

## Virtuoso RDF Views – Getting Started Guide



v1.1 (Virtuoso 5.0)  
June 2007

OpenLink Virtuoso includes [SPARQL](#) support and an [RDF](#) data store tightly integrated with its relational storage engine. A key feature of this RDF offering is Virtuoso's RDF Views, a facility for exposing existing SQL data as RDF.

This guide is intended to supplement the “[RDF Database and SPARQL](#)” section of the [OpenLink Virtuoso Reference Manual](#), focussing specifically on RDF Views and how to create them, and to provide additional background information for readers unfamiliar with RDF. In addition, it briefly outlines the key technologies behind the burgeoning Semantic Web and sets Virtuoso's RDF facilities in context, explaining their relevance as an enabling technology for realising the Semantic Web vision.

# Table of Contents

Introduction.....	5
The Semantic Web.....	5
The Vision.....	5
Current Architecture & W3C Standards.....	5
Resource Description Framework (RDF).....	6
The RDF Data Model.....	6
RDF Notations and Serialization Formats.....	7
Why Is RDF Important?.....	7
The Need for a SQL to RDF Mapping Layer – Virtuoso RDF Views.....	7
Virtuoso Meta-Schema Language.....	8
Ontologies.....	9
Ontology Concepts.....	9
RDF Representation of Ontologies.....	9
Ontology Definition Languages.....	10
RDF Schema.....	10
OWL (Web Ontology Language).....	10
A Conceptual View of SQL to RDF Entity Mapping.....	11
1:1 Mapping of a SQL Table/View to an RDF Entity.....	11
Basic Mapping Algorithm.....	11
Next Steps.....	11
Example Scenario – An On-line Product Catalog.....	12
Our Relational Model.....	12
Our Target RDF Model.....	13
Representing a SQL Table as an RDFS Class.....	13
Loading RDFS Class Definitions into Virtuoso.....	14
Representing Relationships in RDFS.....	14
1:1 and 1:Many Relationships.....	14
Lookup Tables.....	14
Many:Many Relationships.....	15
Virtuoso RDF Views.....	16
Virtuoso RDF Meta-Schema Language.....	16
Quad Map Patterns.....	16
SPARQL-Style Notation.....	16
Named Quad Map Patterns.....	17
Group Map Patterns.....	17
Named Group Map Patterns.....	17
Quad Storage.....	17
Using Non-Default Quad Storage.....	18
IRI Classes – Creating IRIs from Keys.....	18
Literal Classes – Creating RDF Literals from Non-Key Column Values.....	19
Quad Pattern Construction Rules.....	19
A Simple RDF View Definition.....	20
Additional Meta-Schema Language Features.....	20
Table Aliases.....	21
SQL Selection and Joins Through Filters.....	21
SQL Projection.....	21
SQL Selection.....	21
Associating Filters with Table Aliases.....	21

SQL Joins.....	22
Appendix: RDF View for Northwind Traders Sample Database.....	23

## Change History

1.0 Initial draft (C.Blakeley, 7 June 07)

1.1 Corrected typographical errors. Added missing diagrams (C.Blakeley, 25 June 07)

# Introduction

## *The Semantic Web*

### The Vision

The Semantic Web is an evolving extension of the World Wide Web, originally envisioned by W3C director Tim Berners-Lee around 1999, which aims to supplement existing human-readable web content with data that can be interpreted and processed by computers, allowing users or programs to find and share information more easily.

Although the existing Web of HTML documents is easy for humans to navigate using just a browser, HTML pages fail to separate presentation details from the information they contain, making it difficult for software agents to identify, extract and process. The essential aim of this next evolution of the Web is to provide easy access to the underlying data it encompasses by allowing:

- the separation of real data from markup, so that a program doesn't have to strip out formatting and pictures from a Web page.
- people to write machine-interpretable descriptions of the relationship between different sets of data describing the same concepts, e.g. one database with a "telephone" column and another with a "phone\_no" field. Software agents can then follow links, building "**semantic links**" between related information and hence automatically integrate data from different sources.

The Web as it currently exists contains a vast amount of data which remains hidden and unused. The Semantic Web promises to unlock this data, transforming it from raw data into information, making it visible and increasing its utility many-fold.

### Current Architecture & W3C Standards

At its core, the Semantic Web comprises:

- a **data model** (Resource Description Framework ([RDF](#)))
- a variety of **data interchange formats** (e.g. [RDF/XML](#), [N3](#), [Turtle](#), [N-Triples](#)) and
- **notations for formal description of ontologies** or vocabularies, that is the concepts, terms and relationships within a given domain (e.g. [RDF Schema](#) (RDFS) and the Web Ontology Language ([OWL](#)))

The W3C standards used to implement these building blocks are shown in parentheses above. These standards form a layered architecture where successive layers support increasing expressiveness in describing the semantics of data.

Starting from the lowest layer:

- XML - provides a surface syntax for structured data, but imposes no semantic constraints on the meaning of this data.
- XML Schema - a language for restricting the structure and content elements of XML data.
- RDF - a simple data model for referring to objects ("resources") and how they are related. An RDF-based model can be represented in XML syntax.
- RDF Schema - a vocabulary for describing properties and classes of RDF

resources.

- OWL adds more vocabulary for describing properties and classes, including relations between classes, cardinality, equality, richer typing of properties, characteristics of properties (e.g. symmetry) and enumerated classes.

In the context of the Semantic Web, the most important of these standards is RDF, as it is this which underpins the Semantic Web data model.

## ***Resource Description Framework (RDF)***

### **The RDF Data Model**

At its most basic, RDF describes things by making statements about an entity's properties. There are two kinds of entities in RDF: resources and literals.

A resource is an item of interest, which may be abstract or concrete. e.g. A person or web site. Resources are named by uniform resource identifiers (URIs)<sup>1</sup>. Most people are familiar with one large subclass of URIs, namely uniform resource locators (URLs). The other major class of URIs is called uniform resource names (URNs). By convention, RDF resources have their URIs enclosed in angle brackets.

A literal is a string or fragment of XML. Literals are used to express basic properties of resources, such as names, ages, or anything that requires a human-readable description.

Using resources and literals, we can express information in RDF by composing statements. An RDF statement (aka a triple) consists of three parts:

1. A subject: the resource being talked about.
2. A predicate: a resource that describes the relationship between the subject and the object.
3. An object: a resource or literal whose interpretation depends on the predicate.

Some examples of statements are given below:

```
<urn:examples:albert> <urn:examples:job> <urn:examples:softwareEngineer>
```

```
<urn:examples:albert> <urn:examples:surname> "Smith"
```

A thorough description of a resource 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.

