ACTA UNIVERSITATIS UPSALIENSIS
Uppsala Dissertations from the Faculty of Science and Technology
75

Johan Petrini

Querying RDF Schema Views of Relational Databases

UPPSALA UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in 2446, Polacksbacken, hus 2, Lägerhyddsvägen 2, Uppsala, Monday, May 26, 2008 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract
Petrini, J. 2008. Querying RDF Schema Views of Relational Databases. Acta Universitatis Upsaliensis. *Uppsala Dissertations from the Faculty of Science and Technology* 75. 117 pp. Uppsala. ISBN 978-91-554-7202-3.

The amount of data found on the web today and its lack of semantics makes it increasingly harder to retrieve a particular piece of information. With the Resource Description Framework (RDF) every piece of information can be annotated with properties describing its semantics. The higher level language RDF Schema (RDFS) is defined in terms of RDF and provides means to describe classes of RDF resources and properties defined over these classes. Queries over RDFS data can be specified using the standard query language SPARQL. Since the majority of information in the world still resides in relational databases it should be investigated how to view and query their contents as views defined in terms of RDFS meta-data descriptions. However, processing of queries to general RDFS views over relational databases is challenging since the queries and view definitions are complex and the amount of data often is huge. A system, Semantic Web Abridged Relational Databases (SWARD), is developed to enable efficient processing of SPARQL queries to RDFS views of data in existing relational databases. The RDFS views, called universal property views (UPVs), are automatically generated provided a minimum of user input. A UPV is a general RDFS view of a relational database representing both its schema and data. Special attention is devoted to making the UPV represent as much of the relational database semantics as possible, including foreign and composite keys. A general query reduction algorithm, called PARtial evaluation of Queries (PARQ), for queries over complex views, such as UPVs, has been developed. The reduction algorithm is based on the program transformation technique partial evaluation. For UPVs, the PARQ algorithm is shown to elegantly reduce queries dramatically before regular cost-based optimization by a back-end relational DBMS. The results are verified by performance measurements of SPARQL queries to a commercial relational database.

Keywords: query processing, partial evaluation, rdf, rdf schema, view, relational databases, semantic web

Johan Petrini, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden

ISSN 1104-2516
ISBN 978-91-554-7202-3
urn:nbn:se:uu:diva-8740 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-8740)

Printed in Sweden by Universitetstryckeriet, Uppsala 2008
Distributor: Uppsala University Library, Box 510, SE-751 20 Uppsala
www.uu.se, acta@ub.uu.se

Contents

Abbreviations

DBMS	Database Management System
JDBC	Java Database Connectivity
RDF	Resource Description Framework
RDFS	RDF Schema
SQL	Structured Query Language
SWARD	Semantic Web Abridged Relational Databases
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

1 Introduction

The amount of data found on the web today and its lack of semantics makes it increasingly harder to retrieve a particular piece of information. Free-text search engines often return too many and too incorrect results. Another problem arises when trying to combine the collected pieces of information in a meaningful way. For example, applications accessing information from several databases with different structure and content has to decide if a column ‘A’ from one database has the same meaning as some column ‘A’ from another database. This is very challenging due to the lack of semantics in the database schema.

There is clearly a need for a uniform way to provide descriptions of information that could help facilitate for both searching and combining data. With RDF [20][32] every piece of information can be annotated with *properties* describing its semantics. Meta-data descriptions such as Dublin Core [22], Open Directory [41], RSS 1.0 [49], Uniprot catalog of protein sequence and annotation [66], NASA [61], WordNet [69] and GovML [63] use RDF.

RDF is the basis for most semantic web representations and several higher level languages are defined in terms of RDF, e.g. RDF Schema (RDFS) [11], OWL [42], and OWL Lite [42]. RDFS provides means to describe classes of RDF resources and properties defined over these classes. The standard query language SPARQL [58] is used for querying RDF data.

RDF repository systems [12][15][67] offer storage of RDF data and the ability to search RDF data using a query language.

Since the majority of information in the world still resides in relational databases it should be investigated how to expose this information as RDF queryable with SPARQL. RDFS could provide support for representation of both content and schema in relational databases. This would allow for flexible queries combining content and schema information in the relational database as opposed to in SQL.

One way to expose data in relational databases as RDFS representations is by downloading them to RDF repositories. However, this can be very costly when the relational database is large. The fact that all data in the relational database is duplicated as RDF introduces a lot of data redundancy. Also, when the rate of change in the relational database is high, a lot of time is spent on propagating the changes to the RDF repository.

A better solution, proposed in this Thesis, is to define RDF Schema views over existing relational databases and allow database queries in, e.g., SPARQL over these RDFS views. This way redundancy and overhead for updates is eliminated.

The RDFS views over relational databases should be defined in a structured and general way. That is, it should be possible to define an RDFS view over any arbitrary relational database. RDFS view definitions should not be hard-coded for a specific relational database instance. The definition of general and well structured RDF Schema views would increase understandability and minimize the introduction of errors compared to in ad-hoc solutions. Therefore, RDFS views should be generated in a way that demands a minimum of user input.

RDFS views over relational databases should enable:

- Access to all content in the relational database in terms of RDF resources.
- Access to schema information about tables and columns in the relational database in terms of RDFS classes and properties. These classes and properties are instantiated by the RDF resources representing the database content.

Another issue is to retain as much as possible of the semantics of the relational database in the RDFS view. Relational databases are usually designed using the graphical high level conceptual *entity relationship* (ER) model. The designed ER diagrams are then translated to the relational model and implemented in a relational DBMS. ER and RDFS both work on the conceptual level whereas the relational model is more implementation specific. As a reason thereof, ER entities and relationships are implicitly represented in the relational model.

To retain semantics, RDFS views over relational databases should also:

- Make explicit ER type memberships and relationships in terms of RDF Schema constructs.

RDFS views that fulfil all of this are referred to here as *complete RDFS views*.

The processing of queries to an RDFS view over a relational database becomes challenging for two reasons: First, the flexible representation of data with RDF produces queries that involve many self-joins to a large disjunctive view (one disjunct for every viewed column in the relational database) where each disjunct in turn is defined as a conjunctive expression. Traditional processing of such queries produces enormous expressions internally and as a consequence of that unrealistically slow query processing times. Second, the queries are posed to RDFS views over relational databases and require therefore careful optimization in order to scale over the huge amount of data that can be stored in a relational database.

The two main research questions identified in this Thesis are:

- How can general and complete RDF Schema views of relational databases be automatically generated?
- How can scalable processing of realistic size SPARQL queries to large RDF Schema views of relational databases be achieved?

To investigate these questions we developed a system, *Semantic Web Abridged Relational Databases* (*SWARD*) [62][44][45] to enable efficient processing of SPARQL queries to RDFS views of relational databases. Complete RDFS views are automatically generated specified with a minimum of user input.

In SWARD, relational database content and schema information is represented in RDF as a large disjunctive view. We call such a view a *universal property view* or *UPV*. A UPV is an RDFS mapping of a relational database defined as a union of a *schema view*, representing the database schema, and a *content view*, representing the database content. The content view, in turn, is defined as a union of *property views*, each representing one viewed column in the relational database. The UPV is automatically generated by SWARD, given that the user specifies for a given relational database a *property mapping table* that declares how RDFS properties (*mapped properties*) correspond to viewed relational columns in the UPV. The user also specifies a *class mapping table* that declares how RDFS classes (*mapped classes*) correspond to relational tables in the UPV.

