OPENLINK SOFTWARE
Making Technology Work For You

Mapping Relational Data to RDF with Virtuoso's RDF Views

Among its many talents, OpenLink Virtuoso Universal Server includes SPARQL support and an RDF data store tightly integrated with its relational storage engine. This article provides an overview of how to use Virtuoso to dynamically convert relational data into RDF and expose it from a Virtuoso-hosted SPARQL endpoint.

The Resource Description Framework (RDF) forms a fundamental building block in the Semantic Web vision, providing a mechanism for conceptually modeling web data. Today the vast bulk of data held by companies resides in relational databases and, as a result, data that ultimately reaches the web is inherently heterogeneous at both the data schema and DBMS engine levels. Thus a key infrastructural requirement of the Semantic Web vision is the existence of technology that facilitates the generation of RDF views of relational data. OpenLink's Virtuoso provides such a capability through its RDF Views support. This tutorial provides an example of building RDF views of SQL data.

What are RDF Views?

Virtuoso's RDF Views map relational data into RDF and allow the RDF representation of the relational data to be customised. Virtuoso includes a declarative Meta Schema Language for defining the mapping of SQL data to RDF ontologies. The mapping is dynamic; consequently changes to the underlying data are reflected immediately in the RDF representation. No changes are required to the underlying relational schema - so minimising disruption to a critical company asset.

The Simplest Conceptual View of Relational-RDF Mapping

At the most basic level, RDF Views transform the resultset of a SQL SELECT statement into a set triples. The transformations required to synthesize each RDF graph are described by a *quad map pattern*. Collectively, the defined quad map patterns constitute an *RDF meta schema*. Before describing how these transformations are defined, it's useful to picture in general terms how SQL data is transformed to RDF.

Consider first the Entity-Relationship model beloved by database designers for decades as a basis for designing database schemas. Typically, each entity is represented as a table, each attribute of the entity becomes a column in that table, and relationships between entities are indicated by foreign keys. Each table typically defines a particular class of entity, each column one of its attributes. Each row in the table describes an entity instance, uniquely identified by a primary key.

Now consider RDF. The constituent parts of a triple could be described as: *subject*, *subject attribute* and *attribute value*, rather than the usual *subject*, *predicate*, *object*. A thorough description of an entity in RDF would consist of a collection of triples each relating to the same subject, i.e. entity, where each predicate relates to a different entity attribute. The entity's class would be described by a further RDF statement containing the rdf:type (or 'a') predicate. To uniquely identify each distinct subject and predicate, the subject (ignoring blank nodes for now) and predicate must be IRI's, the object can be an IRI or a literal.

The parallels between the relational and RDF descriptions of an entity are clear. In the simplest case, any relational schema can be rendered into RDF by converting all primary keys and foreign keys into IRI's, assigning a predicate IRI to each column, and an rdf:type predicate for each row linking it to a RDF class IRI corresponding to the table. We can then, for each column that is neither part of a primary or foreign key, construct a triple containing the primary key IRI as the subject, the column IRI as the predicate and the column's value as the object.

The Scenario - An Online Product Catalog

That's the theory in a nutshell. So, how do we define an RDF view? For our example, suppose we want to make OpenLink's product portfolio available online as an RDF dataset and that the product descriptions are currently held in relational tables.

Table: PRODUCT		
PRODUCT_ID	VARCHAR(25)	PRIMARY KEY
PRODUCT_DESCRIPTION	VARCHAR(125)	
PRODUCT_CAT_ID	INTEGER	(FK)
PRODUCT_FORMAT_ID	INTEGER	(FK)

Table: PRODUCT_CATEGORY

Table: PRODUCT_CATEGORY		
PRODUCT_CAT_ID	INTEGER	PRIMARY KEY
PRODUCT_CATEGORY_DESCRIPTION	VARCHAR(50)	

Table: PRODUCT_FORMAT		
PRODUCT_FORMAT_ID	INTEGER	PRIMARY KEY
PRODUCT_FORMAT_DESCRIPTION	VARCHAR(75)	

Our Relational Model

A subset of our relational schema could comprise the following three Virtuoso tables, PRODUCT, PRODUCT_CATEGORY and PRODUCT_FORMAT. Some representative data for each table is listed below. (The sample data shown is for illustrative purposes only, is purposely very limited and is not intended to reflect real-world data.) In the examples which follow, the tables are assumed to belong to user DBA in catalog OPLWEB.