Although resources tend to be represented by URIs that intentionally denote actual, accessible data on the Web, RDF is not limited to the description of Internet-based resources. In fact, the URI that names a resource does not have to be dereferenceable at all. For example, a URI that begins with "http:" and is used as the subject of an RDF statement does not necessarily have to represent a resource that is accessible via HTTP, nor does it need to represent a tangible, network-accessible resource.

Rather than viewing RDF as a 'soup' of statements, RDF can alternatively be viewed as a

---

<sup>1</sup> The Internationalized Resource Identifier (IRI) is a generalization of the Uniform Resource Identifier (URI), which is in turn a generalization of the Uniform Resource Locator (URL). While URIs are limited to a subset of the ASCII character set, IRIs may contain Unicode characters. Basically, an IRI is the internationalized version of a URI. A URL is a URI that in addition to identifying a resource, provides a means of locating the resource by describing its network location. A URN is like a person's name, while a URL is like their street address. The URN defines something's identity, while the URL provides a method for finding something. Essentially, "what" vs. "where". A URN does not imply availability of the identified resource.

**graph model** in which statements represent nodes and arcs in a graph. In this notation, a statement is represented by:

- a node for the subject
- a node for the object
- an arc for the predicate, directed from the subject node to the object node.

Groups of statements are represented by corresponding groups of nodes and arcs.

## RDF Notations and Serialization Formats

As indicated above, RDF is essentially a graph model. It is important to differentiate between the abstract graph model and the notations used to express it in textual form. The W3C specifications define an XML format to encode RDF, RDF/XML, but other formats are also commonly used for writing or serializing RDF; examples being N3 (“Notation 3”), Turtle (“Terse RDF Triple Language”) and [JSON](#) (JavaScript Object Notation) Because two of the main benefits of N3 are conciseness and readability, many of the examples in this guide use it.

## Why Is RDF Important?

A majority of the world's data resides in relational databases that are sourced from a variety of vendors and based on a myriad of disparate schemas. There is an ever increasing need at all levels within the enterprise and across the internet to integrate the this disparate data. The only way to achieve this is to mesh the disparate schemas into a conceptual whole that provides the foundation for future application development and data access.

RDF provides an **open and platform independent conceptual data model**<sup>2</sup> that facilitates the meshing of disparate databases and schemas without lock-in at any of the following levels:

- operating system
- database
- application
- deployment platform
- data access protocol
- data representation format

Coupled with the SPARQL query language, RDF provides a powerful mechanism for querying and inferring against heterogeneous data. It provides a critical contribution to the endless challenge of **heterogeneous data integration**.

## The Need for a SQL to RDF Mapping Layer – Virtuoso RDF Views

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 as RDF. Because of databases' tried and trusted strengths in terms of performance, security, maintainability

---

<sup>2</sup> A **conceptual data model** is a map of concepts and their relationships, the concepts in this case being, for example, things of significance in an organization. The conceptual data model, or conceptual schema, is distinct from the logical or physical data models. The **logical data model** expresses the problem domain's data model in terms of a particular data management technology, for instance relational tables and columns. The **physical data model** describes the particular physical mechanisms used to capture data in a storage medium

and so forth, the core data should remain in the database, rather than be duplicated in RDF form outside the DBMS. Thus a key infrastructural requirement is a technology that enables the dynamic generation of RDF views of relational data. Virtuoso provides such a capability through its **RDF Views** support.

### ***Virtuoso Meta-Schema Language***

“RDF Views” is actually a moniker for referring to the two key technologies at the heart of Virtuoso's RDF support – Virtuoso's **RDF Meta<sup>3</sup>-Schema** and its declarative **Meta-Schema Language** for mapping SQL data to RDF ontologies.

What is a meta-schema language? A general definition might be “a declarative language for expressing relationships in abstract data models”. Based on this definition, the Virtuoso Meta Schema Language is a domain-specific extension of this concept for mapping a logical data model expressed in SQL to a conceptual data model expressed in RDF.

Before examining Virtuoso's RDF Views and Meta-Schema Language in detail however, we need first to look at some fundamental mapping concepts; that is: how data can be modelled conceptually through ontologies, how ontologies can be represented in RDF, and how, in the broadest conceptual terms, SQL data can be mapped to RDF.

---

3 “Meta-” - a prefix indicating a concept which is an abstraction from another concept, used to complete or add to the latter.

# ***Ontologies***

## **Ontology Concepts**

An ontology is a data model that represents a set of concepts within a domain and the relationships between those concepts. Ontologies have become increasingly common with the rise of the Semantic Web, as companies embark on making publicly available data once visible only on corporate intranets. As the sharing of information extends beyond company boundaries, so the need grows for a common vocabulary and an explicit formal specification of the concepts and relationships in a particular domain. Ontologies address this need.

Numerous languages exist for describing ontologies, but the principal components of an ontology are the same, irrespective of the language used to express it. These components are:

- Entities
- Attributes
- Classes
- Relationships

An **entity** (aka object instance) represents a discrete object that is distinguishable from other objects. Entities can be thought of as nouns, e.g. a company, a financial report. Entities represent individual objects in the domain of interest. They can have **attributes** (aka properties, predicates or features) which describe some aspect of the entity by giving it a name and a type. e.g. a person entity might have a social security number attribute. The attribute may be a scalar or composite type. Each attribute has an associated **attribute value**. Individual entity instances are uniquely identified by an entity ID (aka object ID). When entity instances are represented in an RDF data model, the entity ID typically forms the *subject* of statements about that entity, with the proviso that the entity ID must take the form of a URI. (The entity ID may also be the *object* in statements expressing relationships between entities.)

A **class** (aka entity type) defines a category of objects and all the common properties of the different objects that belong to it. Individual entities belonging to that category are **instances** of that class. A set of entities of the same type (e.g. the collection of all songs in a database) is known as an **entity set**. Each entity in the set is distinguishable by its entity ID.

A **relationship** captures how two or more entities are related to one another. Relationships can be thought of as verbs. e.g. an *employs* relationship between a company and an employee, a *manages* relationship between an employee and a department. Like other characteristics of an entity, they can be expressed as properties. If a property refers to a scalar or composite type, it is a conventional attribute. If it refers to another entity, it defines a relationship.

## **RDF Representation of Ontologies**

A collection of RDF statements represents a labeled, directed graph. Ontologies can be thought of in similar terms with each entity or attribute constituting a node in the graph and relationships between entities, or between entities and their attributes (i.e. an implicit 'has a' relationship) being represented by directed arcs between the nodes. Consequently an RDF-based data model lends itself naturally to the representation of ontologies and

conceptual data models. Moreover, more expressive ontology languages, such as RDF Schema and OWL, can be built upon RDF.