In other words, an RDFS view over a relational database in SWARD is implemented as a UPV that defines an RDFS meta-data description of the database where the database schema is encoded as classes and properties in the description and the database content is encoded as members of these classes and instantiated properties for these members.

To represent ER entities and relationships in the UPV its definition is augmented with special purpose subviews to model class memberships (*class membership views*) and class relationships (*class relationship views*). The definitions of these views are automatically generated by SWARD.

Composite keys are supported by relational databases but not in RDFS. However, since composite primary keys are very common in relational databases it is very important that they are represented in UPVs so that tables containing such keys can also be viewed in RDFS. The UPVs are therefore generalized to view tables with composite primary keys as well.

SPARQL queries to UPVs are very flexible and can mix schema and content. For example, a query can be expressed that finds the values of all properties (i.e. attribute values) of a given customer. Meta-data descriptions can have many properties (e.g. [61] has 1000s), and this requires efficient processing of queries involving many self-joins over the UPV.

A naive implementation is to define the UPV as an SQL view and to do all query processing over the UPV in a relational database. We show that such an approach scales very badly and is outperformed by the more efficient query processing strategies described in this Thesis.

A general query reduction algorithm, called *PARtial evaluation of Queries* (*PARQ*) for queries to complex views, such as UPVs, has been developed. The reduction algorithm is based on the program transformation technique *partial evaluation* [25][43][29]. Partial evaluation enables the development of elegant and clean solutions that are automatically specialized into efficient reduced programs. For UPVs, the PARQ algorithm is shown to elegantly reduce queries substantially before regular cost-based optimization by a back-end relational DBMS. PARQ simplifies the query by iteratively evaluating at compile time some application specific predicates until a fix-point is reached. We show dramatic improvements in query processing time for conjunctive SPARQL queries to UPVs, while increasing the query size, the size of the database schema over which the UPV is defined, and the database content size.

Queries to the UPV can be of three kinds: i), *content queries* that access the database content ii) *schema queries* that access relational schema information only, and iii) *hybrid queries* that combine schema and content data. SWARD can process queries of all three kinds efficiently. However, it is particularly challenging to process content and hybrid queries searching the large database contents. We show that partial evaluation substantially improves query processing performance for all three kinds of queries.

In summary, the following results are presented:

- UPVs are defined as a general method to map any relational database to a complete RDF Schema view requiring a minimum of user input.
- A new partial evaluation algorithm is developed called PARQ for reducing queries based on controlled partial evaluation of query fragments.
- The PARQ algorithm is shown to provide elegant and scalable processing of conjunctive SPARQL queries to large disjunctive UPVs. In particular, new query processing methods, END-P and DVS-P, are developed for efficient processing of queries to large UPVs, based on applying PARQ on straight-forward query processing methods. It is shown that that the application of PARQ dramatically improves performance of naïve approaches.

- The query processing method END-P is defined as PARQ applied on conventional query processing using view Expansion, Normalization, and Decomposition (END).
- The DVS-P method is PARQ applied on another naive method specialized for UPVs, DPS (Dynamic Plan Selection), which selects precompiled query fragments from a table.
- The query processing strategies were evaluated for scalability of query processing time as i) the database size increases, and ii) the size of the query and the size of the UPV definition increase.

To summarize, in this Thesis, the first research question, regarding automatic generation of general and complete RDFS views of relational databases was thoroughly investigated and answered with the exception of handling of relational tables encoding ER M:N relationships types and relationship types of degree higher than 2, that has yet to be looked into.

The second research question, regarding scalable processing of realistic size queries to RDFS views over relational databases, is fully investigated and answered for conjunctive content and schema queries. The research question was also investigated and answered for the most common subclass of hybrid queries but some work remains to answer the question for all hybrid queries and for disjunctive queries.

2 Background

This Chapter describes key technologies used in SWARD. It first gives an overview of relational database management systems. Then an introduction to the semantic web, its languages and query languages, is provided followed by a presentation of RDF repositories. After that an overview of how to view relational databases in terms of RDF Schema representations is presented. The general purpose program specialization technique called partial evaluation is then presented. After that, an overview of the ER-model is given together with a discussion of its relationship to RDF Schema and the relational model. The Chapter is concluded with a short presentation of the DBMS Amos II and how it is used for implementing SWARD.

2.1 Relational Database Management Systems

A database is simply a large collection of data managed by a database management system (DBMS). The DBMS allows for a) creation of new databases and specification of the logical structure of the database called its *schema* b) querying the data c) updating data d) concurrent access of multiple users to data.

To describe the structure of information in the database, its *schema*, the DBMS utilizes various data models, which provide primitives (metalanguage) for defining a schema. In a paper dated to the beginning of the 1970s Codd [18] proposed the *relational data model* where data were modelled as *rows*, or tuples, in 2-dimensional *tables* with one or more *columns*. The relational data model gained much in popularity due to a) its simplicity and b) closeness to the traditional way of structuring non-digital information in companies. It is by far the most common data model used in databases today. Notice that, while the user of a relational DBMS sees the data as simple tables, internally relational DBMSs organize the data using complex data structures providing efficient retrieval and manipulation.

Figure 1 shows a an example of a small relational database *E-government* with three tables, *LIFEEVENT*, *FORM,* and *SERVICE* representing information about life events, forms associated to life events, and services related to life events, respectively. Every table in the relational model has one or several columns acting as a *primary key* in that table, i.e. making every row in the table unique. In tables *LIFEEVENT* and *FORM* columns *LID* and *FID* are acting as primary keys, respectively. In table *SERVICE* the key is composed of the two columns *LID* and *SNR*. Such keys are called *composite keys*. A *foreign key* is a column referencing the key column in another table. In Figure 1 column *LIFEEVENT* in *FORM* is a foreign key referencing key column *LID* in table *LIFEEVENT*.

LIFEEVENT	LID	NAME	DESCR
	movinghouse	Moving House	A citizen intends to move from one EU country to another.

FORM	FID	URL	LIFEEVENT
	fid_0	http://www.skatteverket.se/…/7665B5.pdf	movinghouse
	fid_1	http://www.workpermit.com/uk/employer_form.htm	movinghouse

SERVICE	LIFEEVENT	SNR	TITLE
	movinghouse	0	Moving Service

Figure 1: E-government relational database

In the relational model the user specifies the query in a *high-level query language,* where the *Structured Query Language (SQL)* being the most widely used. Instead of specifying exactly *how* the information should be accessed in terms of traversing low-level data structures and indexes the user now can focus on *what* information should be accessed leading to an increasing productivity in database development.

The separation of query languages from low-level implementation specific details is one of the most fundamental aspects of relational DBMSs and is referred to as *data independence*. Data independence exists on a logical and on a physical level in a relational DBMS. On a logical level, changes to the database schema should not affect application programs accessing the schema through queries. On a physical level, changes to the data organization should not affect the database schema.

The technique of efficiently calculating the result from a high-level query is called *query processing* and is performed by the DBMS *query processor*.

Figure 2 shows the typical query processing steps in a DBMS [65].