Sample Data - Table: PRODUCT_CATEGORY	
PRODUCT_CAT_ID	PRODUCT_CATEGORY_DESCRIPTION
1	ODBC Drivers
2	JDBC Drivers
3	OLEDB Data Providers
4	ADO.NET Data Providers

Sample Data - Table: PRODUCT_FORMAT	
PRODUCT_FORMAT_ID	PRODUCT_FORMAT_DESCRIPTION
1	Enterprise
2	Single-Tier (Lite Edition)
3	Multi-Tier (Enterprise Edition)
4	Single-Tier (Express Edition)

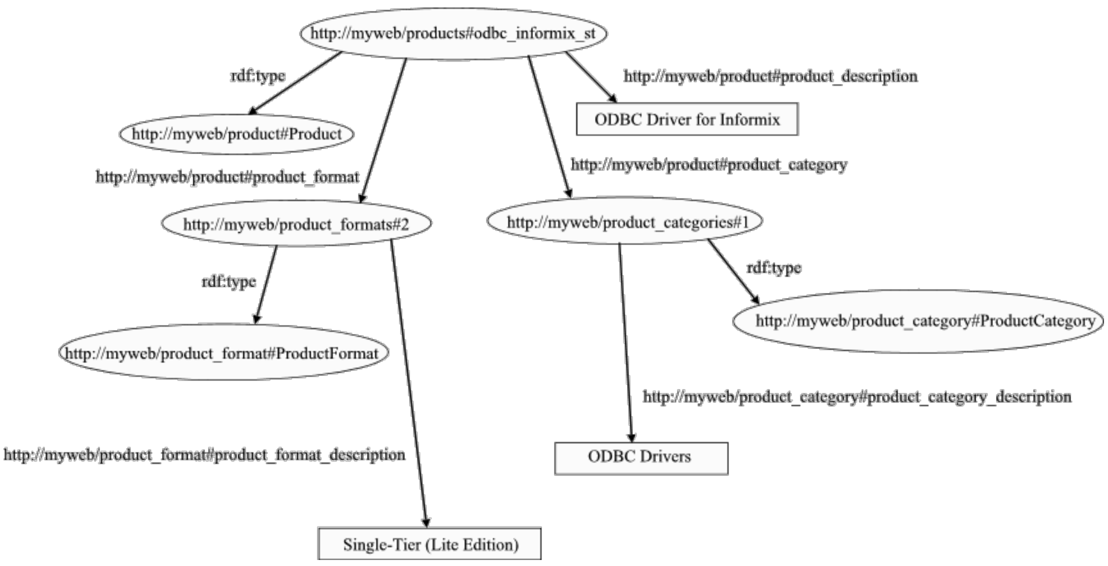
Sample Data - Table: PRODUCT			
PRODUCT_ID	PRODUCT_DESCRIPTION	PRODUCT_CAT_ID	PRODUCT_FORMAT_ID
odbc-informix-ee	ODBC Drivers for Informix	1	4
odbc-informix-mt	ODBC Driver for Informix	1	3
odbc-informix-st	ODBC Driver for Informix	1	2
jdbc-ingres-mt	JDBC Driver for Ingres	2	3
oledb-odbc-st	OLE DB Provider for ODBC	3	2

Sample Data - Table: PRODUCT			
dotnet-postgres-mt	.NET Data Provider for PostgreSQL	4	3

Our Target RDF Model

The figure below shows a representative RDF graph depicting the RDF data model we are aiming to construct using RDF views. Nodes that are IRIs are shown as ellipses, while nodes that are literals are shown as boxes.