### ***Ontology Definition Languages***

The current architecture for the Semantic Web is split into three ontological layers. From the lowest to the highest:

- RDF: lets you assert facts  
e.g. person X has the surname "Smith".
- RDF Schema: lets you describe vocabularies and use them to describe things  
e.g. entity X is an Automobile.
- Web Ontology Language (OWL): lets you describe relationships between vocabularies  
e.g. private-vehicles in schema X are the same thing as automobiles in schema Y.

### ***RDF Schema***

With RDF, we can refer to resources and describe their properties. But how do we know when a resource is, for instance, a CV? We need a facility to state "X is a resource of class CV."

The RDF Schema specification defines a standard vocabulary for talking about RDF properties and defining classes of RDF resources. RDF Schema does not provide actual application-specific classes and properties. Instead it provides the framework to describe application-specific classes and properties. The RDF Schema ontology is defined within the `<http://www.w3.org/2000/01/rdf-schema#>` namespace and is commonly referred to by the `rdfs:` prefix.

### ***OWL (Web Ontology Language)***

RDF Schema allows you to define classes and their properties. However, how do you know if:

- Two different classes are equivalent?
- A property can be multi-valued?

OWL adds the ability to indicate when two classes or properties are identical allowing bodies of data in different schemas be linked together. OWL actually comprises three increasingly expressive sublanguages (Lite, DL and Full) which support inference and provide vocabulary for describing: cardinality, equality, richer typing of properties, characteristics of properties (e.g. symmetry) and enumerated classes. It is, unsurprisingly, a complex language. Fortunately, for the purposes of creating Virtuoso RDF Views, knowledge of OWL is not necessary, and OWL is not discussed further in this document.

## A Conceptual View of SQL to RDF Entity Mapping

At the most basic level, Virtuoso's RDF Views transform the result set of a SQL SELECT statement into a set triples. Before describing how these transformations are defined using Virtuoso's Meta-Schema Language, it is worth considering how, in general terms, SQL data can be transformed to RDF.

### *1:1 Mapping of a SQL Table/View to an RDF Entity*

When building a SQL representation of a problem domain, the starting point is frequently an entity-relationship diagram (ERD). Typically, each entity is represented as a database 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. The table rows collectively describe an entity set.

In an equivalent RDF representation of the same entity set:

- Each column in the table is an attribute (i.e. predicate)
- Each column value is an attribute value (i.e. object)
- Each row key represents an entity ID (i.e. subject)
- Each row represents an entity instance
- Each row (entity instance) is represented in RDF by a collection of triples with a common subject (entity ID).

### Basic Mapping Algorithm

So, to render an equivalent view in RDF, in the simplest case, our basic algorithm could be:

1. create an RDFS class for each table
2. convert all primary keys and foreign keys into IRI's
3. assign a predicate IRI to each column
4. assign an `rdf:type` predicate for each row, linking it to an RDFS class IRI corresponding to the table
5. 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.

### Next Steps

Based on the above algorithm, some of the key requirements are:

- Definition of a RDFS class and IRI for each table
- Construction of a predicate IRI for each non-key column
- Construction of an IRI for each key value

We will concentrate on the first two of these requirements for now. The last requirement we will revisit later in the section "IRI Classes".

## Example Scenario – An On-line Product Catalog

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.

### 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. (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 oplweb in catalog oplweb2.

Table: product		
product_id	varchar(25)	primary key
product_description	varchar(125)	
long_description	long varchar	
product_cat_id	integer	(foreign key)
product_format_id	integer	(foreign key)

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)	

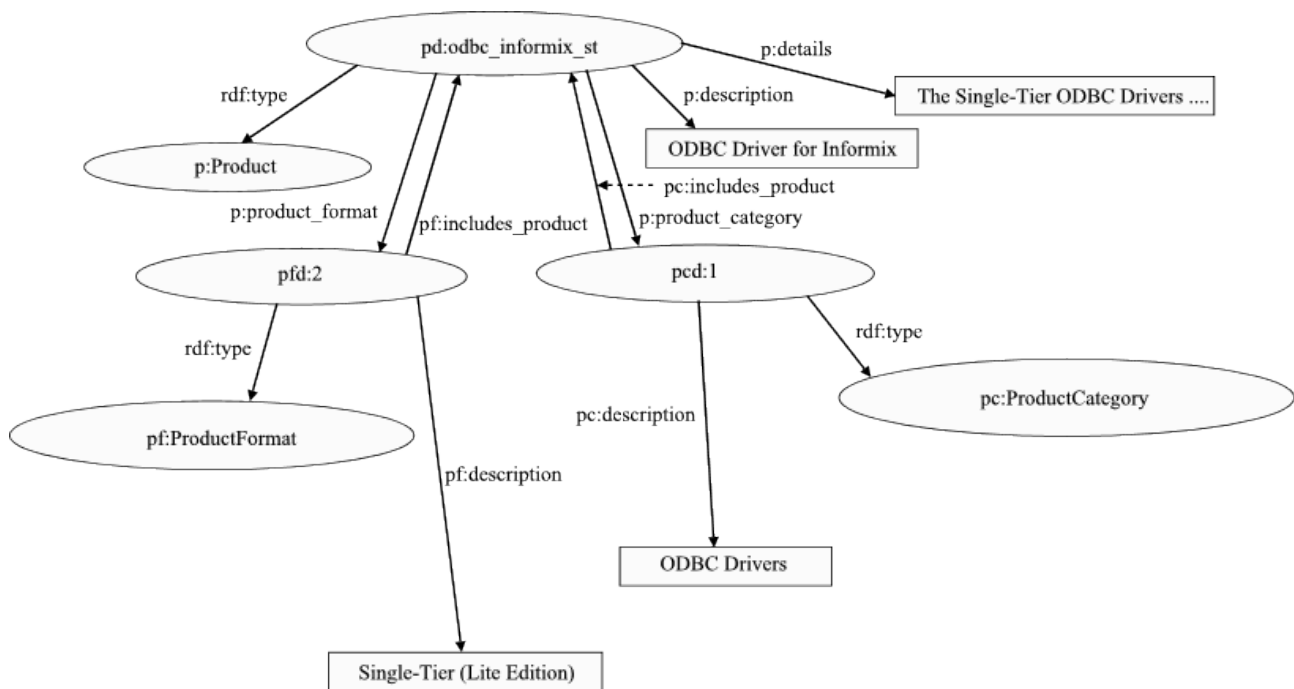
Sample data: product (column long_description not shown)			
product_id	product_description	product_cat_id	product_format_id
odbc-informix-ee	ODBC Drivers for Informix	1	4
odbc-informix-mt	ODBC Driver for Informix	1	3
odbc-informix-st	ODBC Driver for Informix	1	2
jdbc-ingres-mt	JDBC Driver for Ingres	2	3
oledb-odbc-st	OLE DB Provider for ODBC	3	2
dotnet-postgres-mt	.NET Data provider for PostgreSQL	4	3

