1*
information extractors, *4-2*
inverseOf keyword
  using to force use of semantic index, *2-23*
IS_TRIPLE function, *11-59*

J

Java examples
  GATE listener, *4-13*
Java examples *(continued)*
  OTN RDF Semantic Graph page, *1-93*
JavaScript Object Notation (JSON) format support, *7-51*
Join Push Down, *1-63*
JSON format support, *7-51*

L

literals
  data types for, *1-6*
LOAD_INTO_STAGING_TABLE procedure, *11-60*
loading semantic data
  bulk, *11-11*, *11-60*
LOOKUP_ENTAILMENT procedure, *11-61*

M

MAINT_VPD_METADATA procedure, *15-10*
MAINTAIN_TRIPLES procedure, *14-4*
match pattern, *C-7*
mdsys.SemContent index type, *4-5*
MERGE_MODELS procedure, *11-62*
metadata
  semantic, *1-3*
metadata tables and views for semantic data, *1-18*
methods for semantic data, *1-20*
model ID
  getting, *11-50*
model name
  getting, *11-51*
models
  altering, *11-4*
  creating, *11-27*
  deleting (dropping), *11-39*
  disabling support in the database, *11-39*
  enabling support in the database, *11-28*
  merging, *11-62*
  renaming, *11-67*
  SEM_MODELS data type, *1-23*
  SEMI_entailment-name view, *1-11*
  SEMM_model-name view, *1-4*
  swapping names, *11-69*
  virtual, *1-13*

N

N-Quad format, *1-16*
N-QUADS data format, *1-17*
N-Triple format, *1-16*
named graph based inference
  global, *2-15*
  local, *2-15*
named graphs
  support for, *1-16*
named_graphs
  attribute of SEM_MATCH, *1-27*

network indexes
refreshing information, 11-65
SEM_NETWORK_INDEX_INFO view, 1-87
NGGI (named graph based global inference), 2-15
NGLI (named graph based local inference), 2-15

O

OBIEE
using SPARQL Gateway as an XML data source, 7-91
objects, 1-8
ODCIAggregate interface
user-defined aggregates (RDF Semantic Graph), 9-32
ogcf
boundary function, B-2
buffer function, B-3
convexHull function, B-4
difference function, B-5
distance function, B-6
envelope function, B-7
getSRID function, B-7
intersection function, B-8
relate function, B-9
sfContains function, B-10
sfCrosses function, B-11
sfDisjoint function, B-12
sfEquals function, B-13
sfIntersects function, B-14
sfOverlaps function, B-15
sfTouches function, B-16
sfWithin function, B-17
symDifference function, B-18
union function, B-19
OLTP compression, 7-9
OLTP index compression, 1-71
options
attribute of SEM_MATCH, 1-24
Oracle Advanced Compression
OLTP compression, 7-9
Oracle Business Intelligence Enterprise Edition (OBIEE)
using SPARQL Gateway as an XML data source, 7-91
Oracle Label Security (OLS)
applying policy, 12-1, 15-3
disabling policy, 15-8
enabling policy, 15-10
removing policy, 12-2, 15-12
resetting labels associated with a model, 15-13
resource-level security, 5-11
setting sensitivity label for a resource that may be used in the subject and/or object position of a triple, 15-16
setting sensitivity level for a predicate, 15-14
Oracle Label Security (OLS) *(continued)*
setting sensitivity level for a rule belonging to a rulebase, 15-18
setting sensitivity level for RDFS schema elements, 15-15
triple-level security, 5-2
using with RDF data, 5-1
Oracle Spatial and Graph
prerequisite software for RDF and OWL capabilities, A-6
orageo
area function, B-20
buffer function, B-21
centroid function, B-22
convexHull function, B-22
difference function, B-23
distance function, B-24
intersection function, B-25
length function, B-26
mbr function, B-27
nearestNeighbor function, B-28
relate function, B-29
union function, B-31
withinDistance function, B-32
xor function, B-33
OTN page
RDF Semantic Graph, 1-103
owl
sameAs
SEMCL_entailment-name view, 2-12
OWL
queries using the SEM_DISTANCE ancillary operator, 2-20
queries using the SEM_RELATED operator, 2-19
SameAs
optimizing inference, 2-11
OWL 2 EL support, 2-3
OWL 2 RL support, 2-3
OWL2EL rulebase, 2-3
OWL2RL rulebase, 2-3