(Click to enlarge)



http://myweb/products#odbc_informix_st
rdf:type
http://myweb/product#product_description
ODBC Driver for Informix
http://myweb/product#Product
http://myweb/product#product_category
http://myweb/product#product_format
http://myweb/product_formats#2
http://myweb/product_categories#1
rdf:type
rdf:type
http://myweb/product_category#ProductCategory
http://myweb/product_format#ProductFormat
http://myweb/product_category#product_category_description
http://myweb/product_format#product_format_description
ODBC Drivers
Single-Tier (Lite Edition)

Key Steps

Based on the mapping process outlined above, some of the key requirements identified are:

- Definition of an RDF class IRI for each table
- Construction of a subject IRI for each primary key column value
- Construction of a predicate IRI for each non-key column

Defining our Resource Classes Using RDF Schema

Resource classes and the relationships between the resources can be expressed using RDF Schema (RDFS), a semantic extension to RDF. Listing 1 shows an RDFS class, Product, defined in N3 for the PRODUCT table. The RDF class IRI for the table is <http://localhost:8890/rdfv_demo/schemas/product#Product>.

Listing 1:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

@prefix prd: <http://localhost:8890/rdfv_demo/schemas/product#> .
@prefix prdf: <http://localhost:8890/rdfv_demo/schemas/product_format#> .
@prefix prdc: <http://localhost:8890/rdfv_demo/schemas/product_category#>

prd:Product a rdfs:Class ;
  rdfs:label "Product" ;
  rdfs:comment "An OpenLink product" .

prd:product_id a rdf:Property ;
  rdfs:domain prd:Product ;
  rdfs:range xsd:string ;
  rdfs:label "product id" .

prd:product_description a rdf:Property;
  rdfs:domain prd:Product ;
  rdfs:range xsd:string ;
  rdfs:label "product description" .

prd:product_category a rdf:Property ;
  rdfs:domain prd:Product ;
  rdfs:range prdc:ProductCategory ;
  rdfs:label "product category id" .

prd:product_format a rdf:Property ;
  rdfs:domain prd:Product ;
  rdfs:range prdf:ProductFormat ;
  rdfs:label "product format" .
```

Notice that attributes corresponding directly to the foreign keys PRODUCT_CAT_ID and PRODUCT_FORMAT_ID in the PRODUCT table are not included in the RDFS Product class. Had we wanted to include them, they would each have had an rdfs:range of xsd:integer. The relationships between the tables linked by the foreign keys can instead be made more explicit by referencing the relevant classes directly from the Product class, in effect dereferencing the foreign keys. To this end, the attributes product_category and product_format have ranges prdc:ProductCategory and prdf:ProductFormat respectively, where ProductCategory and ProductFormat are RDFS classes corresponding to the entities described by the PRODUCT_CATEGORY and PRODUCT_FORMAT tables. Listing 2 shows both resource class definitions.

Listing 2:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

@prefix prdf: <http://localhost:8890/rdfv_demo/schemas/product_format#> .
@prefix prdc: <http://localhost:8890/rdfv_demo/schemas/product_category#>

prdf:ProductFormat a rdfs:Class ;
  rdfs:label "ProductFormat" ;
  rdfs:comment "An OpenLink product format" .

prdf:product_format_id a rdf:Property ;
  rdfs:domain prdf:ProductFormat ;
  rdfs:range xsd:integer ;
  rdfs:label "product format id" .

prdf:product_format_description a rdf:Property;
  rdfs:domain prdf:ProductFormat ;
  rdfs:range xsd:string ;
  rdfs:label "product format description" .

prdc:ProductCategory a rdfs:Class ;
  rdfs:label "ProductCategory" ;
  rdfs:comment "An OpenLink product category" .

prdc:product_cat_id a rdf:Property ;
  rdfs:domain prdc:ProductCategory ;
  rdfs:range xsd:integer ;
  rdfs:label "product category id" .

prdc:product_category_description a rdf:Property;
  rdfs:domain prdc:ProductCategory ;
  rdfs:range xsd:string ;
  rdfs:label "product category description" .
```