Sample data: product_category	
product_cat_id	product_category_description
1	ODBC Drivers
2	JDBC Drivers
3	OLE DB Data Providers
4	ADO.NET Data Providers

Sample data: 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)

## 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.



```

prefix p: <http://www.openlinksw.com/schemas/oplweb/product#>
prefix pc: <http://www.openlinksw.com/schemas/oplweb/product_category#>
prefix pf: <http://www.openlinksw.com/schemas/oplweb/product_format#>
prefix pd: <http://www.openlinksw.com/oplweb/product/>
prefix pfid: <http://www.openlinksw.com/oplweb/product_format/>
prefix pcid: <http://www.openlinksw.com/oplweb/product_category/>

```

## Representing a SQL Table as an RDFS Class

How do we represent a relational table as an RDFS class? Using the product table as an example, and ignoring foreign keys for now, the entity class described by the table's DDL can be described as an equivalent RDF Schema class as shown below.

```

@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 p: <http://www.openlinksw.com/schemas/oplweb/product#> .

```

```

p:Product a rdfs:Class ;
    rdfs:label "Product" ;
    rdfs:comment "Product" .

p:details a rdf:Property ;
    rdfs:domain p:Product ;
    rdfs:range xsd:string ;
    rdfs:label "details" .

p:description a rdf:Property ;
    rdfs:domain p:Product ;
    rdfs:range xsd:string ;
    rdfs:label "description" .

```

It can be seen that an RDFS class Product, corresponding to the table, is defined by the 'a' (a shorthand for `rdf:type`) predicate. The class IRI is <http://www.openlinksw.com/schemas/oplweb/product#Product>.

Each column in the table becomes an entity attribute tied to the class through the `rdfs:domain` predicate. In each case, the SQL column type has been mapped to an equivalent XML Schema type through the `rdfs:range` predicate. The process of defining an RDF class for a table also fulfils a second requirement identified earlier: viz: construction of a predicate IRI for each non-key column.

Notice that the attribute names need not match the SQL column names. (As will be seen later, the link between the source column and target attribute is defined by the RDF view definition.) In this case, column `product_description` will be mapped to attribute 'description', and column `long_description` mapped to attribute 'details'.

## Loading RDFS Class Definitions into Virtuoso

Any RDFS classes you define for tables must be loaded into Virtuoso's RDF triple store. One method is to use the API function `DB.DBA.TTLP` which parses and loads Turtle or N3.

```

ttlp (file_to_string_output ('tmp/product.n3'), '',
'http://www.openlinksw.com/schemas/oplweb/product#', 0);

```

Here file `product.n3` holds the RDFS class definition for the product table. The directory containing the file to be parsed should be accessible through Virtuoso. To be so, it must be included in the 'DirsAllowed' entry in your `virtuoso.ini` configuration file.

The graph holding the RDFS class definition can be deleted using:

```

sparql drop graph <http://www.openlinksw.com/schemas/oplweb/product#>;

```

## Representing Relationships in RDFS

One area not addressed by our basic mapping algorithm is how to handle foreign keys and replicate table relationships in RDF.

### 1:1 and 1:Many Relationships

Relationships expressed through foreign keys in SQL are represented more naturally in RDF. In the SQL realm, a foreign key is simply an association mechanism. The foreign key column is not a true entity attribute in the same sense as other non-key columns in the table, nor is the nature of the relation ('employs', 'contains' or whatever) apparent.

Looking at the RDFS classes for our product portfolio, 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. Through these attributes, each `Product` entity instance points directly to its parent `ProductFormat` and `ProductCategory` instances.

The same mechanism can be used irrespective of whether a relationship is 1:1 or 1:many. For instance, `ProductCategory` has an attribute `'includes_product'` for collecting all the child `Products` belonging to that category. (The `'includes_product'` attribute expresses the inverse of the relationship expressed by the `'product_category'` attribute in the `Product` class.) Each child `Product` will generate a triple containing the parent `ProductCategory`'s IRI as the subject, the `'includes_product'` predicate and the `Product`'s IRI as the object.

Although RDF includes features for describing collections (indeed these features have been deprecated), these features are unnecessary for 1:many relationships. Note also that RDF and RDFS cannot express cardinality constraints – a more expressive language, such as OWL, is required to describe such meta-information.

### ***Lookup Tables***

Database designers frequently use foreign keys to implement lookup tables. Such tables simulate an enumerated type where the DBMS does not support this notion directly. In these cases it may be better to map the value referenced by the foreign key directly to a typed literal rather than to a separate entity, denoting the value of the enumerated type, which must be referenced via a URI.

### **Many:Many Relationships**

Where a many:many relationship exists between entities, a DBMS generally has to model this using a join table containing foreign keys from each of the tables representing the entities. Assuming that the join table contains no additional fields, i.e. the join table is simply an artifact of expressing the many:many relationship, the relationship may be expressed directly in RDF using property arcs. The many:many relationship can be viewed as two 1:many relationships facing in opposite directions.

## Virtuoso RDF Views

Having outlined the conceptual basis for mapping SQL data to RDF, we're now in a position to look at RDF Views. Virtuoso RDF Views expose pre-existing relational data as virtual RDF graphs available for querying directly through SPARQL or via Virtuoso's in-built support for SPARQL embedded within SQL (SPASQL). The virtual RDF graphs are created without physically regenerating the relational data as RDF data sets.

As indicated earlier, the key components of Virtuoso RDF Views are the Virtuoso RDF Meta-Schema and the RDF Meta-Schema Language.

### *Virtuoso RDF Meta-Schema Language*

The building blocks of the meta schema are **quad map patterns**, **IRI classes** and **literal classes**.

Other meta-schema features such as **group map patterns** and **quad storage** are essentially organizational enhancements aimed at making it easier to administer large sets of quad map patterns through the use of grouping and naming:

- **group map patterns**: group together map patterns which share a common graph
- **quad storage**: groups together group map patterns as a named set

**Naming** is used at three levels – quad map patterns, group map patterns and quad storage can all be named to facilitate altering or deleting map patterns individually, or as a group at the group map pattern or quad storage level. An additional benefit of naming is easier debugging and more readable debug output.