P

parallel inference, 2-14
Partitioning
must be enabled for RDF and OWL, A-6
PelletInfGraph class
support deprecated in RDF Semantic Graph support for Apache Jena, 7-44
PRIVILEGE_ON_APP_TABLES procedure, 11-63
PROCAVFH=F option, 2-11
PROCSVFH=F option, 2-11
properties, 1-8
property chain handling, 3-4
property paths
optimized handling by RDF Semantic Graph support for Apache Jena, 7-16

PURGE_UNUSED_VALUES procedure, *11-64*

Q

quality of search, *4-9*
queries
  using the SEM_DISTANCE ancillary operator, *2-20*
  using the SEM_MATCH table function, *1-22*
  using the SEM_RELATED operator, *2-19*

R

RDF rulebase
  subset of RDFS rulebase, *1-9*
RDF Semantic Graph
  overview, *1-1*
RDF Semantic Graph support
  removing, *A-7*
RDF Semantic Graph support for Apache Jena
  functions supported in SPARQL queries, *7-24*
  optimized handling of SPARQL queries, *7-15*
  optimized handling or property paths, *7-16*
  query examples, *7-56*
  RDFa support with prepareBulk, *7-46*
  SEM_MATCH and RDF Semantic Graph support for Apache Jena queries compared, *7-11*
  setting up software environment, *7-2*
  setting up SPARQL service, *7-4*
  support for connection pooling, *7-40*
  support for semantic model PL/SQL interfaces, *7-41*
  support for virtual models, *7-38*
RDF Semantic Graph support for OpenRDF Sesame
  optimized handling of SPARQL queries, *8-16*
  performance recommendations, *8-16*
  query examples, *8-27*
  SEM_MATCH and RDF Semantic Graph support for OpenRDF Sesame queries compared, *8-15*
  setting up software environment, *8-3*
  setting up SPARQL endpoint, *8-4*
  support for connection pooling, *8-20*
  support for semantic model PL/SQL interfaces, *8-21*
  support for virtual models, *8-19*
RDF views
  creating, *11-25*
  dropping, *11-37*
  exporting, *11-47*
RDF_VALUE$ table, *1-5*
RDF$ET_TAB table, *1-77*
RDFa
  support with prepareBulk (RDF Semantic Graph support for Apache Jena), *7-46*
RDFCTX_INDEX_EXCEPTIONS view, *4-18*
RDFCTX_POLICIES view, *4-17*, *4-18*
RDFOLS_SECURE_RESOURCE view, *5-18*
RDFS entailment rules, *1-9*
RDFS rulebase
  implements RDFS entailment rules, *1-9*
RDFVPD_MODELS view, *C-11*
RDFVPD_POLICIES view, *C-10*
RDFVPD_POLICY_CONSTRAINTS view, *C-11*
RDFVPD_PREDICATE_MDATA view, *C-11*
RDFVPD_RESOURCE_REL view, *C-12*
REFRESH_SEM_NETWORK_INDEX_INFO procedure, *11-65*
relational data as RDF, *10-1*
REMOVE_DUPLICATES procedure, *11-65*
REMOVE_OLS_POLICY procedure, *15-12*
REMOVE_POLICY_FROM_APP_TAB procedure, *12-2*
REMOVE_VPD_POLICY procedure, *15-12*
removing RDF Semantic Graph, *A-7*
RENAME_ENTAILMENT procedure, *11-66*
RENAME_MODEL procedure, *11-67*
REPLACE=T option, *11-31*
RESET_MODEL_LABELS procedure, *15-13*
Resource Description Framework
  See RDF Semantic Graph
resource-level security, *5-11*
resultsPerPage parameter, *7-89*
rulebases
  attribute of SEM_MATCH, *2-20*
  deleting if in failed state, *11-13*
  SEM_RULEBASE_INFO view, *1-10*
  SEM_RULEBASES data type, *1-23*
  SEMR_rulebase-name view, *1-10*
rules, *1-8*
rules indexes
  See entailments