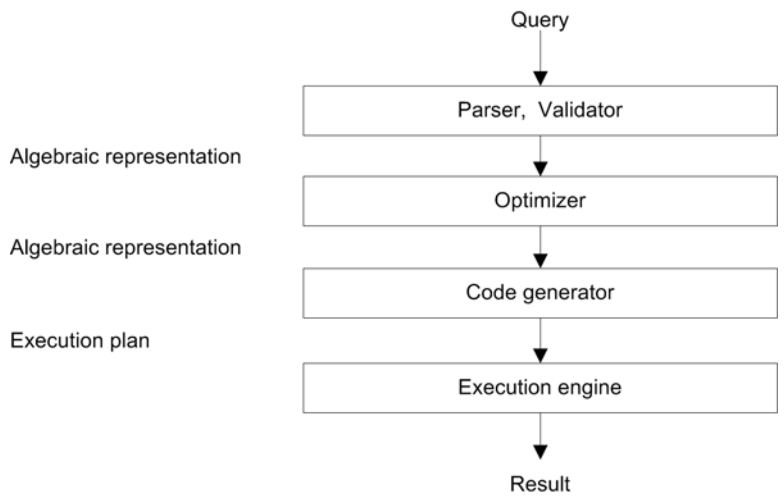


Figure 2: Typical query processing steps in a DBMS

First, the query is checked for syntactic and semantic correctness by the *parser* and *validator,* respectively. The semantic analysis involves, for example, checking that the query refers to only existing table and column names.

The result of parsing and validating the query is an algebraic representation of the query in the form of a *logical query plan* in a *relational algebra*, that is, a sequence of operators to be executed. It is logical in the sense that no algorithms have yet been assigned to implement the operators in the query plan.

Because a query often can be executed in numerous ways the task of the *optimizer* is to produce an efficient execution strategy. This is done in two steps where 1) the optimizer applies algebraic laws on relational algebra expressions to produce a more efficient query plan and 2) algorithms are assigned to the logical relational algebra operators of the query plan producing an algebraic expression in the form of a *physical query plan*. Each query plan has a predefined cost according to some cost model and the cheapest plan is picked. The 'cost' can for example be approximated by the number of disk access performed by the relational database management system when executing a specific query plan. The number of disk accesses is in turn affected by factors such as the ordering of similar operations and sizes of intermediate results. For the query optimizer to correctly estimate the cost of alternative query plans it is therefore important that there exist valid statistics about data characteristics such as number of rows in a table and the number of different values in table columns.

Finally, the resulting physical query plan is interpreted by the *execution engine* producing the result.

In this Thesis a system for generation and querying RDF views over relational databases is presented. Queries in the RDF query language SPARQL are processed over RDF Schema representations of relational databases. The system generates SQL fragments that are sent to the relational database for cost-based optimization and execution.

2.2 Semantic Web

The semantic web effort aims for more focused and relevant web searches and to facilitate for the combination of information by providing internet-wide standards such as *RDF* [20][32] and *RDF Schema* [11], for semantically enriching and describing web data. The formal meaning of RDF and RDF Schema is defined in [28].

2.2.1 The Resource Description Framework (RDF)

The Resource Description Framework (RDF) [20][32] is a W3C standard for representation and description of web resources on the World Wide Web. In other words, it is a language for stating meta-data about web resources. In RDF the concept ‘web resource’ is interpreted as anything that can be identified on the Web. This is a very broad definition of a web resource that allows for representation of: a) things accessible through a network e.g. a web page or a picture, b) things that are not accessible through any network e.g. identifiers for human beings or books in a library, and c) abstract concepts such as, e.g., a ‘creator’.

RDF web resources, or *RDF resources*, are uniquely identified through *Uniform Resource Identifiers* or *URIs* [32]. A URI reference is formed by a *URI namespace* and a local name. The namespace part of the URI can be a rather long string, e.g. *http://udbl.it.uu.se/schemas/eGovern#*. A more compact way of expressing URIs is by using a shorthand notation assigning a *prefix* to the URI namespace and adding the local name to this prefix. For example, in the URI *egov:Concern*, the prefix *egov:* is shorthand for *http://udbl.it.uu.se/schemas/eGovern#*.

With RDF any web resource can be annotated with *properties* describing its semantics. The value of a property for some RDF resource is another

RDF resource. This knowledge is represented in RDF as *triples* or *statements* of RDF resources <*s*, *p*, *v*>[1] where *s* is called the *subject* (modeling some entity), *p* is called the *predicate* (modeling some property of an entity) of s, and *v* is called the *object* (the value of *p*). To avoid confusion with ordinary programming terminology this Thesis uses the terms *property* and *value* instead of *predicate* and *object*. The terms RDF triple and statement will be used interchangeably throughout the Thesis.

For example, a statement

<egov:Form/fid_0, egov:Concern,egov:LifeEvent/movinghouse >

where *s* is *egov:FormID/fid_0*, *p* is *egov:Concern* and *v* is *egov:LifeEvent/movinghouse* expresses the fact that the form identified by the URI *egov:Form/fid_0* concern the life event identified by the URI *egov:LifeEvent/movinghouse*. The statement is represented in graph notation in Figure 3.

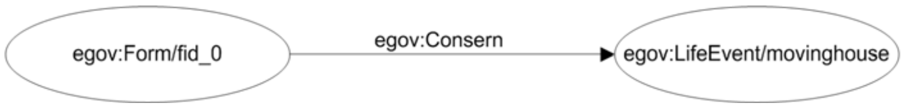


Figure 3: RDF graph

Each node in the graph is unique, meaning that if a resource would exist in more than one statement the node representing that resource would have several property arcs connected to it. RDF statements are exchanged by serializing them into a dialect of the *Extensible Markup Language* (*XML*) called *RDF/XML* [32].

The RDF/XML serialization of Figure 3 is shown in Example 1[2].

```
<xml version=”1.0”?>
<rdf:RDF xmlns:rdf=”http://www.w3.org/1999/02/22-rdf-syntax-
ns#”
  <rdf:Description rdf:about=egov:Form/fid_0>
    <egov:Concern rdf:resource=egov:LifeEvent/movinghouse>
  </rdf:Description>
</rdf:RDF>
```

Example 1: RDF/XML serialization of simple RDF graph

[1] We use the <.> notation to denote RDF statements, or triples, in text.
[2] By convention rdf: is prefix for http://www.w3.org/1999/02/22-rdf-syntax-ns#

The subject and property of an RDF statement are always URIs. However, the value could be either a URI as in Figure 3 or a literal as in Figure 4. Literals can be *simple* or *typed*. A simple literal is a string and a typed literal is a string adorned with a datatype URI. The statement represented in graph notation in Figure 4 expresses that the national tax board of Sweden is the creator of the form *egov:Form/fid_0*[3]. Here the value of the triple is a simple literal.

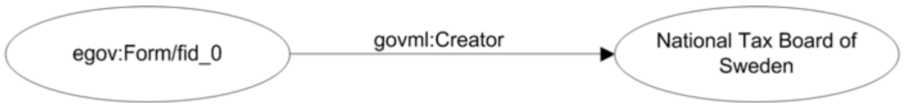


Figure 4: RDF graph with literal value

2.2.2 RDF Schema

RDF is the basis for representing semantic web data and several higher level languages are defined in terms of RDF. One of these higher languages is *RDF Schema* (*RDFS*) [11], a W3C standard that provides means to describe application specific classes of RDF resources and allow properties defined over these classes. Usually, RDF data is defined in terms of such an RDFS meta-data description. Examples of RDFS descriptions (referred to in this Thesis as RDFS descriptions) are RSS 1.0 [49] and WordNet [69].

