Oracle® Database

Semantic Technologies Developer's Guide

11*g* Release 2 (11.2)

E11828-02

August 2009

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 Database Semantic Technologies Developer's Guide, 11*g* Release 2 (11.2)

E11828-02

Primary Author: Chuck Murray

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

Contents

2 OWL Concepts

3 Simple Knowledge Organization System (SKOS) Support

4 Semantic Indexing for Documents

5 Fine-Grained Access Control for RDF Data

6 Workspace Manager Support for RDF Data

7 SEM_APIS Package Subprograms

List of Examples

List of Figures

List of Tables

Preface

Oracle Database Semantic Technologies 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 Semantic Technologies feature is licensed with the Oracle Spatial 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 Oracle semantic technologies support. These actions and prerequisites are explained in Section 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

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

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.

What's New in Semantic Technologies?

This section describes new and changed semantic technologies features for Oracle Database Release 11.

Release 11.2 Features

The following are new and changed features for Oracle Database 11*g* Release 2 (11.2).

This release also includes the features that were supplied in the interim patch 7600122 for Release 11.1.0.7.0, which are listed in "Features Added for Release 11.1.0.7 (November, 2008)".

Semantic Indexing for Documents

You can use semantic indexing to enable queries on information extracted from unstructured documents. Documents indexed semantically can be searched using the SEM_CONTAINS operator within a standard SQL query. For information about using semantic indexing, see Chapter 4.

The SEM_RDFCTX PL/SQL package contains subprograms to manage extractor policies and semantic indexes created for documents. This package is documented in Chapter 9.

Virtual Private Database (VPD) and Oracle Label Security (OLS) Support

You can enforce a fine-grained access control mechanism for RDF data by using either the Virtual Private Database (VPD) or Oracle Label Security (OLS) feature of Oracle Database, as explained in Chapter 5.

Workspace Manager Support for RDF Data

You can use Oracle Workspace Manager to version-enable RDF data in the semantic data store, as explained in Chapter 6.

New Procedures to Enable, Downgrade, and Remove Semantic Technologies Support

The procedures for enabling, downgrading, and removing semantic technologies support in the database have been redesigned. These procedures are explained in Appendix A.

Note that you still must perform certain actions and meet prerequisites before you can use any types, synonyms, or PL/SQL packages related to Oracle semantic technologies support. These actions and prerequisites are explained in Section A.1.

Optimized owl:sameAs Inference

You can optimize the performance of `owl:sameAs` inference, as explained in Section 2.2.8.

Optimizing Entailment Performance Through Compact Structures

You can specify `RAW8=T` in the `options` parameter to the SEM_APIS.CREATE_ENTAILMENT procedure, to cause RAW8 datatypes to be used instead of NUMBER in many intermediate tables created during the inference process. This option can improve entailment performance by up to 30% in some cases.

Incremental Inference

You can use incremental inference update entailments (rules indexes) efficiently after triple additions, as explained in Section 2.2.9.

Parallel Inference

You can use parallel inference to improve inference performance by taking advantage of the capabilities of a multi-core or multi-CPU architectures, as explained in Section 2.2.10.

Filter and Union Support in Curly Brace Syntax for SEM_MATCH

You can now specify FILTER or UNION, or both, in addition to OPTIONAL as keywords in the curly brace syntax supported for the SEM_MATCH table function. These keywords are explained in Section 1.6.2.

Simple Knowledge Organization System (SKOS) Support

You can perform inferencing based on a core subset of the Simple Knowledge Organization System (SKOS) data model, as explained in Chapter 3.

Inference Beyond OWLPrime

You can specify additional inference components to cover OWL constructs, such as `owl:intersectionOf` and `owl:unionOf`, as explained in Table 7–1, " Inferencing Keywords for inf_components_in Parameter" in the Usage Notes for the SEM_APIS.CREATE_ENTAILMENT procedure in Chapter 7.

Systematized Nomenclature of Medicine - Clinical Terms (SNOMED CT) Support

The current release includes a built-in rulebase that supports the expressiveness of Systematized Nomenclature of Medicine - Clinical Terms (SNOMED CT) terminology. To include this support, specify the `SNOMED` keyword in the `inf_components_in` parameter in the call to the SEM_APIS.CREATE_ENTAILMENT procedure .

New SEM_APIS Package Subprograms

The following subprograms have been added to the SEM_APIS PL/SQL package, which is documented in Chapter 7.

- SEM_APIS.MERGE_MODELS
- SEM_APIS.REMOVE_DUPLICATES
- SEM_APIS.RENAME_MODEL
- SEM_APIS.SWAP_NAMES

Parameters for SEM_PERF.GATHER_STATISTICS

The SEM_PERF.GATHER_STATS procedure now accepts two parameters, both optional: `just_on_values_table` and `parallel`. This procedure is documented in Chapter 8.

SEM_APIS.*_RULES_INDEX Subprograms Deprecated

In the SEM_APIS PL/SQL package, the subprograms containing "_RULES_INDEX" in their names are deprecated and removed from this manual. Instead, you should use subprograms containing "_ENTAILMENT" in their names:

- Instead of SEM_APIS.ALTER_SEM_INDEX_ON_RULES_INDEX, use SEM_APIS.ALTER_SEM_INDEX_ON_ENTAILMENT.
- Instead of SEM_APIS.ANALYZE_RULES_INDEX, use SEM_APIS.ANALYZE_ENTAILMENT.
- Instead of SEM_APIS.CREATE_RULES_INDEX, use SEM_APIS.CREATE_ENTAILMENT.
- Instead of SEM_APIS.DROP_RULES_INDEX, use SEM_APIS.DROP_ENTAILMENT.
- Instead of SEM_APIS.LOOKUP_RULES_INDEX, use SEM_APIS.LOOKUP_ENTAILMENT.

The old "*_RULES_INDEX" formats will continue to work, but you are encouraged to switch to using the "*_ENTAILMENT" subprograms.

Note that several metadata views still include "RULES_INDEX" in their names.

The terms *entailment* and *rules index* are synonyms in this manual, although *entailment* is used predominantly

Release 11.1 Features

The following are new and changed features for Oracle Database 11*g* Release 1 (11.1).

Features Added for Release 11.1.0.7 (November, 2008)

This section describes features that are included in interim patch 7600122 for Release 11.1.0.7.0, which was made available on Oracle *MetaLink* in November, 2008.

- Support for virtual models (see Section 1.3.8)
- Curly brace syntax for SEM_MATCH graph pattern, including support for the OPTIONAL construct (see Section 1.6.2)
- Using HINT0 ("hint-zero") in a SEM_MATCH query (see Section 1.6)
- New columns returned from SEM_MATCH: id, _prefix, _suffix (see Section 1.6)
- Ability to create and manage semantic network indexes on models and rules indexes (see Section 1.8)
- SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE procedure: New options in the flags parameter join_hint for tasks IZC, MBV, and MBT; parallel=n)
- Simplification of staging table definition (fewer columns) and privilege requirements (see Section 1.7.1)
- Simpler event tracing during bulk load (see the information about the new RDF$ET_TAB table in Section 1.7.1.1)

Storage Model Enhancements and Migration

The storage model has been enhanced to support OWL inferencing: some internal data structures and indexes have been changed, added, and removed. These changes also result in enhanced performance.

Because of the extent of these changes, if you have semantic data that you used with the previous release, you must upgrade that data to migrate it to the new format before you can use any new features for this release. For more information, see Appendix A, "Enabling, Downgrading, or Removing Semantic Technologies Support".

Support for OWL Inferencing

Support has been added to support storing, validating, and querying Web Ontology Language (OWL)-based ontologies. Support is provided for a subset of the OWL DL language.

To query ontology data, you can use table functions and operators that examine semantic relationships, such as SEM_MATCH, SEM_RELATED, and SEM_DISTANCE.

New Bulk Loading Interface for Improved Performance

You can improve performance for bulk loading of semantic data in bulk using a staging table and calling the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE procedure. For more information, see Section 1.7.1.

Ontology-Assisted Querying of Relational Data

You can go beyond syntactic matching to perform semantic relatedness-based querying of relational data, by associating an ontology with the data and using the new SEM_RELATED operator (and optionally its SEM_DISTANCE ancillary operator). The new SEM_INDEXTYPE index type improves performance for semantic queries.