### Quad Map Patterns

The basic unit of the meta schema is a **quad map pattern**. A simple quad map pattern fully defines one particular transformation from one set of relational columns into triples that match one SPARQL graph pattern. At its heart, an RDF view definition is simply a collection of quad map patterns.

The main part of a quad map pattern is four declarations of **quad map values**, with each declaration specifying how to calculate the value of the corresponding triple field from the SQL data.

A simple example of a quad map pattern is given below. The `product_iri` function will be explained subsequently. The key point to note is that each of the four lines in this example constitutes a quad map value.

```
graph      <http://www.openlinksw.com/oplweb/>
subject    p:product_iri (oplweb2.oplweb.product.product_id)
predicate  p:description
object     oplweb2.oplweb.product.product_description
```

### *SPARQL-Style Notation*

Virtuoso's meta-schema description language also supports a SPARQL-style notation; so the above example can be written more concisely as:

```
graph <http://www.openlinksw.com/oplweb/>
{
    p:product_iri (oplweb2.oplweb.product.product_id) p:description
                oplweb2.oplweb.product.product_description .
}
```

## Named Quad Map Patterns

Quad map patterns (aka quad patterns) can be named. The assigned name then acts as a logical name which identifies the combination of a named graph and its associated triple pattern. For example, the next code extract names the previous quad pattern as `product_product_description`, defining this name in the namespace `virtrdf`.

```
graph <http://www.openlinksw.com/oplweb/>
{
    p:product_iri (oplweb2.oplweb.product.product_id) p:description
        oplweb2.oplweb.product.product_description
    as virtrdf:product_product_description .
}
```

Before dissecting quad map patterns further and their use of IRI classes and literal classes, we'll focus on how quad map patterns can be grouped using group map patterns and quad storages.

## Group Map Patterns

Quad map patterns for the same graph can be grouped together into a **group map pattern**<sup>4</sup>. Refining our previous examples further, the next example defines two unnamed quad map patterns, grouped under a common constant graph IRI `<http://www.openlinksw.com/oplweb/>` and sharing a common subject IRI (the latter synthesised from column `oplweb2.oplweb.product.product_id` by the function `p:product_iri`).

```
graph <http://www.openlinksw.com/oplweb/>
{
    p:product_iri (oplweb2.oplweb.product.product_id)
        a p:Product ;
    p:description oplweb2.oplweb.product.product_description .
}
```

## Named Group Map Patterns

Like quad patterns, group map patterns can also be named. We can, for instance, rewrite the above example to name both the quad map patterns and the group map pattern thus:

```
create virtrdf:product_portfolio as graph <http://www.openlinksw.com/oplweb/>
{
    p:product_iri (oplweb2.oplweb.product.product_id) a p:Product
        as virtrdf:product_product_id ;
    p:description oplweb2.oplweb.product.product_description
        as virtrdf:product_product_description .
}
```

where the group map pattern has been named `virtrdf:product_portfolio` and the quad patterns named `virtrdf:product_product_id` and `virtrdf:product_product_description` respectively.

## Quad Storage

**Quad storage** is a named set of quad patterns, used for compartmentalizing the RDF to SQL mapping. Quad patterns contained by a particular quad storage can then be manipulated en-bloc. The three statements for manipulating storages are :

- `create quad storage storage-name { quad-map declarations } .`
- `alter quad storage storage-name { quad-map declarations or drop commands } .`

---

<sup>4</sup> A better name would be 'map pattern group'.

- drop quad storage *storage-name* .

A map pattern can only be created within a quad storage definition, as a part of **create quad storage** or **alter quad storage** statement. (Initially, the map pattern is used by only one storage but, once created, map patterns can be imported from one quad storage into another). The **drop quad storage** statement deletes the named quad storage and all contained quad patterns. Quad map patterns can be deleted individually using the **drop quad map** *map-name* directive. When used inside an **alter quad storage** statement it removes a map only from that quad storage, otherwise it removes the map from all storages.

### Using Non-Default Quad Storage

In order to use a quad storage other than the default **virtrdf:defaultQuadStorage**, a SPARQL query must include a **define input:storage** *storage-name* directive. This declaration forces the SPARQL query to be executed using quad patterns of the given storage.

Depending on your application, including the directive explicitly in each query may not be practicable. An alternative mechanism to force its implicit inclusion is to use the table DB.DBA.SYS\_SPARQL\_HOST.

```
DB.DBA.SYS_SPARQL_HOST (
SH_HOST          varchar not null primary key, -- host mask
SH_GRAPH_URI     varchar,                      -- default graph uri
SH_USER_URI      varchar,                      -- reserved
SH_DEFINES       long varchar                 -- additional defines
)
```

Each HTTP request received on the SPARQL service endpoint is evaluated against this table. A default graph URI or additional defines can be configured for specific hosts. To set the default quad storage for a host, include a **define input:storage** statement in the SH\_DEFINES column. e.g.

```
insert into DB.DBA.SYS_SPARQL_HOST values (
    'myhost.mynet.private:8891', NULL, NULL,
    'define input:storage virtrdf:SHOPFRONT')
```

### IRI Classes – Creating IRIs from Keys

Recall in the earlier section “1:1 Mapping of a SQL Table/View to an RDF Entity” , one of the key requirements identified in the mapping process was:

- Construction of a subject IRI for each primary key column value

An IRI class performs this 'construction'. It defines how key values (for an atomic or compound key) are combined into an IRI string and how an IRI string is decomposed back into the key value(s).

When declaring that a table's primary key is converted into a IRI according to one IRI class, one usually declares that all foreign keys referring to this class also get converted into an IRI using the same class.

Shown below is 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. )

```
sparql
prefix prd: <http://www.openlinksw.com/schemas/oplweb/product#>
```

```
create iri class prd:product_iri
    "http://www.openlinksw.com/oplweb/product#%s"
    (
        in product_id varchar not null
    ) .
```

The example illustrates the use of a printf–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:

```
create iri class prd:product_iri using
    function oplweb2.oplweb.product_uri (in id varchar)
        returns varchar,
    function oplweb2.oplweb.product_uri_inverse (in id_iri varchar)
        returns varchar .
```

Here the functions `product_uri` and `product_uri_inverse` would be defined elsewhere (not shown), for instance in Virtuoso/PL.

## Literal Classes – Creating RDF Literals from Non-Key Column Values

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 when a column name is used directly as the object in a quad map pattern.