The RDFS meta-classes *rdfs:Class* and *rdf:Property* are used to represent classes and properties in an RDF Schema description, respectively[4]. The RDFS meta-property *rdf:type* is used to define the data type of an RDF resource by associating each RDF resource with one or several RDFS classes. The class for which a property is defined is called the *domain* of the property and is represented by meta-property *rdfs:domain*. The class of the value of a property is called the *range* of the property and is represented by the meta-property *rdfs:range*. Figure 5 shows a small example description of an e-government portal with information about life events and their related forms.

The RDFS description contains the two classes; *egov:LifeEvent* and *egov:Form* modelling life events and forms about life events, respectively. The class *egov:Form* has the properties *egov:Concern* and *govml:Creator,* where the first one expresses which life event a form concern and the latter represents the creator of the form. The domain and range for *egov:Concern*

[3]govml: is prefix for namespace http://www.egov_project.org/GovMLSchema#
[4]By convention rdfs: is prefix for http://www.w3c.org/2000/01/rdf-schema#

are *egov:Form* and *egov:LifeEvent,* respectively. The property *egov:Creator* has the domain *egov:Form* and range *rdfs:Literal*. The mentioned classes are instanced with the RDF resources *egov:Form/fid_0* and *egov:LifeEvent/movinghouse* representing a form and its associated life event, respectively.

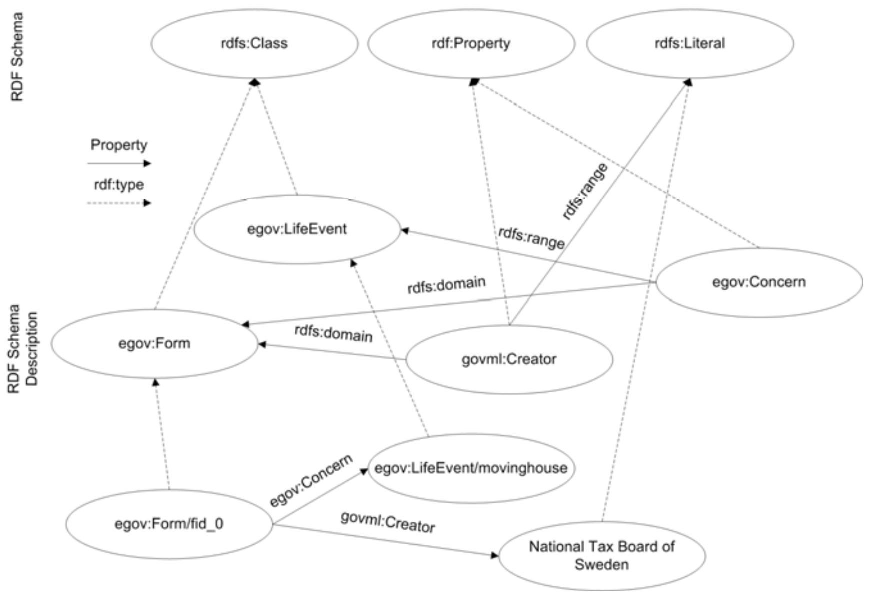


Figure 5: An example RDFS description of an e-government portal

A subset of the RDF triples representing the RDFS description and its data in Figure 5 are:

<egov:LifeEvent,rdf:type,rdfs:Class>
<egov:Form,rdf:type,rdfs:Class>
<egov:Concern,rdf:type,rdf:Property>
<egov:Concern,rdfs:domain,egov:Form>
<egov:Concern,rdf:range,egov:LifeEvent>
…
<egov:Form/fid_0,rdf:type,egov:Form>
<egov:Form/fid_0,egov:Concern,egov:LifeEvent/movinghouse>

The corresponding RDF/XML serialization is shown in Example 2. As illustrated below the format is meant for machines and not for humans to read.

```
<xml version="1.0"?>
<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:egov="http://udbl.it.uu.se/schemas/eGovern#">

  <rdf:Description rdf:about=egov:LifeEvent>
    <rdf:type rdf:resource=rdfs:Class>
  </rdf:Description>
  <rdf:Description rdf:about=egov:Form>
    <rdf:type rdf:resource=rdfs:Class>
  </rdf:Description>
  <rdf:Description rdf:about=egov:Concern>
    <rdf:type rdf:resource=rdf:Property>
    <rdf:domain rdf:resource=rdfs:Form>
    <rdf:range rdf:resource=rdfs:LifeEvent>
  </rdf:Description>
...
  <rdf:Description rdf:about=egov:Form/fid_0>
    <rdf:type rdf:resource=egov:Form>
    <egov:Concern rdf:resource=egov:LifeEvent/movinghouse>
  </rdf:Description>
</rdf:RDF>
```

Example 2: RDF/XML serialization for e-government portal

Another high level language for the description of RDF data is the *OWL Web Ontology Language* [42]. It is more powerful than RDFS in that it provides vocabulary for describing among other things *disjointness* between classes and *cardinality constraints* for properties. Unfortunately the language is computationally undecidable [42]. *OWL Lite* [42] is a simplified version of OWL, guaranteed to be computable, that supports classification hierarchies and simple cardinality constraints.

In this Thesis only basic RDF and RDF Schema languages are used for representing RDF Schema views of relational databases.

2.2.3 RDF Query Languages

Access to semantic web data is enabled through the development of RDF query languages. Several languages have been proposed e.g. SPARQL [58], RDQL [50], RQL [30], SeRQL [12], and QEL [40] where SPARQL is W3C standard.

SPARQL is an extension of the query language RDQL designed in accordance to the requirements described in the W3C RDF Data Group document 'RDF Data Access Use Cases and Requirements' [48]. SPARQL supports features such as a) generalized *triple patterns* i.e. a syntax for navigating RDF graphs where the user defines input graph patterns (queries) expressed as a conjunction/disjunction of RDF statements that are matched against the underlying RDF graph (data), b) comparison of values and support for data types, including arithmetic operations, c) *closure* i.e. the ability to construct a new RDF graph out of the result of a query, and d) *optional values* i.e. the possibility to partially match RDF graphs. An example of a simple SPARQL query that returns the creator of the RDF resource *egov:FormID/fid_0* is shown in Example 3.

```
SELECT ?v
WHERE {egov:FormID/fid_0 govml:Creator ?v .}
```

Example 3: SPARQL query

The result of the query is the literal 'National Tax Board of Sweden'.

In SPARQL variables are prefixed with '?'. The SELECT clause specifies the result. The WHERE clause specifies a selection condition over the RDF graph. The selections in a WHERE clause are specified using *triple patterns* [58] with syntax:

s p v .

In each triple pattern *s* (subject), *p* (property), and *v* (value) are constants or variables. A period, '.', denotes the end of a triple patterns. If more than one triple pattern is specified they are conjuncted. An optional FROM clause specifies the source to query. A SPARQL query with no FROM clause, like the one in Example 3, is executed against some default data source defined by the system processing the query. *Value constraints* can be defined with the FILTER keyword. They are conjuncted with the triple patterns.

Example 4 shows a SPARQL query that contains a pattern matching filter REGEX and returns all forms being created by tax boards.

```
SELECT ?v
WHERE {egov:FormID/fid_0 govml:Creator ?v .
       FILTER REGEX(?v, '.*tax board.*') .}
```

Example 4: SPARQL query with FILTER operator

The result of the query is the same as for the query in Example 3, i.e. 'National Tax Board of Sweden'.