Required Procedure for Using Semantic Technology Support

Before you can use any types, synonyms, or PL/SQL packages related to Oracle semantic technologies support, you must enable support for semantic technologies. For more information, see Appendix A, "Enabling, Downgrading, or Removing Semantic Technologies Support".

1

Oracle Database Semantic Technologies Overview

This chapter describes the support in Oracle Database Enterprise Edition for semantic technologies, specifically Resource Description Framework (RDF) and a subset of the Web Ontology Language (OWL). It 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 Chapter 7.

The RDF and OWL support are features of Oracle Spatial, 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:

For information about OWL concepts and the Oracle Database support for OWL capabilities, see Chapter 2.

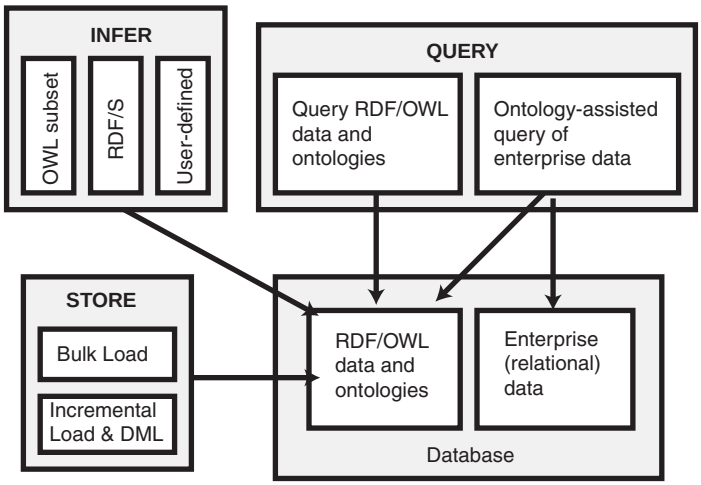
Required Actions to Enable Semantic Technologies Support:

Before performing any operations described in this guide, you must enable semantic technologies support in the database and meet other prerequisites, as explained in Section A.1, "Enabling Semantic Technologies Support".

1.1 Introduction to Oracle Semantic Technologies

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 shows how these capabilities interact.

Figure 1–1 Oracle Semantic Capabilities



As shown in Figure 1–1, 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 Section 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 Section 2.3.

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 Section 1.3.6.

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 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.

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 1–1 (Cont.) MDSYS.SEM_MODEL$ View Columns

Column Name	Data Type	Description
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.

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 RDFM_ *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.

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 graph to which the triple belongs. It logically partitions the table by RDF graphs.
COST	NUMBER	(Reserved for future use)
CTXT1	NUMBER	(Reserved for future use)
CTXT2	VARCHAR2(4000)	(Reserved for future use)
DISTANCE	NUMBER	(Reserved for future use)
EXPLAIN	VARCHAR2(4000)	(Reserved for future use)
PATH	VARCHAR2(4000)	(Reserved for future use)
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, for columns P_VALUE_ID, START_NODE_ID, END_NODE_ID, and CANON_END_NODE_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.

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.
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 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 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.)

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

Column Name	Data Type	Description
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.

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.

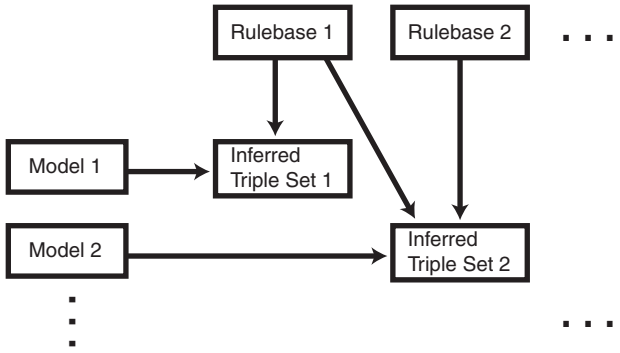
When a triple is inserted into a model, the subject, property, and object text values are checked to see if they already exist in the database. If they already exist (due to previous statements in other models), no new entries are made; if they do not exist, three new rows are inserted into the RDF_VALUE$ table (described in Section 1.3.2).

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 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)
- OWLPrime (empty)
- SKOSCORE (empty)

The RDFS and RDF rulebases are created when you call the SEM_APIS.CREATE_SEM_NETWORK 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 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.

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 Section 1.6.)

Information about all rulebases is maintained in the MDSYS.SEM_RULEBASE_INFO view, which has the columns shown in Table 1–5 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 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 1–20 in Section 1.10.2.)

Example 1–1 Inserting a Rule into a Rulebase

```
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 can be implemented using the OWL 2 property chain construct. For information about property chain handling, see Section 3.2.2.

You can specify one or more rulebases when calling the SEM_MATCH table function (described in Section 1.6), to control the behavior of queries against semantic data. Example 1–2 refers to the `family_rb` rulebase and to the `grandParentOf` relationship created in Example 1–1, to find all grandfathers (grandparents who are male) and their grandchildren. (This example is an excerpt from Example 1–20 in Section 1.10.2.)

Example 1–2 Using Rulebases for Inferencing

```
-- 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 Section 2.2.

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 procedure. To drop (delete) an entailment, use the SEM_APIS.DROP_ENTAILMENT 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 in Section 1.3.1.

Information about all entailments is maintained in the MDSYS.SEM_RULES_INDEX_INFO view, which has the columns shown in Table 1–6 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

Table 1–6 (Cont.) MDSYS.SEM_RULES_INDEX_INFO View Columns

Column Name	Data Type	Description
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 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 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 1–20 in Section 1.10.2.)

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 optionally the corresponding entailment.

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 procedure.

Note that 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 Chapter 6.)

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 procedure. To drop (delete) a virtual model, use the SEM_APIS.DROP_VIRTUAL_MODEL procedure. A virtual model is dropped automatically if any of its component models, rulebases, or entailment are dropped.

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.

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'));
```

For information about the SEM_MATCH table function, see Section 1.6, 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 in Section 1.3.1. However, the values in several of the columns are different for virtual models as opposed to semantic models, as explained in Table 1–8.

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 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
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.
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 entailment) related to virtual models is maintained in the MDSYS.SEM_VMODEL_DATASETS view. This view has

the columns shown in Table 1–10 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

1.3.9 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 procedure, as explained in Section 1.9, and some are created only as needed.) Table 1–11 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	Section 1.3.2
RDFOLS_SECURE_ RESOURCE	Resources secured with Oracle Label Security (OLS) policies and the sensitivity labels associated with these resources	Section 5.2.6
RDFVPD_MODELS	RDF models and their associated VPD policies	Section 5.1.5
RDFVPD_POLICIES	All VPD policies defined in the schema or the policies to which the user has FULL access	Section 5.1.4
RDFVPD_POLICY_ CONSTRAINTS	Constraints defined in the VPD policy that are accessible to the current user	Section 5.1.6
RDFVPD_PREDICATE_ MDATA	Predicate metadata associated with a VPD policy	Section 5.1.7
RDFVPD_RESOURCE_REL	Subclass, subproperty, and equivalence property relationships that are defined between resources in a VPD policy	Section 5.1.8
SEM_MODEL$	All models defined in the database	Section 1.3.1
SEMM_*model-name*	Triples in the specified model	Section 1.3.1
SEM_RULEBASE_INFO	Rulebases	Section 1.3.6
SEM_RULES_INDEX_ DATASETS	Database objects used in entailments	Section 1.3.7
SEM_RULES_INDEX_INFO	Entailments (rules indexes)	Section 1.3.7
SEM_VMODEL_INFO	Virtual models	Section 1.3.8
SEM_VMODEL_ DATASETS	Database objects used in virtual models	Section 1.3.8
SEMCL_*entailment-name*	`owl:sameAs` clique members and canonical representatives	Section 2.2.8
SEMI_*entailment-name*	Triples in the specified entailment	Section 1.3.7
SEMR_*rulebase-name*	Rules in the specified rulebase	Section 1.3.6
SEMU_*virtual-model-name*	Unique triples in the virtual model	Section 1.3.8
SEMV_*virtual-model-name*	Triples in the virtual model	Section 1.3.8

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
  SEM_M_ID NUMBER,  -- Model 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 a triple or a part (subject, property, or object) of a triple:

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

Example 1–5 shows the SDO_RDF_TRIPLE_S methods.

Example 1–5 SDO_RDF_TRIPLE_S Methods

```
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 Without Any Blank Nodes

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 1–6 uses the first constructor format to insert a triple.