S

sameAs
  optimizing inference (OWL), *2-11*
sameTerm built-in function, *1-70*
sdo_rdf_internal.convert_old_rdf_data procedure, *A-5*
SDO_RDF_TERM data type, *9-28*
SDO_RDF_TERM_LIST data type, *9-29*
sdordfwm_rm.sql script, *6-2*
sdordfwm.sql script, *6-2*
search
  quality of, *4-9*
security considerations, *1-17*
SEM_ALIAS data type, *1-23*, *4-6*
SEM_ALIASES data type, *1-23*, *4-6*
SEM_APIS package
  ADD_DATATYPE_INDEX, *11-1*
  ADD_SEM_INDEX, *11-2*
  ALTER_DATATYPE_INDEX, *11-3*
  ALTER_ENTAILMENT, *11-3*
  ALTER_MODEL, *11-4*

SEM_APIS package *(continued)*
ALTER_SEM_INDEX_ON_ENTAILMENT
semantic network indexes
altering on entailment, *11-5*
ALTER_SEM_INDEX_ON_MODEL, *11-6*
ANALYZE_ENTAILMENT, *11-7*
ANALYZE_MODEL, *11-9*
BULK_LOAD_FROM_STAGING_TABLE, *11-11*
CLEANUP_FAILED, *11-13*
COMPOSE_RDF_TERM, *11-13*
CREATE_ENTAILMENT, *11-15*
CREATE_RDFVIEW_MODEL, *11-25*
CREATE_RULEBASE, *11-27*
CREATE_SEM_MODEL, *11-27*
CREATE_SEM_NETWORK, *11-28*
CREATE_SOURCE_EXTERNAL_TABLE, *11-29*
CREATE_VIRTUAL_MODEL, *11-31*
DELETE_ENTAILMENT_STATS, *11-33*
DELETE_MODEL_STATS, *11-33*
DISABLE_CHANGE_TRACKING, *11-34*
DISABLE_INC_INFERENCE, *11-35*
DROP_DATATYPE_INDEX, *11-35*
DROP_ENTAILMENT, *11-36*
DROP_RDFVIEW_MODEL, *11-37*
DROP_RULEBASE, *11-38*
DROP_SEM_INDEX, *11-38*
DROP_SEM_MODEL, *11-39*
DROP_SEM_NETWORK, *11-39*
DROP_USER_INFERENCE_OBJS, *11-40*
DROP_VIRTUAL_MODEL, *11-40*
ENABLE_CHANGE_TRACKING, *11-41*
ENABLE_INC_INFERENCE, *11-42*
ESCAPE_CLOB_TERM, *11-42*
ESCAPE_CLOB_VALUE, *11-43*
ESCAPE_RDF_TERM, *11-44*
ESCAPE_RDF_VALUE, *11-45*
EXPORT_ENTAILMENT_STATS, *11-46*
EXPORT_MODEL_STATS, *11-46*
EXPORT_RDFVIEW_MODEL, *11-47*
GET_CHANGE_TRACKING_INFO, *11-48*
GET_INC_INF_INFO, *11-49*
GET_MODEL_ID, *11-50*
GET_MODEL_NAME, *11-51*
GET_TRIPLE_ID, *11-51*