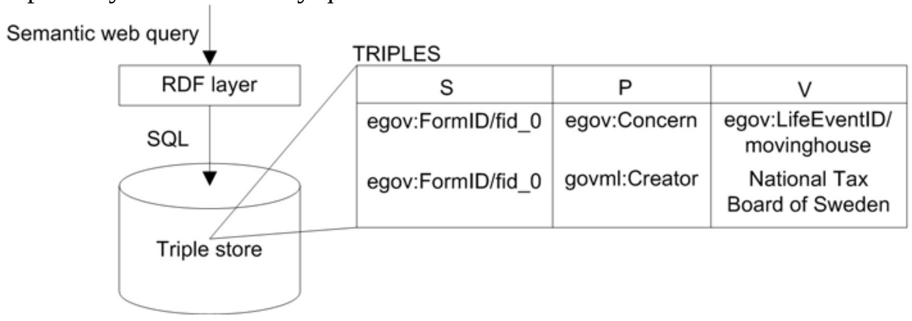
This Thesis is about the generation and querying of RDFS representations of relational databases in terms of RDF views. For simplicity, the language RDFS is chosen before OWL Lite to describe RDF data. The RDFS representations are queried in the standard semantic web query language SPARQL.

2.3 RDF Repository Systems

RDF repository systems offer support for storage of RDF data in special repositories designed for RDF and the ability to search RDF data using a query language. Examples of RDF repository systems are Jena2 [67], Sesame [12], Oracle [15], and AllegroGraph [4] where the first three are based on relational databases and the last one is a native RDF repository.

In relational RDF repositories, the main idea is to internally store all RDF triples in a table with three columns, *S*, *P*, and *V*, representing the subject, property and value of an RDF triple, respectively. URIs can be mapped to integer identifiers [15][12] to save space producing one table containing all the triples and another table with all URIs to identifier mappings. Many repositories (including [15][12][67]) implement a multi-layered approach where all RDF-specific processing (such as query translation) is done at the *RDF layer* above the back-end relational DBMS.

An example relational RDF repository with only one table, *TRIPLES*, storing all the triples that contain a subset of the data from Figure 5 is shown in Figure 6. Here URIs and literal values are represented as strings in the repository. For readability quotations are omitted.



S	P	V
egov:FormID/fid_0	egov:Concern	egov:LifeEventID/ movinghouse
egov:FormID/fid_0	govml:Creator	National Tax Board of Sweden

Figure 6: RDF repository

A semantic web query is first translated to SQL in the RDF layer and then passed to the relational DBMS for optimization and execution over the table storing the RDF statements.

```
SELECT ?creator, ?lifeevent
WHERE {?form govml:Creator ?creator .
       ?form egov:Concern ?lifeevent .}
```

Example 5: SPARQL query with two triple patterns

For example, the query returning the creator and the life event for all forms would be converted to the following SQL executed over the triple table, *TRIPLES*, in Figure 6:

```
SELECT T1.V, T2.V
FROM T1 TRIPLES, T2 TRIPLES
WHERE T1.S = T2.S           AND
      T1.P = govml:Creator  AND
      T2.P = egov:Concern
```

The query result variables *?creator* and *?concern* are bound to the RDF resources *'National Tax Board of Sweden'* and *egov:LifeEvent/movinghouse* respectively.

Notice that the SQL produced from the SPARQL query in Example 5 is a self-join over the table storing all triples, a *triple table join*. In general, every two triple patterns in the semantic web query will be translated into a triple table join in SQL.

SPARQL queries over relational RDF repositories with one table storing all the triples, like in Figure 6, can be very slow to execute since when the number of triples in the table is increased the triple table may not fit in main memory any more, meaning that each triple table join in the SPARQL query requires a disk access.

Another problem is that it is hard to access proper information about the distribution of values for different properties in an RDF repository with only one table storing all the triples. Insufficient statistics about the data will prevent the cost-based optimizer of doing a good job during query processing.

Furthermore, it is difficult to cluster data and decide which indexes to create since the repository is offered no information about the characteristics of data needed by the applications. Typically there will be indexes on each three columns of the table storing all the triples in the repository (e.g. columns *S*, *P*, *V* in Figure 6).

To speed up queries over relational RDF repositories, Jena 2 [67] and Oracle [15] allow non-triple representation of RDF properties. Properties that are often accessed together are clustered using so called *property tables* eliminating the need for triple table joins when the queries can be answered from a single property table.

However, in reality queries often need to combine data from many property tables. It is very unlikely that one property table holds all data and therefore the produced SQL gets complicated. Another problem is the unstructured nature of RDF data resulting in a lot of NULL values in property tables.

An alternative way of storing RDF data is to use one table for each property, i.e. column tables [1]. This way the size of the triple store can be kept smaller when data is highly unstructured. The column-based approach will require more SQL joins compared to the property-table based one but efficient merge join algorithms can be used for this [1].

In AllegroGraph [4] RDF triples are stored as objects in a store. Indices are used to speed up access of the triples.

This Thesis is about the processing of SPARQL queries. However, instead of queries to special purpose RDF repositories this work focus on querying *views* defined in terms of RDFS classes and properties that describes both structure and content of data stored in the relational databases This is important since most existing information still resides in relational databases, which are optimized for handling very large data volumes. By defining RDF views over relational databases they are made queryable with SPARQL without having to be copied to some RDF repository. Furthermore, a view will always reflect any changes in the relational database.

2.4 Mapping Relational Data to RDF

An RDF repository offers persistent storage of RDF data. There is a vast amount of additional high quality information stored in databases accessible from the web but not as RDF. This information should be exposed to the semantic web too. Two different approaches here referred to as *RDF materialization* and *RDF views*, have emerged for mapping data in relational databases to RDF. Whereas RDF materialization means loading relational data into large RDF repositories for persistent storage, the RDF view approach keeps the data in the source and instead provides RDF views over the data that can be queried with, e.g., SPARQL.

2.4.1 RDF Materialization

The RDF materialization approach for publishing relational data as RDF consists of two phases.

First, the relational database is materialized and duplicated as RDF data in some RDF repository such as for example [15].

In the next phase the RDF repository can be queried using some semantic web query language. Figure 7 illustrates the approach.

NASA for example uses the RDF materialization approach in their POPS project [27].

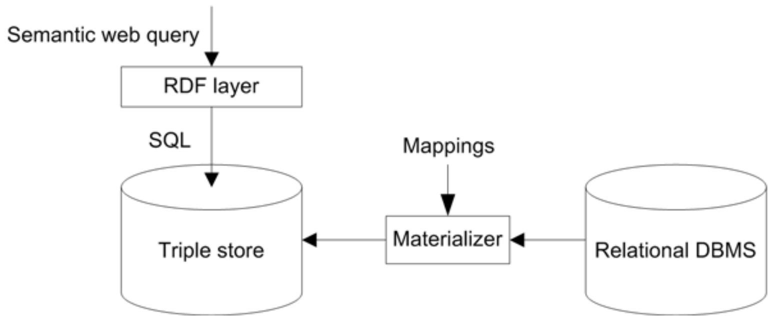


Figure 7: RDF materialization

The difference between the approach in Figure 7 and ordinary RDF repositories (Figure 6) is that when materializing relational data as RDF the data first has to be converted to RDF. This is done by the *materializer* taking the data in the relational database and a set of relational to RDF mappings as input and returning the materialized RDF data as output.