Example 1–6 SDO_RDF_TRIPLE_S Constructor to Insert a Triple

```
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"'));
```

1.5.2 Constructors for Inserting Triples With or Without Any Blank Nodes

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

```
SDO_RDF_TRIPLE_S (
  model_name VARCHAR2, -- Model name
  sub_or_bn  VARCHAR2, -- Subject or blank node
  property   VARCHAR2, -- Property
  obj_or_bn  VARCHAR2, -- Object or blank node
  bn_m_id    NUMBER) -- ID of the model from which to reuse the blank node
  RETURN SELF;

SDO_RDF_TRIPLE_S (
  model_name VARCHAR2, -- Model name
  sub_or_bn  VARCHAR2, -- Subject or blank node
  property   VARCHAR2, -- Property
  object     CLOB, -- Object
  bn_m_id    NUMBER) -- ID of the model from which to reuse the blank node
  RETURN SELF;
```

If the value of `bn_m_id` is positive, it must be the same as the model ID of the target model.

Example 1–7 uses the first constructor format to insert a triple that reuses a blank node for the subject.

Example 1–7 SDO_RDF_TRIPLE_S Constructor to Reusing a Blank Node

```
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>',
  4));
```

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
 ) 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 enclosed in parentheses. 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 Section 1.3.8.)

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:

```
('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).

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 Section 1.6.1.

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

- 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.

- 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*>). Hints used for influencing the results of queries include GET_CANON_VALUE(<*set of aliases for variables*>), which ensures that the values

returned for the referenced variables will be their canonical lexical forms. These hints have the same format and basic meaning as hints in SQL statements, which are explained in *Oracle Database SQL Language Reference*.

Example 1–9 shows the HINT0 option used in a SEM_MATCH query.

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

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.

Example 1–8 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 1–20 in Section 1.10.2.)

Example 1–8 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 1–9 is functionally the same as Example 1–8, but it adds the `HINT0` option.

Example 1–9 HINT0 Option with 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,
    null,
    'HINT0={LEADING(t0 t1) USE_NL(?x ?y) GET_CANON_VALUE(?x ?y)}'));
```

Example 1–10 uses the Pathway/Genome BioPax ontology to get all chemical compound types that belong to both Proteins and Complexes:

Example 1–10 SEM_MATCH Table Function

```
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 1–10, 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 is must be a subclass of both Proteins and Complexes.

To use the SEM_RELATED operator to query an OWL ontology, see Section 2.3.

When you are querying multiple models or querying one or more models and the corresponding entailment, consider using virtual models (explained in Section 1.3.8) because of the potential performance benefits.

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 Section 1.3.7. The SEM_MATCH table function is described in Section 1.6.)

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, and UNION 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, or UNION 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

Example 1–11 is functionally the same as Example 1–8, but it uses the syntax with curly braces and a period to express a graph pattern in the SEM_MATCH table function.

Example 1–11 Curly Brace Syntax

```
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 1–12 uses the OPTIONAL construct to modify Example 1–11, 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.

Example 1–12 Curly Brace Syntax and OPTIONAL Construct

```
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)}'));
```

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 1–13 modifies Example 1–12 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.

Example 1–13 Curly Brace Syntax and Multi-Pattern OPTIONAL Construct

```
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));
```

A single query can contain multiple OPTIONAL graph patterns, which can be nested or parallel. Example 1–14 modifies Example 1–13 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 1–14 a value is returned for `?game` even if the nested OPTIONAL graph pattern `?y :plays ?game . ?y :age ?age` is not matched.

Example 1–14 Curly Brace Syntax and Nested OPTIONAL Construct

```
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 1–15 modifies Example 1–13 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 e-mail address or null if he has no e-mail address. Note that, unlike nested OPTIONAL graph patterns, parallel OPTIONAL graph patterns are treated independently. That is, if an e-mail 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 e-mail address was found.

Example 1–15 Curly Brace Syntax and Parallel OPTIONAL Construct

```
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 1–16 uses the FILTER construct to modify Example 1–11, so that it returns grandchildren information for only those grandfathers who are residents of either NY or CA.

Example 1–16 Curly Brace Syntax and FILTER Construct

```
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));
```

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 lists built-in functions that you can use in the FILTER clause. In the Description column of Table 1–12, *x*, *y*, and *z* are arguments of the appropriate types.

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

Function	Description
BOUND(variable)	BOUND(x) returns true if x is bound (that is, non-null) in the result, false otherwise.
DATATYPE(literal)	DATATYPE(x) returns a URI representing the datatype of x.
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.
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 x and y have the same language tag, false otherwise.
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 '$' matches the end of any line); 'i' – case insensitive mode; 'x' – remove whitespace characters from the regular expression before matching.
sameTERM(RDF term, RDF term)	sameTERM(x, y) returns true if x and y are the same RDF term, false otherwise.
STR(RDF term)	STR(x) returns a plain literal of the string representation of x (that is, what would be stored in the VALUE_NAME column of MDSYS.RDF_VALUE$ enclosed within double quotes).

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

Example 1–17 uses the REGEX built-in function to select all grandfathers that have an Oracle e-mail address. Note that backslash (\) characters in the regular expression pattern must be escaped in the query string; for example, `\\.` produces the following: pattern `\.`

Example 1–17 Curly Brace Syntax and FILTER with REGEX and STR Built-In Constructs

```
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 1–18 uses the UNION construct to modify Example 1–16, 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).

Example 1–18 Curly Brace Syntax and UNION and FILTER Constructs

```
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));
```

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 using HINT0={<*hint-string*>} (explained in Section 1.6), should be constructed only on the basis of the portion of the graph pattern outside the OPTIONAL construct. For example, the only valid aliases for use in a hint specification for the query in Example 1–12 are `t0`, `t1`, `?x`, and `?y`.
- Hints are not supported for queries involving UNION.
- The FILTER construct is not supported for variables bound to long literals.

1.7 Loading and Exporting Semantic Data

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

- Bulk load using a SQL*Loader direct-path load to get data from an N-Triple format into a staging table and then use a PL/SQL procedure to load or append data into the semantic data store, as explained in Section 1.7.1.

  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 Section 1.7.2).

  This option is slower than bulk loading, but it handles triples containing object values with more than 4000 bytes.
- Load into tables using SQL INSERT statements that call the SDO_RDF_TRIPLE_S constructor, as explained in Section 1.7.3.

To export semantic data, use the Java API, as described in Section 1.7.4.

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. The data must first be parsed to check for syntax correctness and then loaded into the staging table. Then, you can call the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE procedure (described in Chapter 7).

The following example shows the format for the staging table, including all required columns and the required names for these columns:

```
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
);
```

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
);
```

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.

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.

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, triples containing object values longer than 4000 bytes can be loaded using the following approach:

1. Use the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE procedure to load all rows that can be stored in the staging table.
2. Load the remaining rows (that is, the rejected rows when using SQL*Loader with the sample control file) from an N-Triple format file, as described in Section 1.7.2.

1.7.1.1 Recording Event Traces during Bulk Loading

If a table named RDF$ET_TAB exists in the invoker's schema and if the MDSYS user has been granted the INSERT and UPDATE privileges on this table, event traces for some of the tasks performed during executions of the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE procedure will be added to the table. You may find the content of this table useful if you ever need to report any problems in bulk load. The RDF$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 Semantic Data Using the Java API

You can perform a batch load operation 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, with Oracle Spatial, 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 `ojdbc5.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/ojdbc5.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\ojdbc5.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>. Note that 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.

1.7.2.1 When to Choose Batch Loading

Batch loading is faster than loading semantic data using INSERT statements (described in Section 1.7.3). However, bulk loading (described in Section 1.7.1) is much faster than batch loading for large amounts of data. Batch loading is typically a good option when the following conditions are true:

- The data to be loaded is less than a few million triples.
- The data contains a significant amount long literals (longer than 4000 bytes).