## Quad Pattern Construction Rules

The basic elements of the Virtuoso RDF Meta Schema Language have been described individually. An RDF view is defined by combining these elements to declare a collection of quad patterns, where each quad pattern can consist of the following components:

- Subject - This can be an IRI class function call taking one or more table columns as arguments. Alternately, it can be a constant IRI.
- Predicate - This is most often a constant IRI but can also be an IRI class as with the subject.
- Object - This can be an IRI class applied to columns, a literal class applied to a column, or a literal IRI or scalar.
- Graph - This is most often a constant IRI but can also be an IRI class as with the subject. It is very common for many quad patterns of one SQL schema to share one constant graph, but one meta schema may consist of quad patterns with any number of different graph declarations.
- Condition - This is an optional SQL search condition. If the search condition is true when applied to a row of the table designated in the subject, predicate and object patterns, then the triple is considered to exist, otherwise not.

Notice that this definition of a quad pattern differs slightly from our earlier definition in

which we asserted that the pattern comprised four parts – subject, object, predicate and graph. Here, we see an extra optional search condition. This option will be described more fully in the section “Implicit Joins and Selection Through Filters”.

## ***A Simple RDF View Definition***

Putting everything together we can, finally, present a simple RDF view definition for the product catalog. (A more complex example, showing a full view definition for the Northwind Traders sample database, can be found in the Appendix.)

```
prefix p: <http://www.openlinksw.com/schemas/oplweb/product#>
prefix pc: <http://www.openlinksw.com/schemas/oplweb/product_category#>
prefix pfo: <http://www.openlinksw.com/schemas/oplweb/product_format#>

alter quad storage virtrdf:DefaultQuadStorage
from oplweb2.oplweb.product as product_tbl
from oplweb2.oplweb.product_category as product_category_tbl
from oplweb2.oplweb.product_format as product_format_tbl
{
    create virtrdf:product_portfolio as
        graph <http://www.openlinksw.com/oplweb/>
        {
            p:product_iri(product_tbl.product_id)
                a p:Product
                as virtrdf:product_product_id ;
            p:details product_tbl.long_description
                as virtrdf:product_long_description ;
            p:product_category pc:product_category_iri(product_tbl.product_cat_id)
                as virtrdf:product_product_cat_id ;
            p:description product_tbl.product_description
                as virtrdf:product_product_description ;
            p:product_format pfo:product_format_iri(product_tbl.product_format_id)
                as virtrdf:product_product_format_id .

            pc:product_category_iri(product_category_tbl.product_cat_id)
                a pc:ProductCategory
                as virtrdf:product_category_product_cat_id ;
            pc:description product_category_tbl.product_category_description
                as virtrdf:product_category_product_category_description ;
            pc:includes_product p:product_iri(product_tbl.product_id)
                where (^{product_tbl.}^.product_cat_id =
                    ^{product_category_tbl.}^.product_cat_id)
                as virtrdf:product_category_product_collection .

            pfo:product_format_iri(product_format_tbl.product_format_id)
                a pfo:ProductFormat
                as virtrdf:product_format_product_format_id ;
            pfo:description product_format_tbl.product_format_description
                as virtrdf:product_format_product_format_description ;
            pfo:includes_product p:product_iri(product_tbl.product_id)
                where (^{product_tbl.}^.product_format_id =
                    ^{product_format_tbl.}^.product_format_id)
                as virtrdf:product_format_product_collection .
        } .
    } .
```

## ***Additional Meta-Schema Language Features***

The above example introduces a number of additional Meta-Schema Language features not yet covered.

## Table Aliases

Although fully qualified column names can be used directly in patterns, table aliases provide a concise alternative. By defining an alias with a statement such as

```
from oplweb2.oplweb.product as product_tbl
```

a four-part column reference *oplweb2.oplweb.product.product\_id* can be replaced by *product\_tbl.product\_id* in quad map values. Because quad map patterns of an application usually share a common set of source tables, a table alias is declared once at the beginning of the storage declaration and shared between all the quad map declarations it contains.

## SQL Selection and Joins Through Filters

### SQL Projection

For security reasons etc, it may not always be desirable to expose all the columns in a table or view. Also, there may be no point in exposing a key column's value directly as the value may have no meaning beyond being a unique identifier which is subsequently transformed into an IRI. To achieve the required SQL projection, obviously all that is necessary is to omit defining quad map patterns for the relevant columns, and to omit property definitions for these columns in the RDFS class for the table.

### SQL Selection

A quad map pattern can include a *condition*, a boolean SQL expression to filter out unwanted rows of source data. Every condition is an SQL expression with placeholders where a reference to the table should be printed.

```
from oplweb2.oplweb.product as product_tbl
from oplweb2.oplweb.product_category as product_category_tbl
graph <http://www.openlinksw.com/oplweb/>
{
...
    pc:product_category_iri(product_category_tbl.product_cat_id)
    a pc:ProductCategory
    as virtrdf:product_category_product_cat_id ;
    pc:includes_product p:product_iri(product_tbl.product_id)
    where (^{product_tbl.}^.product_cat_id =
           ^{product_category_tbl.}^.product_cat_id)
    as virtrdf:product_category_product_collection .
...
}
```

In this example, the condition is expressed through a “where” clause, with placeholders “*^{product\_tbl.}^*” and “*^{product\_category\_tbl.}^*”.

### Associating Filters with Table Aliases

An alternative to specifying filter conditions directly in quad patterns is to specify them alongside a table alias.

*(Note: The remaining examples of schema language features do not relate directly to the product catalog RDF view, as it does not use the features being described.)*

e.g.

```
from DB.DBA.SYS_USERS as user where (^{user.}^.U_IS_ROLE = 0)
```

```
from DB.DBA.SYS_USERS as group where (^{group.}^.U_IS_ROLE = 1)
```

All quad map patterns using the alias are then subject to the same filter condition. When a quad map pattern declaration refers to several aliases with conditions, the WHERE clause of the generated SQL contains a combination of all relevant conditions.

### **SQL Joins**

As well as filtering, a quad map pattern condition can also specify join criteria, as illustrated in the definition of the 'grant' table alias below:

```
from DB.DBA.SYS_USERS as user where (^{user.}^.U_IS_ROLE = 0)
from DB.DBA.SYS_USERS as group where (^{group.}^.U_IS_ROLE = 1)
from DB.DBA.SYS_USERS as account from user as active_user
    where (^{active_user.}^.U_ACCOUNT_DISABLED = 0)
from DB.DBA.SYS_ROLE_GRANTS as grant
    where (^{grant.}^.GI_SUPER = ^{account.}^.U_ID)
    where (^{grant.}^.GI_SUB = ^{group.}^.U_ID)
    where (^{grant.}^.GI_SUPER = ^{user.}^.U_ID)
```

Notice also that multiple conditions have been associated with the alias through three 'where' clauses.