The mappings can be generic or separate user defined scripts to define the details. Generic mappings has to be specified by the user once only, scripts require separate user defined mapping files for each new RDF view. User defined mapping scripts are important because they provide the user with the possibility to define application specific mappings.

The RDF materialization approach can be very costly when the relational database is large. The fact that all data in the relational database is duplicated as RDF introduces a lot of data redundancy. Also, when the rate of change in the relational database is high, a lot of time is spent on propagating the changes to the RDF repository.

2.4.2 RDF Views

Rather that materializing all the contents of a relational database as RDF, with the RDF view approach data is streamed through the system when the RDF view of a relational database is queried. Retrieved data is not permanently materialized and stored in an RDF repository and the view will always reflect any changes in the wrapped relational database. Only meta-data describing the viewed relational database is stored in the RDF layer. Figure 8 illustrates exposing relational data as RDF using the RDF view approach.

Notice that only the result of a query has to be mapped to RDF as opposed to the RDF materialization approach where the whole relational database is converted and duplicated to an RDF representation.

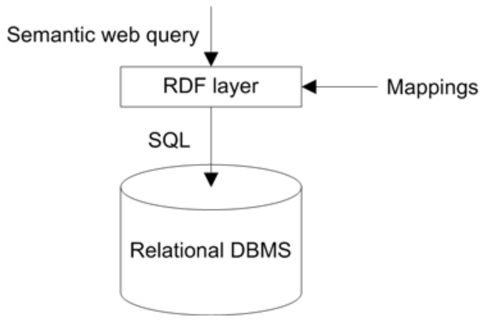


Figure 8: RDF view

Processing queries over RDF views in general is more demanding than processing queries over RDF repositories since a query stated in terms of several RDF triple patterns is translated into one SQL query fragment for each RDF triple pattern. These SQL query fragments then has to be combined to complete SQL queries over one or more relational tables.

In RDFS views over relational databases relational tables and columns are usually mapped to classes and properties in some RDFS description. The domain of a column is converted to the domain of its corresponding property. The range of a property is either a typed or a simple literal. The RDFS classes and properties describing the relational database schema are then instanciated in the RDF layer based on the corresponding schema meta-data in the relational database.

This Thesis is about generation and querying of RDFS views over relational databases i.e. the RDF view approach. No data is materialized as in an RDF repository. Each view is generated automatically given very simple user defined mappings between relational database tables and columns and RDFS classes and properties. The RDFS views are defined in a general way and are then specialized by partial evaluation (Section 2.6) during query processing. SWARD leverages on the Amos II system [51] for query processing and utilizes its facility for transparent access to external data sources to access the back-end relational database. No explicit user specification of SQL code is needed but the SQL queries are dynamically generated during query processing.

2.5 The Entity-Relationship (ER) Model

The Entity-Relationship model [14] is a high level conceptual data model used to graphically describe structure and constraints in a database. Such a high level model of information in the database has the advantage that it is easier to understand for non-technicians since it does not include implementation specific details. Furthermore it avoids misconceptions and multiple interpretations and it is implementation independent. The ER-model is used during the conceptual database design and is then translated to the relational model.

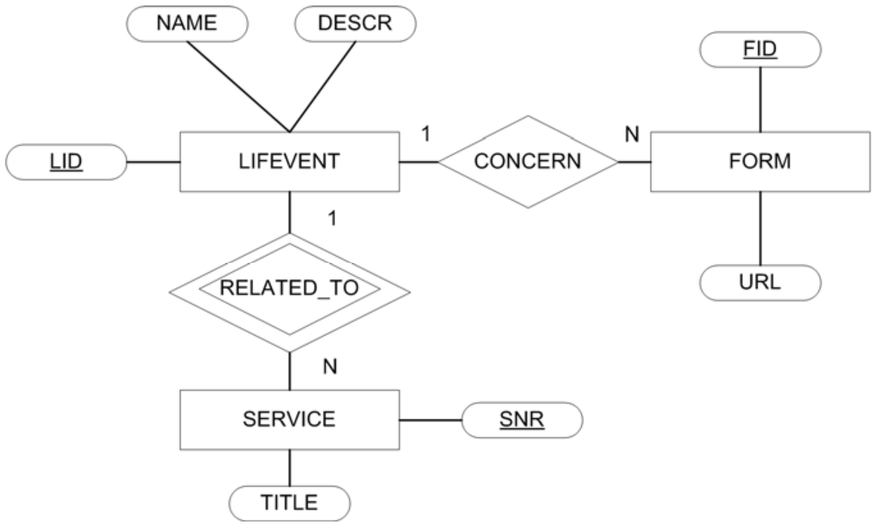


Figure 9:ER diagram of E-government database

Figure 9 shows an example of an ER diagram representing the *E-government* relational database presented in Figure 1.

With ER, *entity types* are used to describe a physical or abstract concept. In Figure 9 they are used to describe life events and their associated forms and services corresponding to the tables *LIFEVENT*, *FORM,* and *SERVICE* in the *E-government* relational database respectively.

Attributes are defined over entity types to describe their characteristics. For example, attributes *LID*, *NAME* and *DESCR* are defined over entity type *LIFEEVENT*. Attributes can be *simple* or *composite*. Each simple attribute is associated with a *value set* which specifies the set of allowable values for an attribute to take. For example, the domain of the *DESCR* attribute is the set of strings of alphabetic characters separated by blank characters. The domains are not explicitly declared in the ER diagram. Composite attributes can be broken up into smaller more basic subparts.

An *entity* is an individual instance of an entity type. Entities are not part of the ER-model, which describes meta-data only, and entities would correspond to the rows in a relational table. For example, an entity of the *LIFEVEVENT* entity type would be a row in the *LIFEVENT* table shown in Figure 1.

A *key* is an attribute that has unique value for every entity of the entity type it is defined over. For example, the attribute *FID* is a key for entity type *FORM*. A key composed out of several columns is called a *composite key*.

Entity types can be associated through *relationship types*. In Figure 9 the relationship type *CONCERNS* represent the relationship between life events and their associated forms i.e. a form concern a particular life event.

Cardinality constraints are defined over relationships. In Figure 9 the constraint 1:N between a form and a life event specifies that a life event can have several associated forms but a form can only concern one specific life event. Other cardinality constraints are 1:1 for one-to-one or M:N for many-to-many constraints over relationship types.

Sometimes an entity type does not have a key attribute of its own. For example, entity type *SERVICE* does not have any attribute uniquely identifying its entities. Entities belonging to such *weak entity types* are identified by being related to entities from a *strong entity type* with a key attribute. More specifically, the key of a weak entity type is formed by combining the key of its strong entity type with one or several attributes of its own (its *partial key*). The relationship type *RELATED_TO* representing the relationship between life events and services is an example of such an *identifying relationship*. Attribute *SNR* is partial key of *RELATED_TO*.

When an ER diagram is translated into the relational model, entity types with their attributes are represented as relations with columns. Attribute value sets define the domains of the columns. Key attributes defines primary keys in their respective table. ER relationship types are supported in modern relational DBMSs through the use of *foreign keys*. Cardinality constraints are used to guide the mapping of ER relationships to relational foreign keys. For an 1:N relationship like *CONSERN* in Figure 9 the primary key column in the table representing the entity type *LIFEVENT* is placed as foreign key in the table representing the entity type *FORMS* as illustrated in Figure 1. Figure 1 shows the resulting relations and columns from translating the ER diagram in Figure 9 to the relational model.