GETV$DATETIMETZVAL, *11-52*
GETV$DATETZVAL, *11-53*
GETV$NUMERICVAL, *11-54*
GETV$STRINGVAL, *11-55*
GETV$TIMETZVAL, *11-56*
IMPORT_ENTAILMENT_STATS, *11-57*
IMPORT_MODEL_STATS, *11-58*
LOAD_INTO_STAGING_TABLE, *11-60*
LOOKUP_ENTAILMENT, *11-61*
MERGE_MODELS, *11-62*
PRIVILEGE_ON_APP_TABLES, *11-63*
PURGE_UNUSED_VALUES, *11-64*
SEM_APIS package *(continued)*
reference information, *11-1*, *13-1*
REFRESH_SEM_NETWORK_INDEX_INFO, *11-65*
REMOVE_DUPLICATES, *11-65*
RENAME_ENTAILMENT, *11-66*
RENAME_MODEL, *11-67*
SET_ENTAILMENT_STATS, *11-68*
SET_MODEL_STATS, *11-68*
SWAP_NAMES, *11-69*
TRIPLE, *11-59*
UNESCAPE_CLOB_TERM, *11-70*
UNESCAPE_CLOB_VALUE, *11-71*
UNESCAPE_RDF_TERM, *11-72*
UNESCAPE_RDF_VALUE, *11-72*
VALIDATE_ENTAILMENT, *11-73*
VALIDATE_MODEL, *11-74*
VALUE_NAME_PREFIX, *11-75*, *11-76*
SEM_CONTAINS operator
syntax, *4-6*
SEM_CONTAINS_COUNTancillary operator
syntax, *4-7*
SEM_CONTAINS_SELECT ancillary operator
syntax, *4-7*
using in queries, *4-8*
SEM_DISTANCE ancillary operator, *2-20*
SEM_DTYPE_INDEX_INFO view, *1-89*
SEM_GRAPHS data type, *11-17*
SEM_INDEXTYPE index type, *2-22*
SEM_MATCH table function, *1-22*
SEM_MODEL$ view
virtual model entries, *1-14*
SEM_MODELS data type, *1-23*
SEM_NETWORK_INDEX_INFO view, *1-87*
SEM_OLS package
APPLY_POLICY_TO_APP_TAB, *12-1*
REMOVE_POLICY_FROM_APP_TAB, *12-2*
SEM_PERF package
DELETE_NETWORK_STATS, *13-1*
EXPORT_NETWORK_STATS, *13-2*
GATHER_STATS, *13-3*
IMPORT_NETWORK_STATS, *13-4*
SEM_RDFCTX package
ADD_DEPENDENT_POLICY, *14-1*
CREATE_POLICY, *14-2*
DROP_POLICY, *14-3*
MAINTAIN_TRIPLES, *14-4*
reference information, *14-1*
SET_DEFAULT_POLICY, *14-5*
SET_EXTRACTOR_PARAM, *14-6*
SEM_RDFSA package
ADD_VPD_CONSTRAINT, *15-1*
APPLY_OLS_POLICY, *15-3*
APPLY_VPD_POLICY, *15-5*
CREATE_VPD_POLICY, *15-6*
DELETE_VPD_CONSTRAINT, *15-8*