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 Section 1.5.1. 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 (the model_id for nsu is 4), 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>', 4));
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) 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
```

1.7.4 Exporting Semantic Data

To output semantic data to a file in N-Triple format, use the `NTripleConverter` Java class. The `NDM2NTriple(String, int)` method exports all the triples stored for the specified model.

For information about using the `NTriple converter` class, see the `README.txt` file in the `sdordf_converter.zip` file, which you can download from the Oracle Technology Network.

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
- SEM_APIS.ALTER_SEM_INDEX_ON_MODEL
- SEM_APIS.ALTER_SEM_INDEX_ON_ENTAILMENT
- SEM_APIS.DROP_SEM_INDEX

All of these subprograms have an `index_code` parameter, which can contain any sequence of the following letters (without repetition): `P`, `C`, `S`, `M`, `O`. 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, MODEL_ID, and END_NODE_ID.

The SEM_APIS.ADD_SEM_INDEX 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 `PSCM` index with the following key: <P_VALUE_ID, START_NODE_ID, CANON_END_NODE_ID, MODEL_ID>.

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

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 and SEM_APIS.ALTER_SEM_INDEX_ON_ENTAILMENT 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 `PSCM` index usable for the `FAMILY` model:

```
EXECUTE SEM_APIS.ALTER_SEM_INDEX_ON_MODEL('FAMILY','PSCM','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 `'PSCF'` when you call the subprograms mentioned in this section. (The "F" in PSCF stands for "function": a predefined function that returns NULL when `end_node_id` is equal to `canon_end_node_id`, and otherwise returns `end_node_id`. PSCF is very close to PSCO. However, because most triples have the same `canon_end_node_id` and `end_node_id`, using NULL for those cases can save resources when the index is built.)
- 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.

1.9 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 Section 1.10.

1.10 Semantic Data Examples (PL/SQL and Java)

This section contains the following PL/SQL examples:

- Section 1.10.1, "Example: Journal Article Information"
- Section 1.10.2, "Example: Family Information"

In addition to the examples in this guide, see the "Semantic Technologies Code Samples and Examples" page on the Oracle Technology Network (http://www.oracle.com/technology/tech/semantic_technologies/sample_code/). That page includes several Java examples.

1.10.1 Example: Journal Article Information

This section presents a simplified PL/SQL example of model for statements about journal articles. Example 1–19 contains descriptive comments, refer to concepts that are explained in this chapter, and uses functions and procedures documented in Chapter 7.

Example 1–19 Using a Model for Journal Article Information

```
-- Basic steps:
-- After you have connected as a privileged user and called
-- SEM_APIS.CREATE_RDF_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_RDF_MODEL).
-- 3. For each table to hold semantic data, insert data into the table.
-- 4. Use various subprograms and consructors.

-- 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_RDF_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'));
```

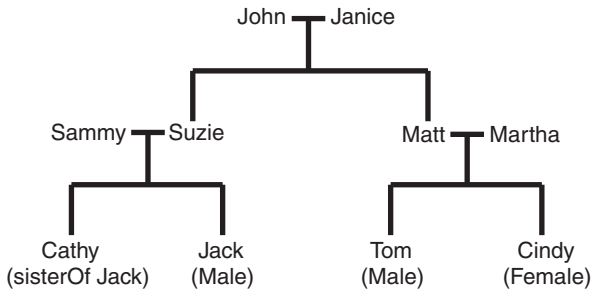
```
-- Article1 references (refers to) Article3.
INSERT INTO articles_rdf_data VALUES (4,
  SDO_RDF_TRIPLE_S ('articles',
    'http://nature.example.com/Article1',
    'http://purl.org/dc/terms/references',
    'http://nature.example.com/Article3'));

-- 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;
```

1.10.2 Example: Family Information

This section presents a simplified PL/SQL example of a model for statements about family tree (genealogy) information. Example 1–19 contains descriptive comments, refer to concepts that are explained in this chapter, and uses functions and procedures documented in Chapter 7.

The family relationships in this example reflect the family tree shown in Figure 1–3. 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 1–20 Using a Model for Family Information

```
-- Basic steps:
-- After you have connected as a privileged user and called
-- SEM_APIS.CREATE_RDF_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_RDF_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_rdf_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));
```

1.11 Software Naming Changes for Semantic Technologies

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 1–13 lists the old and new names for some objects related to support for semantic technologies, in alphabetical order by old name.

Table 1–13 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
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

1.12 For More Information About Semantic Technologies

For more information about Oracle Database semantic technology support and related topics, you may find the following resources helpful:

- Semantic Technologies Center (OTN), which includes links for downloads, technical and business white papers, a discussion forum, and other sources of information: `http://www.oracle.com/technology/tech/semantic_technologies/`
- 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 Chapter 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:

- Section 2.1, "Ontologies"
- Section 2.2, "Using OWL Inferencing"
- Section 2.3, "Using Semantic Operators to Query Relational Data"

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.

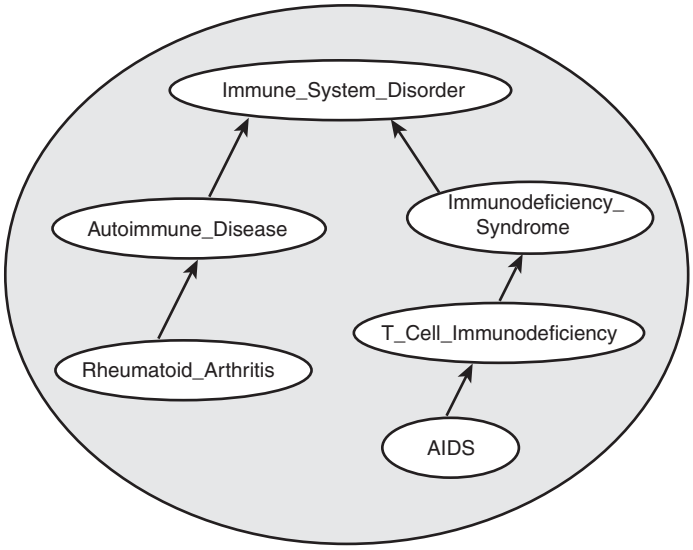
OWL ontologies build on RDFS-based ontologies by additionally allowing specification of property characteristics. OWL ontologies can be further classified as OWL-Lite, OWL-DL, and OWL Full. OWL-Lite restricts the cardinality minimum and maximum values to 0 or 1. OWL-DL relaxes this restriction by allowing 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.

Section 2.1.2 describes OWL capabilities that are supported and not supported with semantic data.

2.1.1 Example: Cancer Ontology

Figure 2–1 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, 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.

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 Section 1.6) or the SEM_RELATED operator and its ancillary operators (described in Section 2.3).

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. 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).

The following OWL capabilities are not yet supported in any Oracle-supported OWL subset:

- Property restrictions: cardinality
- Class expressions: set operations (union, intersection), enumeration

Table 2–2 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 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

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 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 calls the SEM_APIS.CREATE_ENTAILMENT procedure. You do not need to create the rulebase and add rules to it, because the OWL rules are already built into the Oracle semantic technologies 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 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 procedure.

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 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*

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 displays the LINK_ID value and associated triple contents using the model view MDSYS.SEMM_OWLTST and the entailment view MDSYS.SEMI_OWLTST_IDX.

Example 2–4 Displaying Proof Information

```
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, 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 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 and SEM_APIS.VALIDATE_ENTAILMENT functions to detect inconsistencies in the original data model and in the entailment, respectively.

Example 2–5 shows uses the SEM_APIS.VALIDATE_ENTAILMENT function, which returns a null value if no errors are detected or a VARRAY of strings if any errors are detected.

Example 2–5 Validating an Entailment

```
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)
- Error code (`10001` Example 2–5)
- One or more triples (shown in parentheses in the output; `(4_2_4_8 2 4 8)` in Example 2–5).

  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)

The output in Example 2–5 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 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 procedure. See the Usage Notes for that procedure (in Chapter 8) 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.

See also Section 2.2.8, "Optimizing owl:sameAs Inference".