Creating IRIs From Primary Keys

IRI Classes

Virtuoso manages the conversion of column values to IRIs using IRI classes, a feature of Virtuoso's Meta Schema Language which itself is implemented as extensions to SPARQL. An IRI class defines how a column or set of columns gets converted into an IRI. Listing 3 shows the definition of an IRI class for converting the primary key of the PRODUCT table into an IRI. The listing is a SPARQL snippet which could, for instance, be passed to Virtuoso's *isql* command line utility for execution. In fact, SPARQL can be used inline wherever SQL can be used; the only requirement being the inclusion the *sparql* keyword at the start.

Listing 3:

```
sparql

prefix prd: <http://localhost:8890/rdfv_demo/schemas/product#>
create iri class prd:product_iri "http://localhost:8890/rdfv_demo/testdat
    (in product_id varchar not null) .
```

The example illustrates the use of a sprintf–style format string for performing conversion. In addition to %s, other format specifiers are also supported, for example %d. The reverse conversion is inferred automatically. For more complex conversions, it is possible to specify functions that assemble and disassemble an IRI from/into its constituent parts, as shown in Listing 4. Here the functions PRODUCT_URI and PRODUCT_URI_INVERSE would be defined elsewhere (not shown), for instance in Virtuoso/PL, Virtuoso's SQL procedure language.

Listing 4:

```
create iri class prd:product_iri using
  function OPLWEB.DBA.PRODUCT_URI (in id varchar)
    returns varchar,
  function OPLWEB.DBA.PRODUCT_URI_INVERSE (in id_iri varchar)
    returns varchar .
```

Creating RDF Literals From Non-Key Column Values

Literal Classes

IRI classes define the conversion of primary key values into subject IRIs. For columns which are neither part of a primary or foreign key, the column value will normally form the object of a triple. While RDF mandates that the subject and predicate must be IRIs, the object can be an IRI or a literal. As an adjunct to IRI classes, Virtuoso's Meta Schema Language also supports *literal classes*, which define how a column or set of columns gets converted into a literal. A special case of literal class is the *identity class* that converts a value from a SQL varchar column into an untyped literal and a value from a column of any other SQL datatype into an XML Schema typed literal i.e. xsd:integer, xsd:dateTime and so on.

Identity classes are a special case in that you, as a developer, need not define them or refer to them explicitly. They are built into Virtuoso and are invoked implicitly, as we shall see when we look at the key element in Virtuoso's RDF Meta Schema infrastructure, *quad map patterns*.

Quad Map Patterns & Quad Storage

Simple Quad Map Patterns

A quad map pattern defines one particular transformation from one set of relational columns into triples that match one SPARQL graph pattern. The main part of a quad map pattern is four declarations of "quad map values" specifying how to derive the value of a triple field from the SQL data using, if necessary, IRI classes and literal classes to assist in the transformation. The four declarations correspond to the graph, subject, predicate and object components of the query pattern. The quad map pattern can also include boolean SQL expressions to filter out unwanted rows of source data or to join tables if the source columns belong to different tables.

The following declaration shows a simple quad map pattern using the IRI class product_iri to map the product's ID into an IRI.

Listing 5:

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix prd: <http://localhost:8890/rdfv_demo/schemas/product#>

graph <http://localhost:8890/rdfv_demo/testdata/products#>
subject prd:product_iri(OPLWEB.DBA.PRODUCT.PRODUCT_ID)
predicate rdf:type
object prd:Product
```

The meta schema description language also supports a more concise SPARQL–like notation which requires fewer keywords and eliminates duplicate graphs, subjects and predicates. The next example expands on the previous one by rewriting it using the shorter notation and adding three more patterns sharing the same graph and subject IRIs. The new pattern declarations introduce the prd:product_description, prd:product_category and prd:product_format predicates. The object portion of the prd:product_description pattern implicitly uses the identity literal class to convert the column value into an RDF literal. Notice that it isn't necessary to name the identity literal class, instead the column name, PRODUCT_DESCRIPTION, is used directly.

Listing 6:

```
sparql

prefix prd: <http://localhost:8890/rdfv_demo/schemas/product#>
prefix prdc: <http://localhost:8890/rdfv_demo/schemas/product_category#>
prefix prdf: <http://localhost:8890/rdfv_demo/schemas/product_format#>

graph <http://localhost:8890/rdfv_demo/testdata/products#>
{
  prd:product_iri(OPLWEB.DBA.PRODUCT.PRODUCT_ID)
    a prd:Product ;
    prd:product_description OPLWEB.DBA.PRODUCT.PRODUCT_DESCRIPTION ;
    prd:product_category prdc:product_category_iri(OPLWEB.DBA.PRODUCT.PRO
    prd:product_format prdf:product_format_iri(OPLWEB.DBA.PRODUCT.PRODUCT
}
```

Two new IRI classes have also been introduced, prdc:product_category_iri and prdf:product_format_iri for converting the PRODUCT_CAT_ID and PRODUCT_FORMAT_ID foreign keys into IRIs. For brevity, these IRI class definitions are not shown. They take a similar form to that for the prd:product_iri IRI class whilst using the '%d' format specifier rather than '%s'.

Named Quad Pattern Sets - Quad Storage

When dealing with large numbers of quad patterns, it is often useful to be able to manipulate them as a set. Virtuoso provides this facility through *quad storage*. Quad storage is a named set of quad patterns. Three statements are provided for creating, altering and deleting quad pattern sets:

- create quad storage *storage–name* { *quad–map–declarations* } .
- alter quad storage *storage–name* { *quad–map–declarations–or–drops* } .
- drop quad storage *storage–name* .

Listing 7 extends the example in Listing 6 by defining the quad map pattern for the graph <http://localhost:8890/rdfv_demo/testdata/products#> as a member of the quad storage <http://localhost:8890/rdfv_demo/quad_storage/default>. Each quad map pattern for the products graph is now named. For example the second quad map pattern is named qs:product_productDesc. Using these map pattern names, individual map patterns can be dropped from the containing quad storage using the "`drop` *map-name*" directive inside an "`alter quad storage`" statement. Additional quad map patterns have also been added to synthesize the product_formats and product_categories graphs from the PRODUCT_FORMAT and PRODUCT_CATEGORY tables.

Describing Source Relational Tables

Listing 7 also illustrates, as an aid to conciseness, the use of table aliases in place of table names in quad map values. The table aliases are introduced using the "`from` *table-name* `as` *table-alias*" declarations. Join and filtering conditions can also be specified by including a WHERE clause. Every condition so specified is a SQL expression with placeholders where a reference to a table should be printed. In this example "`where (^{product_tbl.}^.PRODUCT_CAT_ID = 1)`" restricts the products contained in the products graph to those with a PRODUCT_CAT_ID = 1, with the result that only ODBC drivers are included in the RDF dataset. This filter is included for illustration only. Obviously it would not be present in a production system.

Listing 7:

```
sparql

prefix qs: <http://localhost:8890/rdfv_demo/quad_storage/>
prefix prd: <http://localhost:8890/rdfv_demo/schemas/product#>
prefix prdf: <http://localhost:8890/rdfv_demo/schemas/product_format#>
prefix prdc: <http://localhost:8890/rdfv_demo/schemas/product_category#>

create quad storage qs:default
  from OPLWEB.DBA.PRODUCT as product_tbl
      where (^{product_tbl.}^.PRODUCT_CAT_ID = 1)
  from OPLWEB.DBA.PRODUCT_FORMAT as product_format_tbl
  from OPLWEB.DBA.PRODUCT_CATEGORY as product_category_tbl
{
  create qs:products as
      graph <http://localhost:8890/rdfv_demo/testdata/products#>
  {
    prd:product_iri(product_tbl.PRODUCT_ID) a prd:Product
        as qs:product_Product ;
    prd:product_description product_tbl.PRODUCT_DESCRIPTION
        as qs:product_productDesc ;
    prd:product_category prdc:product_category_iri(product_tbl.PRODUCT_CA
        as qs:product_productCatId ;
    prd:product_format prdf:product_format_iri(product_tbl.PRODUCT_FORMAT
        as qs:product_productFormat .
  } .