Obviously for more complex joins, a quad map pattern can reference a SQL view joining multiple tables, rather than a true table.

## Appendix: RDF View for Northwind Traders Sample Database

The RDF View example below is based on a sample database, Northwind Traders, provided with Virtuoso. The database tables can be found in the Demo catalog.

```
--
-- mkdemo.sql
--
-- $Id: sql_rdf.sql,v 1.2 2007/05/30 12:48:30 aziz Exp $
--
-- RDF Views for demo database
--
-- This file is part of the OpenLink Software Virtuoso Open-Source (VOS)
-- project.
--
-- Copyright (C) 1998-2006 OpenLink Software
--
-- This project is free software; you can redistribute it and/or modify it
-- under the terms of the GNU General Public License as published by the
-- Free Software Foundation; only version 2 of the License, dated June 1991.
--
-- This program is distributed in the hope that it will be useful, but
-- WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
-- General Public License for more details.
--
-- You should have received a copy of the GNU General Public License along
-- with this program; if not, write to the Free Software Foundation, Inc.,
-- 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
--
--
create procedure dump_large_text(inout _text varchar)
{
  declare _strings any;
  declare _slen, _sctr any;
  declare STRG varchar;
  result_names(STRG);
  _strings := split_and_decode (_text,0,'\0\0\n');
  _slen := length (_strings);
  _sctr := 0;
  while (_sctr < _slen)
  {
    result (aref (_strings, _sctr));
    _sctr := _sctr+1;
  }
}
;

create procedure DB.DBA.SPARQL_NW_RUN (in txt varchar)
{
  declare REPORT, stat, msg, sqltext varchar;
  declare metas, rowset any;
  result_names (REPORT);
  sqltext := string_output_string (sparql_to_sql_text (txt));
  dump_large_text (sqltext);
  stat := '00000';
  msg := '';
  rowset := null;
  exec (sqltext, stat, msg, vector (), 1000, metas, rowset);
  result ('STATE=' || stat || ': ' || msg);
  if (rowset is not null)
  {
```

```

        foreach (any r in rowset) do
            result (r[0] || ': ' || r[1]);
    }
}
;

DB.DBA.SPARQL_NW_RUN ('
drop quad map graph iri("http://^{URIQADefaultHost}^/demo") .
')
;

DB.DBA.SPARQL_NW_RUN ('
prefix demo: <http://www.openlinksw.com/schemas/demo#>
prefix oplsioc: <http://www.openlinksw.com/schemas/oplsioc#>
prefix sioc: <http://rdfs.org/sioc/ns#>
create iri class demo:Category "http://^{URIQADefaultHost}^/demo/Category?id=%d"
(in category_id integer not null) .
create iri class demo:Shipper "http://^{URIQADefaultHost}^/demo/Shipper?id=%d"
(in shipper_id integer not null) .
create iri class demo:Supplier "http://^{URIQADefaultHost}^/demo/Supplier?id=%d"
(in supplier_id integer not null) .
create iri class demo:Product "http://^{URIQADefaultHost}^/demo/Product?id=%d"
(in product_id integer not null) .
create iri class demo:Customer "http://^{URIQADefaultHost}^/demo/Customer?id=%d"
(in customer_id integer not null) .
create iri class demo:Employee "http://^{URIQADefaultHost}^/demo/Employee?id=%d"
(in employee_id integer not null) .
create iri class demo:Order "http://^{URIQADefaultHost}^/demo/Order?id=%d" (in
order_id integer not null) .
create iri class demo:OrderLine
"http://^{URIQADefaultHost}^/demo/OrderLine?id=%d&prod_id=%d" (in order_id
integer not null, in product_id integer not null) .
create iri class demo:Province
"http://^{URIQADefaultHost}^/demo/Province?country=%s&province=%s" (in
country_name varchar not null, in province_name varchar not null) .
create iri class demo:Country
"http://^{URIQADefaultHost}^/demo/Country?country=%s" (in country_name varchar
not null) .
')
;

DB.DBA.SPARQL_NW_RUN ('
prefix demo: <http://www.openlinksw.com/schemas/demo#>
prefix oplsioc: <http://www.openlinksw.com/schemas/oplsioc#>
prefix sioc: <http://rdfs.org/sioc/ns#>
create quad storage virtrdf:Northwind
from Demo.demo.Products as products
from Demo.demo.Suppliers as suppliers
from Demo.demo.Shippers as shippers
from Demo.demo.Categories as categories
from Demo.demo.Customers as customers
from Demo.demo.Employees as employees
from Demo.demo.Orders as orders
from Demo.demo.Order_Details as order_lines
from Demo.demo.Countries as countries
from Demo.demo.Provinces as provinces
{
    create virtrdf:Demo as graph iri ("http://^{URIQADefaultHost}^/demo")
option (exclusive)
{
    demo:Product (products.ProductID)
        a demo:Product
            as virtrdf:Product-ProductID ;
    demo:has_category demo:Category (products.CategoryID)

```

```

        as virtrdf:Product-product_has_category ;
demo:has_supplier demo:Supplier (products.SupplierID)
        as virtrdf:Product-product_has_supplier ;
demo:ProductName products.ProductName
        as virtrdf:Product-name_of_product ;
demo:QuantityPerUnit products.QuantityPerUnit
        as virtrdf:Product-quantity_per_unit ;
demo:UnitPrice products.UnitPrice
        as virtrdf:Product-unit_price ;
demo:UnitsInStock products.UnitsInStock
        as virtrdf:Product-units_in_stock ;
demo:UnitsOnOrder products.UnitsOnOrder
        as virtrdf:Product-units_on_order ;
demo:ReorderLevel products.ReorderLevel
        as virtrdf:Product-reorder_level ;
demo:Discontinued products.Discontinued
        as virtrdf:Product-discontinued .
demo:Supplier (suppliers.SupplierID)
  a demo:Supplier
        as virtrdf:Supplier-SupplierID ;
demo:CompanyName suppliers.CompanyName
        as virtrdf:Supplier-company_name ;
demo:ContactName suppliers.ContactName
        as virtrdf:Supplier-contact_name ;
demo:ContactTitle suppliers.ContactTitle
        as virtrdf:Supplier-contact_title ;
demo:Address suppliers.Address
        as virtrdf:Supplier-address ;
demo:City suppliers.City
        as virtrdf:Supplier-city ;
demo:Region suppliers.Region
        as virtrdf:Supplier-region ;
demo:PostalCode suppliers.PostalCode
        as virtrdf:Supplier-postal_code ;
demo:Country suppliers.Country
        as virtrdf:Supplier-country ;
demo:Phone suppliers.Phone
        as virtrdf:Supplier-phone ;
demo:Fax suppliers.Fax
        as virtrdf:Supplier-fax ;
demo:HomePage suppliers.HomePage
        as virtrdf:Supplier-home_page .
demo:Category (categories.CategoryID)
  a demo:Category
        as virtrdf:Category-CategoryID ;
demo:CategoryName categories.CategoryName
        as virtrdf:Category-home_page ;
demo:Description categories.Description
        as virtrdf:Category-description ;
demo:Picture categories.Picture
        as virtrdf:Category-picture .
demo:Shipper (shippers.ShipperID)
  a demo:Shipper
        as virtrdf:Shipper-ShipperID ;
demo:CompanyName shippers.CompanyName
        as virtrdf:Shipper-company_name ;
demo:Phone shippers.Phone
        as virtrdf:Shipper-phone .
demo:Customer (customers.CustomerID)
  a demo:Customer
        as virtrdf:Customer-CustomerID ;
demo:CompanyName customers.CompanyName
        as virtrdf:Customer-company_name ;
demo:ContactName customers.ContactName