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 run-time 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 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.SEMCL_*entailment-name*, where *entailment-name* is the name of the entailment (rules index). Each MDSYS.SEMCL_*entailment-name* view has the columns shown in Table 2–3.

Table 2–3 MDSYS.SEMCL_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.SEMCL_*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.SEMCL_*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.SEMCL_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.SEMCL_M_IDX
  UNION ALL
    SELECT DISTINCT clique_id, clique_id
      FROM MDSYS.SEMCL_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 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 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 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 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'));
```

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 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 procedure. To get additional information about incremental inference for an entailment, use the SEM_APIS.GET_INC_INF_INFO 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 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 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 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 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 2–6 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 procedure.

Example 2–6 Performing Selective Inferencing

```
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 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 Section 2.1.1, 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 Section 2.1.1) 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';
```

However, many rows in the patient data table are relevant, because their diagnoses fall under this class. Example 2–7 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 Section 2.3.5.)

Example 2–7 SEM_RELATED Operator

```
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 Section 2.3.2 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 2–8). 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 2–8 expands Example 2–7 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 Section 2.1.1, the distance between `AIDS` and `Immune_System_Disorder` is 3.

Example 2–8 SEM_DISTANCE Ancillary Operator

```
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
  WHERE SEM_DISTANCE(123) <= 3;
```

Example 2–9 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.

Example 2–9 Using SEM_DISTANCE to Restrict the Number of Rows Returned

```
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 2–9, 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 2–8.

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 2–10 creates a semantic index named DIAGNOSIS_SEM_IDX on the DIAGNOSIS column of the PATIENTS table using the `Cancer_Ontology` ontology.

Example 2–10 Creating a Semantic Index

```
CREATE INDEX diagnosis_sem_idx
  ON patients (diagnosis)
  INDEXTYPE IS MDSYS.SEM_INDEXTYPE;
```

The column on which the index is built (DIAGNOSIS in Example 2–10) 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.

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 2–11 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.

Example 2–11 Creating a Semantic Index Specifying a Model and Rulebase

```
CREATE INDEX diagnosis_sem_idx
  ON patients (diagnosis)
  INDEXTYPE IS MDSYS.SEM_INDEXTYPE('ONTOLOGY_MODEL(medical_ontology),
    RULEBASE(OWLPrime)');
```

The statistical information is useful for queries that return top-k results sorted by semantic distance. Example 2–12 shows such a query.

Example 2–12 Query Benefitting from Generation of Statistical Information

```
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 2–13 creates a semantic index that uses a URI prefix.

Example 2–13 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/>)');
```

Note that 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 Oracle Database semantic technologies 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 procedure, as in the following example:

```
EXECUTE sem_apis.create_entailment('tstidx',sem_models('tst'), sem_
rulebases('skoscore'));
```

Example 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 Section 3.2.2.

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 Section 3.1.1.

3.2 Performing Inference on SKOS Models

To create an SKOS model, use the same procedure (SEM_APIS.CREATE_SEM_MODEL) 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 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 and SEM_APIS.VALIDATE_MODEL 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 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 inserts the necessary chain definition triples for S55 into an SKOS model. After the insertion, an invocation of SEM_APIS.CREATE_ENTAILMENT 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

Creating a first semantic index for documents 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 a first semantic index depends on the amount of existing data in these models.

The application program interface (API) for managing extractor policies and semantic indexes created for documents is provided in the SEM_RDFCTX PL/SQL package. Chapter 9 provides the reference information about the programs in SEM_RDFCTX package.

This chapter contains the following major sections:

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 the other. 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 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 procedure startDriver,
  member function  extractRDF(document CLOB,
                              docId    VARCHAR2) 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.

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 Section 4.10. For an example of creating a new information extractor, see Section 4.11.

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 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 Section 4.9.

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 Section 4.3.

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 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 SEM_CONTAINS_SELECT Syntax

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,
  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 `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 Section 4.5.

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 Section 4.6.

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 Section 4.2), 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 from Matched Documents (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 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 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 Section 1.10, "Semantic Data Examples (PL/SQL and Java)" 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 Section 4.13) 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;
  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 Metadata Views for Semantic Indexing

This section describes views that contain metadata about semantic indexing.

4.12.1 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 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

4.12.2 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 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
POLICY_NAME	VARCHAR2(32)	Name of the extractor policy

Table 4–2 (Cont.) MDSYS.RDFCTX_INDEX_POLICIES View Columns

Column Name	Data Type	Description
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

4.12.3 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 and one row for each exception.

Table 4–3 MDSYS.IRDFCTX_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
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

4.13 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 Section 4.10.)

```
<?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 may 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 either the Virtual Private Database feature or the Oracle Label Security option of Oracle Database:

- 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*. For information about support for VPD with semantic data, see Section 5.1.

- 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*. For information about support for OLS with semantic data, see Section 5.2

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. Chapter 10 provides reference information about the programs in the SEM_RDFSA package.

VPD and OLS support for RDF data is included in the semantic technologies support for Release 11.2. (For information about enabling, downgrading, or removing semantic technologies support, see Appendix A.)

This chapter contains the following major sections:

5.1 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.

5.1.1 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 Section 5.1.3, "Data Access Constraints"). 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.

5.1.2 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.

5.1.3 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 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.

5.1.4 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 5–1.

Table 5–1 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.

5.1.5 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 5–2.

Table 5–2 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.

Table 5–2 (Cont.) MDSYS.RDFVPD_MODELS View Columns

Column Name	Data Type	Description
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`.

5.1.6 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 5–3.

Table 5–3 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.).

5.1.7 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 5–4.

Table 5–4 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.

5.1.8 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 5–5.

Table 5–5 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.

5.2 Oracle Label Security (OLS) for RDF Data

Oracle Label Security (OLS) 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 Section 5.2.5.

The following example creates an entailment (rules index) for the contracts data using a specific rule base. 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 (VPD and OLS) Considerations in the Usage Notes for the SEM_APIS.CREATE_ENTAILMENT procedure in Chapter 7 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 procedure. To use a system-defined label generator, specify a SEM_RDFSA package constant, as explained in Section 5.2.4; 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);
```

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 creates the type body, specifying the constructor function and the getNumericLabel member function. (Application-specific logic is not included in this example.)

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;
 /
```

In Example 5–1, 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 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 5–6.

Table 5–6 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

RDF data stored in the Oracle Database semantic data store can be version-enabled using Oracle Database 11*g* 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 Section 1.3.8)

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:

- Section 6.1, "Enabling Workspace Manager Support for RDF Data"
- Section 6.2, "Version-Enabling an RDF Model"
- Section 6.3, "Inferring from Version-Enabled RDF Models"
- Section 6.4, "Merging and Refreshing Workspaces in Version-Enabled RDF Models"

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 semantic technologies support in the database (explained in Section A.1, "Enabling Semantic Technologies Support"). 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 semantic technologies support in the database, you must remove the Workspace Manager support for RDF data. (For information about downgrading and removing semantic technologies support, see Appendix A, "Enabling, Downgrading, or Removing Semantic Technologies Support".)

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 semantic technologies support in the database.

6.2 Version-Enabling an RDF Model

To version-enable an RDF model, you 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.

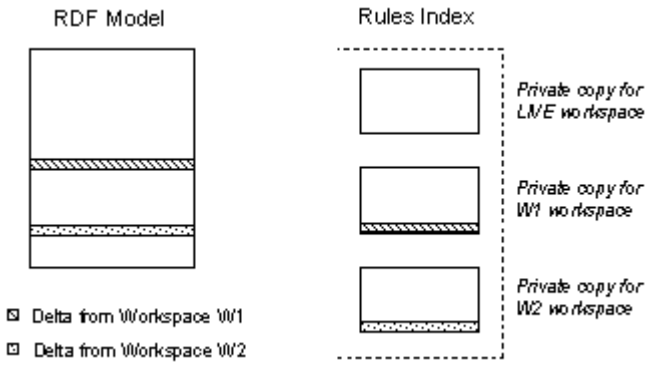
You can use Workspace Manager locks on the application table rows to avoid conflicting data manipulation operations in concurrent workspaces. The conflicts in such cases are detected based on the primary keys for the locked rows, and they may not directly translate to any meaningful locks on the RDF data, such as a lock on a sub-graph or a node. (See Section 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 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 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 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.