  create qs:product_formats as
      graph <http://localhost:8890/rdfv_demo/testdata/product_formats#>
  {
    prdf:product_format_iri(product_format_tbl.PRODUCT_FORMAT_ID) a prdf:
        as qs:prdFrmt_ProductFormat ;
    prdf:product_format_description product_format_tbl.PRODUCT_FORMAT_DES
        as qs:prdFrmt_productFormatDescription .
  } .

  create qs:product_categories as
      graph <http://localhost:8890/rdfv_demo/testdata/product_categories#
  {
    prdc:product_category_iri(product_category_tbl.PRODUCT_CAT_ID) a prdc
        as qs:prdCat_ProductCategory ;
    prdc:product_category_description product_category_tbl.PRODUCT_CATEGO
        as qs:prdCat_productCategoryDescription .
  } .
} .
```

Listing 8:

```
define input:storage
<http://localhost:8890/rdfv_demo/quad_storage/default>

prefix prd: <http://localhost:8890/rdfv_demo/testdata/products#>
prefix prds: <http://localhost:8890/rdfv_demo/schemas/product#>
prefix prdfs: <http://localhost:8890/rdfv_demo/schemas/product_format#>
prefix prdcs: <http://localhost:8890/rdfv_demo/schemas/product_category#>

SELECT * WHERE {
  ?s prds:product_format _:pf .
  ?s prds:product_description ?pd .
  _:pf prdfs:product_format_description ?pfd .
  ?s prds:product_category _:pc .
  _:pc prdcs:product_category_description ?pcd .
}
```

Listing 9:

s	pd	pfd	pcd
http://localhost:8890/rdfv_demo /testdata/products#dotnet-postgres-mt	.NET Data Provider for PostgreSQL	Multi-Tier (Enterprise Edition)	ADO.NET Data Providers
http://localhost:8890/rdfv_demo /testdata/products#jdbc-ingres-mt	JDBC Driver for Ingres	Multi-Tier (Enterprise Edition)	JDBC Drivers
http://localhost:8890/rdfv_demo /testdata/products#odbc-informix-ee	ODBC Drivers for Informix	Single-Tier (Express Edition)	ODBC Drivers
http://localhost:8890/rdfv_demo /testdata/products#odbc-informix-mt	ODBC Driver for Informix	Multi-Tier (Enterprise Edition)	ODBC Drivers
http://localhost:8890/rdfv_demo /testdata/products#odbc-informix-st	ODBC Driver for Informix	Single-Tier (Lite Edition)	ODBC Drivers
http://localhost:8890/rdfv_demo /testdata/products#oledb-odbc-st	OLE DB Provider for ODBC	Single-Tier (Lite Edition)	OLEDB Data Providers

Querying Quad Storage

Having defined our quad mapping patterns, how do we invoke them? In order to use a particular set of mappings, Virtuoso must be instructed to use the necessary quad storage. Virtuoso extends SPARQL with a "`define input:storage` *storage–name*" directive which specifies that a SPARQL query be executed using quad patterns defined by the given quad storage. If the directive is omitted, the graphs defined by the quad storage are not visible.

Assuming our Virtuoso instance's HTTP listener is running on http://localhost:8890, we can issue the query in Listing 8 using the simple SPARQL query interface available at http://localhost:8890/sparql. (There is also a rich Web-based SPARQL user interface available at /isparql.) The query is intended to force joins across all three of the relational tables referenced by the RDF view. Listing 9 shows the query results.

Trying the Examples

Should you want to try setting up your own RDF views, all the sample code presented in the listings is available for download. Evaluation copies of the commercial Virtuoso product offerings can be downloaded from OpenLink's site. Alternatively, Virtuoso is available in open source form (VOS).

About the Example Code

The example code contains files with an assortment of extensions. Those with a '.sql' extension define the sample relational tables or hold sample data for these tables. The contained SQL can be executed directly by Virtuoso's isql command line interface with a command similar to:

```
isql localhost:1111 username password filename.sql
```

or through the isql component of the Virtuoso Conductor browser interface. Files with a '.spql' extension contain SPARQL which must be executed through the isql browser interface, as they are not recognized by the command line isql interface. The IRI classes must be created before the quad storage, so run the file create_iri_classes.spql before file create_quad_storage.spql.

Loading RDFS Class Definitions into Virtuoso

The example code includes three files, product.n3, product_category.n3 and product_format.n3, which define the RDFS classes Product, ProductCategory and ProductFormat. These must be loaded into the RDF triple store. One method is to use the API function DB.DBA.TTLP which parses and loads Turtle or N3. e.g.

```
SQL> ttlp (file_to_string_output ('product.n3'), '',
           'http://localhost:8890/rdfv_demo/schemas/product#', 0);
```

When loading the ProductCategory and ProductFormat classes, change the target graph name to end in 'product_category#' or 'product_format#' as appropriate. Also, ensure that the directory containing the file to be parsed is accessible through Virtuoso. To be so, it must be included in the '*DirsAllowed*' entry in your virtuoso.ini configuration file.

An alternative to TTLP is to use the OpenLink Data Spaces (ODS) Briefcase application. Using the 'Upload' feature, you can upload RDF/N3/Turtle data directly into the RDF store from a file. Give the destination as 'RDF Store' and the target RDF graph name as 'http://localhost:8890/rdfv_demo/schemas/product#', 'http://localhost:8890/rdfv_demo/schemas/product_format#' or 'http://localhost:8890 /rdfv_demo/schemas/product_category#' depending on which .n3 file you are uploading.

SPARQL Access to Relational Tables

Before you can query a relational table via an RDF View, you must grant the SELECT privilege on the table to user SPARQL or role SPARQL_SELECT. If you fail to do this, you will receive the error `'42000 Error SQ033: Access denied for column ...'`. Use the `Database` > `Schema Objects` menu commands in Conductor to navigate to the table then select the `Privileges` action.

Deleting RDF Graphs

If at any time you want to delete a graph from the RDF store, use the `drop graph` command:

```
sparql drop graph <http://localhost:8890/rdfv_demo/schemas/product#>;
```

Prior to Virtuoso 5.0, `drop graph` is not supported, in which case the following SQL might prove useful. The command deletes the given graph directly from the tables which underpin the RDF storage system. Change the graph name passed to the RDF_MAKE_IID_OF_QNAME function as appropriate.

```
delete from DB.DBA.RDF_QUAD where G =
    DB.DBA.RDF_MAKE_IID_OF_QNAME ('http://localhost:8890/rdfv_demo/schema
```

Finally, graphs defined in quad storage can be deleted by the command:

```
sparql
prefix qs: <http://localhost:8890/rdfv_demo/quad_storage/>
drop quad storage qs:default
```

Conclusion

Starting with version 4.5, Virtuoso provides built-in support for SPARQL and RDF. Adoption of SPARQL with Virtuoso is effortless, as any existing SQL client applications and stored procedures can take advantage of SPARQL simply by using it in the place of or inside SQL queries. Additionally, Virtuoso offers the standard SPARQL protocol to HTTP clients.

Since most of the data that is of likely use for the emerging semantic web is stored in relational databases, there is a clear need to expose this data in RDF form for access through SPARQL. Virtuoso provides this mechanism through RDF Views.

The example presented in this article has hopefully given you an overview of how to create RDF Views to expose relational data as RDF. Out of necessity, the descriptions of many of Virtuoso's features in this area have been simplified. For full details, please refer to the Virtuoso documentation, particularly as Virtuoso's RDF support is at an early implementation stage and evolving rapidly.