In Figure 9 relationship type *CONCERNS* is encoded as the foreign key column *LIFEVENT* in table *FORM* referencing the primary key column *LID* in table *LIFEVENT*. The identifying relationship type, *RELATED_TO* is encoded by augmenting the partial key of *SNR*, with the key of the *LIFEVENT* table forming the composite key (*LIFEVENT, SNR*) of table *SERVICE*.

Similar to the ER-model RDFS is also a high-level data model used for the description of information. In Figure 10 a summary of the correspondence between elements in ER, the relational model, and RDF Schema, is presented.

ER model	**Relational model**	**RDF Schema**
Entity type	Table	Class
Entity	Row	URI
Attribute	Column	Property
Value set	Domain of column	Range of property
Entity type of attribute	Table of column	Domain of property
Key attribute	Primary key	-
Binary 1:1 or 1:N relationship type	Foreign Key	Properties associating classes

Figure 10:Correspondence between ER, the relational model, and RDF Schema

For example, ER relationships, which are represented implicitly in the relational model through foreign keys, should be made explicit again as properties associating classes in an RDFS description.

Inheritance in RDF Schema could be modelled also using an Enhanced ER-model [23] supporting class inheritance. Since this work focus on how to query RDFS views over pure relational databases not supporting inheritance there is no need for support for EER constructs in the RDFS view provided by SWARD.

2.6 Partial Evaluation

Partial evaluation [25][43][29] is a technique for optimization of programs by specialization, given that some input data is known. Partial evaluation is illustrated in Figure 11.

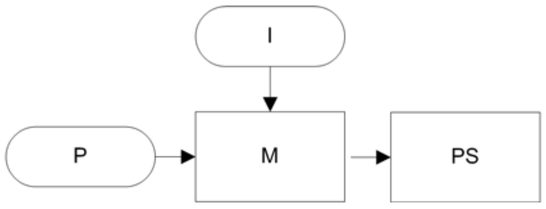


Figure 11: Partial evaluator M

A partial evaluator [29] (or specializer) is a function *M* that takes two arguments, the source of a program *P* and its static input *I*, and produces a specialized and more efficient program *PS*:

M(P,I) = PS

This is done by performing all calculations in *P* that depends only on known input data *I*.

Partial evaluation enables the programmer to develop ‘well-structured and cleanly written software’ [29]. The program is then specialized to a more efficient (faster and simpler) program producing the same output as the original program.

Application of partial evaluation can be found in several areas such as for example automatic compiler generation [6][7], operating systems [46], programming languages [6][7][33][53] and computer graphics [5].

In [6][7] partial evaluation is used to automatically generate compilers from interpreters. In [46] a commercial operating system is optimized by specializing the kernel code for system states that are likely to occur. In [5] a ray tracer, a method used in computer graphics to produce a good picture rendition of a scene, is specialized with respect to objects and light sources in the scene. In [6][7][33][53] partial evaluation of imperative, functional and logical programming languages is shown to produce faster specialized programs.

In this Thesis a new algorithm for partial evaluation of query fragments is presented called PARtial evaluation of Queries (PARQ). It is shown that PARQ provides scalable processing of real world semantic web queries to RDF Schema views of relational databases. In contrast to the query processing approach where the programmer introduces ad-hoc optimizations bound to introduce errors, partial evaluation is driven by well defined rules that reduce the query the to a much faster and simpler query producing the same output as the original one.

2.7 The Amos II System, Data Model and Query Language

AMOS II is a main memory functional DBMS [52]. The DBMS contains functionality for processing and executing queries over data stored locally but also external data sources, such as relational databases [24]. AMOS II provides transparent access to and hides the details of the data sources from users and application programmers.

AMOSQL is the declarative query language of AMOS II and can be described as an extended subset of the object-oriented parts of SQL:99. It is relationally complete. AMOSQL is based on the functional query languages OSQL [37] and DAPLEX [57]. AMOSQL queries are internally represented as *ObjectLog* [36] expressions. ObjectLog is an extension of Datalog [36] with disjunctions, objects, types, and external predicates.

The data model of Amos II is an object-oriented extension of the DAPLEX functional data model. It is founded on the three concepts; *objects*, *types* and *functions*.

All entities in the database are represented as objects and managed by the system. An object is either a *literal* or a *surrogate*. Literals are self-described objects that have no explicit object identifier (OID). They are maintained by the system and automatically garbage collected when no longer needed. Surrogates have associated OIDs and are explicitly created and deleted by the user.

Objects are classified into instances of types. Types are organized in a supertype/subtype hierarchy with multiple inheritance.

Functions model the semantics of objects e.g. properties of objects, computations over objects, and the relationship between objects. Basic functions can be classified into four different categories namely, *stored functions*, *derived functions*, *foreign functions*, and *database procedures*.

Stored functions represent properties (attributes) of objects in the database. For example, common properties of an object of type person are name and age. Stored functions also model relationships between objects.

Derived functions are defined in terms of other predefined functions or queries. They cannot have any side effects, e.g. they are not allowed to manipulate the database, and are compiled and optimized for later use.

Foreign functions are defined as external predicates in ObjectLog. The external predicates can have their inverses associated with them and this enables for *multi-directional foreign functions* in Amos II, which are invertible functions implemented in some external programming language. Handling of multi-directional foreign functions is an integral part of the query processing facilities of Amos II [52].

Database procedures correspond to methods with side effects and are defined using procedural AMOSQL statements.

The functional data model of Amos II is well suited for representing RDFS classes, properties and their instances [51] in terms of types and properties and objects. The system described in this Thesis, SWARD, utilizes the data model, query language and query execution of the Amos II system. However, is does not utilize object oriented aspects such as OIDs or inheritance. In SWARD, all RDFS data is represented as literals and is streamed through the system when the RDFS view of a relational database is queried.

SWARD extends and utilizes the Amos II system in the following ways:

- SWARD implements functionality in Amos II for automatic generation of RDFS views of large relational databases.
- SPARQL queries are first parsed into the internal ObjectLog language of Amos II before being transformed into a query plan by the Amos II query processor.
- SWARD extends the Amos II query processing facilities with general partial evaluation of query fragments expressed in ObjectLog. The technique of partial evaluating ObjectLog query fragments is shown to be critical for scalability reasons when processing queries over RDFS views.

3 The SWARD System

In this Chapter an overview of the SWARD system is presented. The system enables efficient processing of SPARQL queries over an automatically generated RDFS view, a universal property view (UPV), of the relational database. An example scenario is used to illustrate how a UPV is automatically generated, given user specifications of mappings between classes and properties in an RDF Schema description and corresponding tables and columns from an example relational database, *Company*.

The same scenario is used throughout the Thesis to demonstrate the processing of SPARQL queries to UPVs.

3.1 Overview

Figure 12 shows the SWARD system architecture.