7

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 Chapter 1, "Oracle Database Semantic Technologies Overview" and Chapter 2, "OWL Concepts".

This chapter provides reference information about the subprograms, listed in alphabetical order.

SEM_APIS.ADD_SEM_INDEX

Format

```
SEM_APIS.ADD_SEM_INDEX(
   index_code     IN VARCHAR2);
```

Description

Creates 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 Section 1.8.

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');
```

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 Section 1.3.7.

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');
```

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 Section 1.2, "Semantic Data Modeling" and Section 1.3, "Semantic Data in the Database".

Examples

The following example moves the model named `family` to the tablespace named `my_tbs`.

```
EEXECUTE SEM_APIS.ALTER_MODEL('family', 'MOVE',  'my_tbs');
```

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 Section 1.8.

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');
```

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 Section 1.8.

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');
```

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*.

For information about entailments, see Section 1.3.7.

Examples

The following example collects statistics for the entailment named `rdfs_rix_family`.

```
EXECUTE SEM_APIS.ANALYZE_ENTAILMENT('rdfs_rix_family');
```

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*.

Examples

The following example collects statistics for the semantic model named `family`.

```
EXECUTE SEM_APIS.ANALYZE_MODEL('family');
```

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 BINARY_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:

- `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.
- *<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 Section 1.7.1 for more information.

Using BULK_LOAD_FROM_STAGING_TABLE with Fine Grained Access Control (VPD and OLS)

When fine-grained access control (explained in Chapter 5) 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 Section 5.1.1). 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 and SEM_RDFSA.SET_PREDICATE_LABEL).

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');
```

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 Section 1.3.6) or the SDO_RULES_INDEX_INFO view (described in Section 1.3.7), 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');
```

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 Section 1.3.2) 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 Section 1.3.2) 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 Section 1.3.2) 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 Section 1.3.2) 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 Section 1.3.2) 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>
```

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);
```

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 Section 1.3.6.

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`, `DISTANCE`, `DOP`, `ENTAIL_ANYWAY`, `HASH_PART`, `INC`,

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 Section 5.2.4.
- If you use a custom label generator, specify the custom label generator type. For information about creating and implementing a custom label generator, see Section 5.2.5.

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 7–1, 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 7–1. 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 7–1 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 Section 3.2.2.) (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.
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.

Table 7–1 (Cont.) Inferencing Keywords for inf_components_in Parameter

Keyword	Explanation
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.)
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.
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.)

Table 7–1 (Cont.) Inferencing Keywords for inf_components_in Parameter

Keyword	Explanation
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 uses hybrid columnar compression to create 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.

  Note: You can specify `COL_COMPRESS=T` only if the storage is on Exadata Storage Server.

- `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 Section 2.2.10.
- `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 Section 2.2.9.

- 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 Section 2.2.8.
- RAW8=T uses RAW8 datatypes 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 7–2.

Table 7–2 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 (VPD and OLS) 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 Section 5.1.1, "VPD Policy for RDF Data"). 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 Section 5.1.1, 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. --
```

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 Section 1.3.6. The SEM_MATCH table function is described in Section 1.6,

Examples

The following example creates a rulebase named family_rb. (It is an excerpt from Example 1–20 in Section 1.10.2.)

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

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 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 Section 1.9.

This procedure adds the model to the MDSYS.SEM_MODEL$ view, which is described in Section 1.3.1.

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 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 1–19 in Section 1.10.2.)

```
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);
```

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 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 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');
```

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);
```

Description

Creates a virtual model containing the specified semantic models and rulebases.

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)`. At least one semantic model must be specified.

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 Section 1.3.6.

options
(Reserved for future use.)

Usage Notes

For an explanation of virtual models, including usage information, see Section 1.3.8.

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.

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 Section 1.3.8 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'));
```

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 Section 2.2.9.

Examples

The following example disables change tracking for the `family` model.

```
EXECUTE sem_apis.disable_change_tracking(sem_models('family'));
```

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 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 Section 2.2.9.

Examples

The following example enables incremental inference for the entailment named `RDFS_RIX_FAMILY`.

```
EXECUTE sem_apis.disable_inc_inference('rdfs_rix_family');
```

SEM_APIS.DROP_ENTAILMENT

Format

```
SEM_APIS.DROP_ENTAILMENT(
   entailment_name_in  IN VARCHAR2);
```

Description

Drops (deletes) an entailment (rules index).

Parameters

entailment_name_in
Name of the entailment to be deleted.

Usage Notes

You can use this procedure to delete an entailment that you created using the SEM_APIS.CREATE_ENTAILMENT procedure.

Examples

The following example deletes a entailment named `OWLTST_IDX`.

```
EXECUTE sem_apis.drop_entailment('owltst_idx');
```

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 Section 1.3.6.

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');
```

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 procedure.

Usage Notes

For an explanation of semantic network indexes, see Section 1.8.

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');
```

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 Section 1.3.1.

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');
```

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;
```

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.
```

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 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 Section 1.3.8.

Examples

The following example deletes a virtual model named `VM1`.

```
EXECUTE sem_apis.drop_virtual_model('VM1');
```

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 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 procedure.

For an explanation of incremental inference, including usage information, see Section 2.2.9.

Examples

The following example enables change tracking for the `family` model.

```
EXECUTE sem_apis.enable_change_tracking(sem_models('family'));
```

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 procedure to enable change tracking on those models.

To disable incremental inference for an entailment, use the SEM_APIS.DISABLE_INC_INFERENCE procedure.

For an explanation of incremental inference, including usage information, see Section 2.2.9.

Examples

The following example enables incremental inference for the entailment named `RDFS_RIX_FAMILY`.

```
EXECUTE sem_apis.enable_inc_inference('rdfs_rix_family');
```

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 Section 1.3.1.

To enable change tracking for a set of models, use the SEM_APIS.ENABLE_CHANGE_TRACKING procedure.

For an explanation of incremental inference, including usage information, see Section 2.2.9.

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;
/
```

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 procedure.

For an explanation of incremental inference, including usage information, see Section 2.2.9.

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
/
```

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 Section 1.3.1.

Examples

The following example returns the model ID number for the model named `articles`. (This example is an excerpt from Example 1–19 in Section 1.10.2.)

```
SELECT SEM_APIS.GET_MODEL_ID('articles') AS model_id FROM DUAL;

  MODEL_ID
----------
         1
```

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 Section 1.3.1.

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 1–19 in Section 1.10.2.)

```
SQL> SELECT SEM_APIS.GET_MODEL_NAME(1) AS model_name FROM DUAL;

MODEL_NAME
--------------------------------------------------------------------------------
ARTICLES
```

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 Section 1.3.1.

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 Section 1.3.1.

subject
Subject. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Section 1.3.2.

property
Property. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Section 1.3.2.

object
Object. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Section 1.3.2.

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 1–19 in Section 1.10.2.)

```
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
```

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 Section 1.3.1.

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 Section 1.3.1.

subject
Subject. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Section 1.3.2.

property
Property. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Section 1.3.2.

object
Object. Must match a value in the VALUE_NAME column of the MDSYS.RDF_VALUE$ table, which is described in Section 1.3.2.

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 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 1–19 in Section 1.10.2.)

```
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)
```

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 Section 1.3.6.

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 1–20 in Section 1.10.2.)

```
SELECT SEM_APIS.LOOKUP_ENTAILMENT(SEM_MODELS('family'),
  SEM_RULEBASES('RDFS','family_rb')) AS lookup_entailment FROM DUAL;

LOOKUP_ENTAILMENT
--------------------------------------------------------------------------------
RDFS_RIX_FAMILY
```

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 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 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 versioning-enabled semantic models.
- This procedure is not supported on virtual models (explained in Section 1.3.8).
- 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');
```

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 Section 1.3.8).

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.

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');
```

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

(None.)

Examples

The following example renames a entailment named `OWLTST_IDX` to `MY_OWLTST_IDX`.

```
EXECUTE sem_apis.rename_entailment('owltst_idx', 'my_owltst_idx');
```

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 versioning-enabled semantic models.
- This procedure is not supported on virtual models (explained in Section 1.3.8).

Contrast this procedure with SEM_APIS.SWAP_NAMES, 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');
```

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.