```

```

        as virtrdf:Customer-contact_name ;
demo:ContactTitle customers.ContactTitle
        as virtrdf:Customer-contact_title ;
demo:Address customers.Address
        as virtrdf:Customer-address ;
demo:City customers.City
        as virtrdf:Customer-city ;
demo:Region customers.Region
        as virtrdf:Customer-region ;
demo:PostalCode customers.PostalCode
        as virtrdf:Customer-postal_code ;
demo:Country customers.Country
        as virtrdf:Customer-country ;
demo:Phone customers.Phone
        as virtrdf:Customer-phone ;
demo:Fax customers.Fax
        as virtrdf:Customer-fax .
demo:Employee (employees.EmployeeID)
  a demo:Employee
    as virtrdf:Employee-EmployeeID ;
demo:LastName employees.LastName
        as virtrdf:Employee-last_name ;
demo:FirstName employees.FirstName
        as virtrdf:Employee-first_name ;
demo:Title employees.Title
        as virtrdf:title ;
demo:TitleOfCourtesy employees.TitleOfCourtesy
        as virtrdf:Employee-title_of_courtesy ;
demo:BirthDate employees.BirthDate
        as virtrdf:Employee-birth_date ;
demo:HireDate employees.HireDate
        as virtrdf:Employee-hire_date ;
demo:Address employees.Address
        as virtrdf:Employee-address ;
demo:City employees.City
        as virtrdf:Employee-city ;
demo:Region employees.Region
        as virtrdf:Employee-region ;
demo:PostalCode employees.PostalCode
        as virtrdf:Employee-postal_code ;
demo:Country employees.Country
        as virtrdf:Employee-country ;
demo:HomePhone employees.HomePhone
        as virtrdf:Employee-home_phone ;
demo:Extension employees.Extension
        as virtrdf:Employee-extension ;
demo:Photo employees.Photo
        as virtrdf:Employee-photo ;
demo:Notes employees.Notes
        as virtrdf:Employee-notes ;
demo:ReportsTo employees.ReportsTo
        as virtrdf:Employee-reports_to .
demo:Order (orders.OrderID)
  a demo:Order
    as virtrdf:Order-Order ;
demo:has_customer demo:Customer (orders.CustomerID)
        as virtrdf:Order-order_has_customer ;
demo:has_employee demo:Employee (orders.EmployeeID)
        as virtrdf:Order-order_has_employee ;
demo:OrderDate orders.OrderDate
        as virtrdf:Order-order_date ;
demo:RequiredDate orders.RequiredDate
        as virtrdf:Order-required_date ;
demo:ShippedDate orders.ShippedDate

```

```

        as virtrdf:Order-shipped_date ;
demo:order_ship_via demo:Shipper (orders.ShipVia)
    as virtrdf:Order-order_ship_via ;
demo:Freight orders.Freight
    as virtrdf:Order-freight ;
demo:ShipName orders.ShipName
    as virtrdf:Order-ship_name ;
demo:ShipAddress orders.ShipAddress
    as virtrdf:Order-ship_address ;
demo:ShipCity orders.ShipCity
    as virtrdf:Order-ship_city ;
demo:ShipRegion orders.ShipRegion
    as virtrdf:Order-ship_region ;
demo:ShipPostal_code orders.ShipPostalCode
    as virtrdf:Order-ship_postal_code ;
demo:ShipCountry orders.ShipCountry
    as virtrdf:ship_country .
demo:OrderLine (order_lines.OrderID, order_lines.ProductID)
    a demo:OrderLine
        as virtrdf:OrderLine-OrderLines ;
demo:has_order_id demo:Order (order_lines.OrderID)
    as virtrdf:order_lines_has_order_id ;
demo:has_product_id demo:Product (order_lines.ProductID)
    as virtrdf:order_lines_has_product_id ;
demo:UnitPrice order_lines.UnitPrice
    as virtrdf:OrderLine-unit_price ;
demo:Quantity order_lines.Quantity
    as virtrdf:OrderLine-quantity ;
demo:Discount order_lines.Discount
    as virtrdf:OrderLine-discount .
demo:Country (countries.Name)
    a demo:Country
        as virtrdf:Country-Name ;
demo:Code countries.Code
    as virtrdf:Country-Code ;
demo:SmallFlagDAVResourceName
countries.SmallFlagDAVResourceName
    as virtrdf:Country-SmallFlagDAVResourceName ;
demo:LargeFlagDAVResourceName
countries.LargeFlagDAVResourceName
    as virtrdf:Country-LargeFlagDAVResourceName ;
demo:SmallFlagDAVResourceURI
countries.SmallFlagDAVResourceURI
    as virtrdf:Country-SmallFlagDAVResourceURI ;
demo:LargeFlagDAVResourceURI
countries.LargeFlagDAVResourceURI
    as virtrdf:Country-LargeFlagDAVResourceURI ;
demo:Lat countries.Lat
    as virtrdf:Country-Lat ;
demo:Lng countries.Lng
    as virtrdf:Country-Lng .
demo:Province (provinces.CountryCode, provinces.Province)
    a demo:Province
        as virtrdf:Province-Provinces ;
demo:has_country_code demo:Country
(provinces.CountryCode)
    as virtrdf:has_country_code ;
demo:Province provinces.Province
    as virtrdf:Province-Province .
}
')
;
```