SEM_RDFSA package *(continued)*
DISABLE_OLS_POLICY, *15-8*
DROP_VPD_POLICY, *15-9*
ENABLE_OLS_POLICY, *15-10*
MAINT_VPD_METADATA, *15-10*
reference information, *12-1*, *15-1*
REMOVE_OLS_POLICY, *15-12*
REMOVE_VPD_POLICY, *15-12*
RESET_MODEL_LABELS, *15-13*
SET_PREDICATE_LABEL, *15-14*
SET_RDFS_LABEL, *15-15*
SET_RESOURCE_LABEL, *15-16*
SET_RULE_LABEL, *15-18*
SEM_RELATED operator, *2-19*
SEM_RULEBASE_INFO view, *1-10*
SEM_RULEBASES data type, *1-23*
SEM_RULES_INDEX_DATASETS view, *1-12*
SEM_RULES_INDEX_INFO view, *1-12*
SEM_VMODEL_DATASETS view, *1-15*
SEM_VMODEL_INFO view, *1-14*
semantic data
blank nodes, *1-8*
constructors, *1-20*
data types, *1-20*
demo files, *1-93*
examples (Java), *1-93*
examples (PL/SQL), *1-93*
in the database, *1-3*
metadata, *1-3*
metadata tables and views, *1-18*
methods, *1-20*
modeling, *1-3*
objects, *1-8*
properties, *1-8*
queries using the SEM_MATCH table function, *1-22*
security considerations, *1-17*
statements, *1-5*
steps for using, *1-92*
subjects, *1-8*
semantic index
creating (MDSYS.SEM_INDEXTYPE), *2-22*
indexing documents, *4-5*
using for documents, *4-1*
semantic indexes
RDFCTX_INDEX_EXCEPTIONS view, *4-18*
semantic network indexes
adding, *11-2*
altering on model, *11-6*
dropping, *11-38*
using, *1-86*
semantic technologies support
downgrading to Release 11.1, *A-6*
enabling, *A-1*
upgrading from Release 11.1, *A-4*
SEMCL_entailment-name view, *2-12*
SemContent
mdsys.SemContent index type, *4-5*
SEMI_entailment-name view, *1-11*
SEMM_model-name view, *1-4*
SEMR_rulebase-name view, *1-10*
semrelod.sql script, *A-5*
semrelod111.sql script, *A-7*
semremov.sql script, *A-8*
SERVICE_CLOB=f
query option for SEM_MATCH, *1-26*
SERVICE_ESCAPE=f
query option for SEM_MATCH, *1-26*
SERVICE_JPDWN=t
query option for SEM_MATCH, *1-26*
SERVICE_PROXY
query option for SEM_MATCH, *1-27*
SET_DEFAULT_POLICY procedure, *14-5*
SET_ENTAILMENT_STATS procedure, *11-68*
SET_EXTRACTOR_PARAM procedure, *14-6*
SET_MODEL_STATS procedure, *11-68*
SET_PREDICATE_LABEL procedure, *15-14*
SET_RDFS_LABEL procedure, *15-15*
SET_RESOURCE_LABEL procedure, *15-16*
SET_RULE_LABEL procedure, *15-18*
Simple Knowledge Organization System (SKOS)
property chain handling, *3-4*
support for, *3-1*
SKOS (Simple Knowledge Organization System)
property chain handling, *3-4*
support for, *3-1*
SNOMED CT (Systematized Nomenclature of Medicine - Clinical Terms)
support for, *11-20*
SPARQL
configuring the service, *7-6*, *7-9*, *8-14*
optimized handling of queries, *7-15*, *8-16*
searching for documents using SPARQL query pattern, *4-7*
setting up service for RDF Semantic Graph support for Apache Jena, *7-4*
setting up service for RDF Semantic Graph support for OpenRDF Sesame, *8-4*
SPARQL endpoints
accessing with HTTP Basic authentication, *1-64*
SPARQL Gateway
customizing the default XSLT file, *7-82*
features and benefits overview, *7-76*
installing and configuring, *7-76*
Java API, *7-82*
specifying best effort for SPARQL query, *7-81*
specifying content type other than text/xml, *7-82*
specifying timeout value for SPARQL query, *7-81*
using as an XML data source to OBIEE, *7-91*
using with semantic data, *7-78*
SPARQL SERVICE
federated queries, *1-61*
Join Push Down, *1-63*