Contrast this procedure with SEM_APIS.RENAME_MODEL, 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');
```

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 Section 1.3.6.

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 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 Section 2.2.5.

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 procedure.

Examples

For an example of this procedure, see Example 2–5 in Section 2.2.5.

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 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 Section 2.2.5.

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;
```

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 Section 1.3.2.

value_type
Value type. Must match a value in the VALUE_TYPE column in the MDSYS.RDF_VALUE$ table, which is described in Section 1.3.2.

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.
```

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 Section 1.3.2.

value_type
Value type. Must match a value in the VALUE_TYPE column in the MDSYS.RDF_VALUE$ table, which is described in Section 1.3.2.

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
```

8

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 Chapter 1, "Oracle Database Semantic Technologies Overview" and Chapter 2, "OWL Concepts".

This chapter provides reference information about the subprograms, listed in alphabetical order.

SEM_PERF.GATHER_STATS

Format

```
SEM_PERF.GATHER_STATS();
   just_on_values_table  IN BOOLEAN DEFAULT FALSE,
   parallel              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).

parallel
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 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 Section 2.2.7. 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.

Examples

The following example gathers statistics about RDF and OWL related tables and their indexes.

```
EXECUTE SEM_PERF.GATHER_STATS;
```

9

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 Chapter 4, "Semantic Indexing for Documents".

This chapter provides reference information about the subprograms, listed in alphabetical order.

SEM_RDFCTX.CREATE_POLICY

Format

```
SEM_RDFCTX.CREATE_POLICY(
   policy_name  IN VARCHAR2,
   extractor       mdsys.rdfctx_extractor,
   preferences   sys.XMLType DEFAULT NULL);
```

or

```
SEM_RDFCTX.CREATE_POLICY(
   policy_name   IN VARCHAR2,
   base_policy    IN VARCHAR2,
   user_models  mdsys.rdf_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.

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 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 Section 4.8, "Indexing External Documents".

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;
/
```

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;
/
```

SEM_RDFCTX.MAINTAIN_TRIPLES

Format

```
SEM_RDFCTX.MAINTAIN_TRIPLES(
   index_name      IN VARCHAR2,
   where_clause    IN VARCHAR2,
   rdfxml_content  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;
/
```

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 Section 4.3, "Semantically Indexing Documents", 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;
/
```

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 Section 4.9), 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 Section 4.10), 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:

- Section 4.9, "Configuring the Calais Extractor type"

- Section 4.10, "Working with General Architecture for Text Engineering (GATE)"

10

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 Chapter 1, "Oracle Database Semantic Technologies Overview" and Chapter 5, "Fine-Grained Access Control for RDF Data".

This chapter provides reference information about the subprograms, listed in alphabetical order.

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 Section 5.1.1). A constraint with no constraint group is always active. See Section 5.1.3 for details about the match and apply patterns passed to this procedure.

See also the MDSYS.RDFVPD_POLICY_CONSTRAINTS view, which is described in Section 5.1.6.

For information about support for VPD, see Section 5.1.

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;
/
```

SEM_RDFSA.APPLY_OLS_POLICY

Format

```
SEM_APIS.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 10–1 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_APIS.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 10–1 lists these constants and their descriptions.

Table 10–1 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.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 procedure.

For information about support for OLS, see Section 5.2.

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;
/
```

SEM_RDFSA.APPLY_VPD_POLICY

Format

```
SEM_APIS.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 Section 5.1.4.

For information about support for VPD, see Section 5.1.

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;
/
```

SEM_RDFSA.CREATE_VPD_POLICY

Format

```
SEM_APIS.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 procedure and the policy's data access constraints defined using the SEM_RDFSA.ADD_VPD_CONSTRAINT procedure.

The name of the application context assigned to the `policy_context` parameter is used to manage the constraint groups dynamically at run time. 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 run time, 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 Section 5.1.1).

See also the MDSYS.RDFVPD_POLICIES view, which is described in Section 5.1.4.

For information about support for VPD, see Section 5.1.

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;
/
```

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 Section 5.1.6.

For information about support for VPD, see Section 5.1.

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_policy(
      policy_name   => 'contracts_policy',
      constr_name   => 'vp_access_to_contr_value');
end;
/
```

SEM_RDFSA.DISABLE_OLS_POLICY

Format

SEM_APIS.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 procedure; to enable an OLS policy that had been disabled, use the SEM_RDFSA.ENABLE_OLS_POLICY procedure.

For information about support for OLS, see Section 5.2.

Examples

The following example disables the OLS policy for the semantic data store.

```
begin
  sem_rdfsa.disable_ols_policy;
end;
/
```

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 Section 5.1.

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_policy(
      policy_name   => 'contracts_policy',
      constr_name   => 'vp_access_to_contr_value');
end;
/
```

SEM_RDFSA.ENABLE_OLS_POLICY

Format

SEM_APIS.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 procedure.

For information about support for OLS, see Section 5.2.

Examples

The following example enables the OLS policy for the semantic data store.

```
begin
  sem_rdfsa.enable_ols_policy;
end;
/
```

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 Section 5.1.7 and Section 5.1.8, respectively.

For information about support for VPD, see Section 5.1.

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;
```

SEM_RDFSA.REMOVE_OLS_POLICY

Format

SEM_APIS.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 procedure.

For information about support for OLS, see Section 5.2.

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;
/
```

SEM_RDFSA.REMOVE_VPD_POLICY

Format

```
SEM_APIS.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 Section 5.1.

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;
/
```

SEM_RDFSA.RESET_MODEL_LABELS

Format

```
SEM_APIS.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 Section 5.2.

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;
/
```

SEM_RDFSA.SET_PREDICATE_LABEL

Format

```
SEM_APIS.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 Section 5.2.

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;
/
```

SEM_RDFSA.SET_RDFS_LABEL

Format

```
SEM_APIS.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 Section 5.2.

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;
/
```

SEM_RDFSA.SET_RESOURCE_LABEL

Format

```
SEM_APIS.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 Section 5.2.

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;
/
```

SEM_RDFSA.SET_RULE_LABEL

Format

```
SEM_APIS.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 Section 5.2.

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 Semantic Technologies Support

This appendix describes the required steps that you must perform before you can use any types, synonyms, or PL/SQL packages related to Oracle semantic technologies support in the current Oracle Database release. You must run one or more scripts, and you must ensure that Spatial is installed and the Partitioning option is enabled. These requirements are explained in Section A.1.

This appendix also describes the steps if, after enabling semantic technologies support, you need to do any of the following:

- Downgrade the semantic technologies support to that provided with a previous Oracle Database release, as explained in Section A.2.
- Remove all support for semantic technologies from the database, as explained in Section A.3.

A.1 Enabling Semantic Technologies Support

Before you can use any types, synonyms, or PL/SQL packages related to Oracle semantic technologies 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:

- Section A.1.1, "Enabling Semantic Technologies Support in a New Database Installation"
- Section A.1.2, "Upgrading Semantic Technologies Support from Release 11.1"
- Section A.1.3, "Upgrading Semantic Technologies Support from Release 10.2"

In addition, you must ensure that Spatial is installed and the Partitioning option is enabled, as explained in Section A.1.4.

A.1.1 Enabling Semantic Technologies Support in a New Database Installation

To enable semantic technologies support in a new installation of Oracle Database Release 11.2 (or for an upgrade from Release 10.2 if no RDF network exists, as explained in Section A.1.3), 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. 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 semantic technologies 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: `112`
- 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 Section 6.1, "Enabling Workspace Manager Support for RDF Data".

A.1.2 Upgrading Semantic Technologies Support from Release 11.1

If you are upgrading from Oracle Database Release 11.1 that includes the semantic technologies support for Release 11.1, the semantic technologies support is automatically upgraded to Release 11.2 when the database is upgraded. If you use Workspace Manager with RDF data, you must also run the appropriate script *after* the upgrade of semantic technologies support, as explained in Section 6.1.

However, you will also need to migrate RDF data if you have an existing Release 11.1 RDF network containing triples that include any of the following:

- Typed literal values of type `xsd:float` or `xsd:double`
- Literal values with a zero-length string as the value component

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', 'NULL_LITERAL');
```

If the FLOAT_DOUBLE_DECIMAL and NULL_LITERAL attributes have the string value INVALID, you need to migrate RDF data. If the FLOAT_DOUBLE_DECIMAL and NULL_LITERAL attributes do not exist or have the string value VALID, you do not need to migrate RDF data, and you can skip the rest of 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).
2. Start SQL*Plus, and enter the following statement:

   ```
   EXECUTE sdo_rdf_internal.convert_old_rdf_data;
   ```

   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` or `xsd:double` as the literal type or that are using a zero-length string as the object component.
3. 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 11.2.

A.1.3 Upgrading Semantic Technologies Support from Release 10.2

If you are upgrading from Oracle Database Release 10.2 that includes the semantic technologies support for Release 10.2, the actions required depend on whether there is an existing RDF network.

If there is not an existing RDF network, skip the rest of this section and follow the instructions in Section A.1.1, "Enabling Semantic Technologies Support in a New Database Installation".

If there is an existing RDF network, perform the following steps after the Oracle Database upgrade:

1. Connect to the database (Release 11.2) 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/catsem10i.sql`
   - Windows: `@%ORACLE_HOME%\md\admin\catsem10i.sql`

   The `catsem10i.sql` script enables Release 11.2 semantic technologies support and migrates RDF data from Release 10.2 to Release 11.2 format. The script may take a long time to run if the existing Release 10.2 RDF network contains a large amount of RDF data.

A.1.4 Spatial and Partitioning Requirements

Oracle Spatial must be installed before you can use any of the RDF and OWL capabilities. Oracle Locator is not sufficient. For information about Spatial and Locator, see *Oracle Spatial 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 Semantic Technologies Support to a Previous Release

This section explains how to downgrade the semantic technologies support, in conjunction with an Oracle Database downgrade to Release 11.1 or 10.2.

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 semantic technologies support in the database, as explained in Section 6.1.1.

A.2.1 Downgrading to Release 11.1 Semantic Technologies Support

If you need to downgrade to Oracle Database Release 11.1, the semantic technologies component will be downgraded automatically when you downgrade the database. However, any RDF or OWL data that is specific to Release 11.2 (that is, Release 11.2 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 11.1.

Any virtual models will not be preserved during a downgrade, because they are only available through a Release 11.1.0.7.0 patch (number 7600122) that was made available on My Oracle Support (MetaLink) in November, 2008. To use virtual models, you must reapply this patch after the downgrade, and then re-create the virtual models. For convenience, you can use the `semvmrecreate.sql` script to generate a script that you can later use (after the downgrade) to automatically re-create any virtual models that were present in the Release 11.2 semantic network. To do so, follow these steps before you downgrade the database :

1. Connect to the database (Release 11.2) as the SYS user with SYSDBA privileges (SYS AS SYSDBA, and enter the SYS account password when prompted).
2. Start SQL*Plus, and enter a statement in the following format:
   - Linux: `@$ORACLE_HOME/md/admin/semvmrecreate.sql /`*my_dir_path*`/my_vm_recreate.sql`
   - Windows: `@%ORACLE_HOME%\md\admin\semvmrecreate.sql \`*my_dir_path*`\my_vm_recreate.sql`

   where *my_dir_path* is the path to the location in which to create the `my_vm_recreate.sql` script file. (You can use another file name for the script, if you want.)

To check if any Release 11.2 RDF data is incompatible with Release 11.1, perform the following steps:

1. Connect to the database (Release 11.2) 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_111_COMPATIBLE;
```

If any RDF data is incompatible with Release 11.1, the procedure generates an error and displays a list of the incompatible data. In this case, you must run the `semrelod111.sql` script after the database downgrade, as follows:

1. In the Release 11.2 database, save a copy of the `semrelod111.sql` file, which is in the following location:
   - Linux: `$ORACLE_HOME/md/admin/semrelod111.sql`
   - Windows: `%ORACLE_HOME%\md\admin\semrelod111.sql`
2. If you have any virtual models that you want to be able to re-create after the download, run the `semvmrecreate.sql` script (if you have not already done so), as explained earlier in this section.
3. Remove any Release 11.2 release-specific RDF or OWL data (including virtual models) if you have not already done so, as explained earlier in this section.
4. Perform the database downgrade.

5. In the Release 11.1 database, copy the saved copy of the `semrelod111.sql` file to the following location:
   - Linux: `$ORACLE_HOME/md/admin/`
   - Windows: `%ORACLE_HOME%\md\admin\`
6. Connect to the Release 11.1 database as the SYS user with SYSDBA privileges (SYS AS SYSDBA, and enter the SYS account password when prompted).
7. Start SQL*Plus, and enter the following statement:
   - Linux: `@$ORACLE_HOME/md/admin/semrelod111.sql`
   - Windows: `@%ORACLE_HOME%\md\admin\semrelod111.sql`

   After the `semrelod111.sql` script completes successfully, Oracle semantic technologies support for Release 11.1 is enabled and ready to use, and all Release 11.1-compatible data is preserved.
8. If you want to enable support for virtual models, apply the Release 11.1.0.7 interim patch number 7600122.
9. If you previously generated a `my_vm_recreate.sql` script to re-create your virtual models after the downgrade, go to the *my_dir_path* location, connect to the database as the SYS user with SYSDBA privileges, start SQL*Plus, and run the `my_vm_recreate.sql` script.

A.2.2 Downgrading to Release 10.2 Semantic Technologies Support

If you need to downgrade to Oracle Database Release 10.2, follow the instructions in Section A.2.1 to downgrade to Release 11.1, and then follow the instructions in the rest of this section (taken from the Release 11.1 manual) to downgrade from Release 11.1 to Release 10.2.

If you need to downgrade to Oracle Database Release 10.2, and if you used Oracle Database Release 11 RDF or OWL features and want to preserve existing semantic data and rulebases, you must execute a statement to prepare for the downgrade, perform the downgrade, and execute another statement to restore the semantic data.

However, the following considerations apply:

- Entailed graph data will not be preserved, because the same information can be regenerated using the Oracle Database Release 10.2 RDF inference API.
- No rulebases or rules indexes related to OWL are preserved, because Oracle Database Release 10.2 did not support the OWL vocabulary.

Perform the following steps:

1. Before the database downgrade, connect to the Release 11 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:

   ```
   EXECUTE SDO_SEM_DOWNGRADE.PREPARE_DOWNGRADE_FROM_11;
   ```

   When this statement executes successfully, all existing semantic data and rulebases are saved. You will restore the semantic data after the database downgrade.
3. Perform the database downgrade.

4. Download the following file from the Semantic Technologies page of the Oracle Technology Network site: `sdosemdgu.plb`
5. If (and only if) your Oracle Database Release 10.2 release number is 10.2.0.1, click the Software link, and download and install the RDF-specific patch. (This patch is needed because Release 10.2.0.1 did not have the batch loading feature, which is used to restore the semantic data.)
6. Connect to the Release 10.2 database as the SYS user with SYSDBA privileges.
7. Start SQL*Plus and enter a statement in the following statement:

   ```
   EXECUTE SDO_SEM_DOWNGRADE_UTL.PREPARE_DOWNGRADE_TO_102('<tablespace-name>');
   ```

   Where <*tablespace-name*> is the name of the tablespace in which the RDF network will be created.

   When this statement executes successfully, all semantic data that had been saved before the downgrade is restored and ready to use.

A.3 Removing Semantic Technologies Support

This section explains how to remove the semantic technologies 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 Oracle semantic technologies support.

If you have enabled Workspace Manager support for RDF data, you must remove that support *before* you remove semantic technologies support from the database, as explained in Section 6.1.1.

To remove the semantic technologies support from the database, perform the following steps:

1. Connect to the database (Release 11.2) 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 semantic technologies types, tables, and PL/SQL packages:

Glossary

The following are selected terms relevant to the Oracle Database implementation of semantic technologies 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

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 via 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.

Jena Adaptor for Oracle Database

An Oracle-supplied adaptor (available for download) for Jena, which is a Java framework for building Semantic Web applications.

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 Oracle Database semantic technologies native inferencing engine.

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

A

B

C

D

P

Q

R

S

T

U

V

W