SPARQL SERVICE *(continued)*
SILENT keyword, *1-63*
using a proxy server with, *1-64*
Spatial and Graph
prerequisite software for RDF and OWL capabilities, *A-6*
spatial support
ogcf
boundary function, *B-2*
buffer function, *B-3*
convexHull function, *B-4*
difference function, *B-5*
distance function, *B-6*
envelope function, *B-7*
getSRID function, *B-7*
intersection function, *B-8*
relate function, *B-9*
sfContains function, *B-10*
sfCrossesfunction, *B-11*
sfDisjoint function, *B-12*
sfEquals function, *B-13*
sfIntersects function, *B-14*
sfOverlaps function, *B-15*
sfTouches function, *B-16*
sfWithin function, *B-17*
symDifference function, *B-18*
union function, *B-19*
orageo
area function, *B-20*
buffer function, *B-21*
centroid function, *B-22*
convexHull function, *B-22*
difference function, *B-23*
distance function, *B-24*
intersection function, *B-25*
length function, *B-26*
mbr function, *B-27*
nearestNeighbor function, *B-28*
relate function, *B-29*
union function, *B-31*
withinDistance function, *B-32*
xor function, *B-33*
staging table
loading data from, *11-11*
loading data into, *11-60*
staging table for bulk loading semantic data, *1-74*
statements
RDF_VALUE$ table, *1-5*
statistics
gathering for RDF and OWL, *13-3*
STRICT_AGG_CARD=T
query option for SEM_MATCH, *1-27*
subjects, *1-8*
subproperty chaining, *3-4*
SWAP_NAMES procedure, *11-69*
Systematized Nomenclature of Medicine - Clinical Terms (SNOMED CT)
Systematized Nomenclature of Medicine - Clinical Terms (SNOMED CT
support for, *11-20*

T

timeout value
specifying for SPARQL query, *7-81*
TriG data format, *1-17*
triple-level security, *5-2*
triples
constructors for inserting, *1-21*
duplication checking, *1-6*
IS_TRIPLE function, *11-59*

U

UNESCAPE_CLOB_TERM procedure, *11-70*
UNESCAPE_CLOB_VALUE procedure, *11-71*
UNESCAPE_RDF_TERM procedure, *11-72*
UNESCAPE_RDF_VALUE procedure, *11-72*
uninstalling RDF Semantic Graph support, *A-7*
unused values
purging from semantic network, *11-64*
upgrading
semantic technologies support from Release 11.1, *A-4*
URI prefix
using when values are not stored as URIs, *2-24*
URIPREFIX keyword, *2-24*
user-defined aggregates, *9-28*
user-defined functions, *9-28*
user-defined inferencing, *9-1*
user-defined inferencing function, *9-2*
user-defined querying, *9-1*

V

VALIDATE_ENTAILMENT procedure, *11-73*
VALIDATE_MODEL procedure, *11-74*
VALUE_NAME_PREFIX function, *11-75*, *11-76*
version-enabling an RDF model, *6-2*, *6-4*
views
RDF
creating, *11-25*
dropping, *11-37*
exporting, *11-47*
virtual models
SEM_MODEL$ view entries, *1-14*
SEM_VMODEL_DATASETS view, *1-15*
SEM_VMODEL_INFO view, *1-14*
support in RDF Semantic Graph support for Apache Jena, *7-38*
support in RDF Semantic Graph support for OpenRDF Sesame, *8-19*
virtual private database (VPD)
adding constraint, *15-1*
applying policy, *15-5*

virtual private database (VPD) *(continued)*
creating policy, *15-6*
deleting constraint, *15-8*
dropping policy, *15-9*
maintaining metadata, *15-10*
removing policy, *15-12*
using with RDF data, *C-2*
Virtual Private Database (VPD), *C-10*–*C-12*

W

Workspace Manager
enabling support for RDF data, *6-2*
inferring from version-enabled RDF models, *6-3*
Workspace Manager *(continued)*
merging and refreshing workspaces, *6-4*
removing support for RDF data, *6-2*
restrictions and considerations with RDF data, *6-5*
restrictions with version-enabled models, *6-5*
usage flow example, *6-5*
version-enabling an RDF model, *6-2*
workspaces
merging and refreshing, *6-4*

X

XSLT file
customizing default for SPARQL Gateway, *7-82*