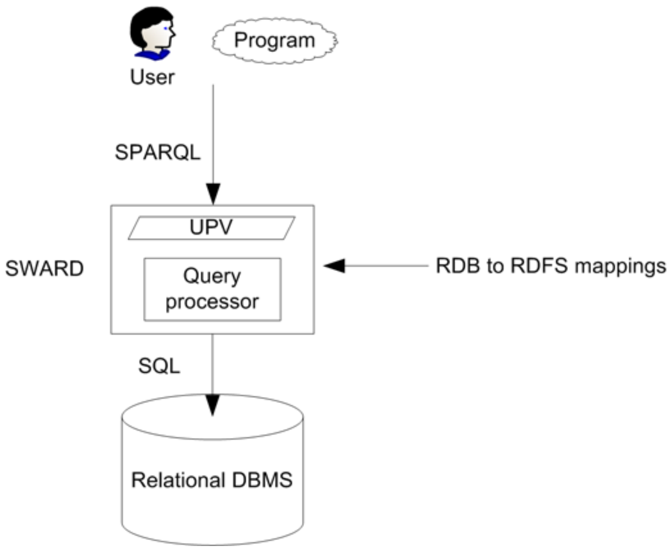


Figure 12: SWARD system overview

The system enables the user to query any relational database using SPARQL. The user could be a person sitting at a terminal or a program containing embedded SPARQL statements as strings.

A UPV is an RDFS representation of the content and schema in the viewed relational database. SWARD automatically generates a UPV over a relational database given a user defined set of mappings between tables and columns in the relational database and classes and properties in an RDFS meta-data description. Such mappings are provided by the SWARD system administrator upon UPV definition.

The SPARQL queries are simplified and translated into SQL fragments by the *query processor*. The SQL is dynamically generated after query simplification (reduction) based on the mappings between elements in the RDFS description and relational database schema constructs that are provided by the SWARD administrator. The generated SQL queries are sent to the relational database for cost-based optimization and execution. The results of the SQL queries are post-processed by SWARD to evaluate those query fragments that cannot be handled by the relational back-end, e.g. construction of URIs uniquely identifying RDF resources based on the provided mappings.

3.2 Scenario

Here an example scenario is presented to demonstrate the generation and querying of UPVs in SWARD.

A small ER diagram is presented in Figure 13. It is used during the design of the *Company* database to model customers (entity type *CUSTOMER*), orders (entity type *ORDERS*) and the relationship between them (*PLACED_BY*) denoting that an order is placed by a customer.

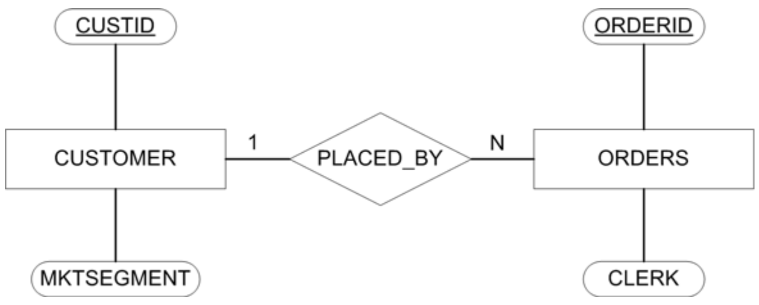


Figure 13: ER diagram of Company database.

The relationship between customer and an order is 1:N meaning that a customer can place more than one order but an order can only be placed by one customer. A customer has two attributes, an identifier (*CUSTID*) uniquely identifying a customer and a market segment (*MKTSEGMENT*). An order also has two attributes, a unique identifier (*ORDERID*) and the name of the person that filed the order (*CLERK*).

The ER diagram is translated to the relational model and is implemented in a relational DBMS as the database *Company* having the following two tables populated with one customer and two orders:

CUSTOMER	CUSTID	MKTSEGMENT
	120	AUTOMOBILE

ORDERS	ORDERID	OCUSTID	CLERK
	1	120	Wesson
	2	120	Doe

Figure 14: Company database

The columns *CUSTID* and *ORDERID* are primary keys in tables *CUSTOMER* and *ORDERS,* respectively. The relationship between entity types CUSTOMER and ORDERS is represented by the column *OCUSTID* in table *ORDERS* that is foreign key for *CUSTID* of *CUSTOMER.* All values in the relational tables are strings but for readability quotation marks are omitted.

In SWARD relational databases are searched by SPARQL queries to a UPV. Before generating a UPV in SWARD a *data source* must be defined. Such a data source represents properties that SWARD needs in order to access the back-end relational database. In our example these properties are the database URL, which particular JDBC[5] driver to use, along with a username and a password to be used by SWARD when accessing the database. SWARD also needs specification of the catalog and schema used in the database. The following command stores in SWARD the properties of a UPV data source where the argument *DSName* is the name of the data source:

```
defineDS(DSName, URL, Driver, Catalog, Schema, UserName,
 PassWord);
```

For example,

```
  defineDS(‘COMPANYDS’,
‘jdbc:microsoft:sqlserver://localhost;DatabaseName=COMPANY’,
‘com.microsoft.jdbc.sqlserver.SQLServerDriver’
’COMPANYCATALOG’,
‘COMPANYSCHEMA’,
‘COMPANYMGR’,
‘12345’);
```

Other properties may be needed to access other database management systems. A data source has to be defined only once for every viewed rela-

[5] SWARD uses JDBC to connect to back-end relational databases.

tional database and the information is stored in a table in SWARD. This is done by the administrator of the SWARD system.

When the data source is defined, the administrator declares to the system the name of the UPV and its URI to be used when accessing the RDF view in SPARQL queries. This is done by calling a procedure:

```
defineUPV(DSName, UPVName, URL);
```

For example:

```
defineUPV('COMPANYDS', 'Comp',
 'http://udbl.it.uu.se/upv/comp/');
```

The *URL* argument is used in the FROM clause in SPARQL queries to uniquely identify the RDFS graph represented by the UPV named *UPVName*. A default UPV name can be specified for SPARQL queries with no FROM clauses:

```
defineUPV('COMPANYDS', 'Comp','');
```

Through the rest of this Thesis the UPV named, *Comp*, represented by the URL *http://udbl.it.uu.se/upv/comp/* is default in SPARQL queries.

SWARD can contain several data source definitions and each data source can be used by different UPVs to access a viewed relational database. In our example, for simplicity, we limit the number of UPVs and data sources to one.

To generate a UPV for the database, SWARD requires the user to provide two tables that specify what tables and columns in the relational schema to view in RDFS data, the *class mapping table*, *cMap*, and the *property mapping table*, *pMap*.

The class mapping table, *cMap(Table,UPV,ClassID)*, maps 1:1 between a relational table name (*Table*), a UPV, and a *class identifier* (*ClassID*), representing a class in the UPV mapped to a table in the relational database. Such classes are referred to as *mapped classes* (Figure 15)[6]. A class identifier is a special purpose URI constructed out of a prefix and a local name where the local name is the name of the mapped class in the UPV. In Figure 15 the name of the class mapped to relational table *CUSTOMER* given the prefix *co:* is *Customer*. The rows in tables represented by mapped classes become *mapped instances* of that class.

[6] We use co: as prefix for namespace *http://udbl.it.uu.se/schemas/company#*.

Table	UPV	ClassID
CUSTOMER	Comp	co:Customer
ORDERS	Comp	co:Orders

Figure 15: Class mapping table cMap

The property mapping table, *pMap(Table,Column,UPV,PropID)*, (Figure 16) maps 1:1 between a viewed relational column (*Column*) in a table and a *property identifier* (*PropID*) representing properties in the UPV of the relational database. Such properties are called *mapped properties*.

Table	Column	UPV	PropID
CUSTOMER	CUSTID	Comp	co:CustID
CUSTOMER	MKTSEGMENT	Comp	co:Market
ORDERS	ORDERID	Comp	co:OrderID
ORDERS	OCUSTID	Comp	co:OrderCustomer
ORDERS	CLERK	Comp	co:Clerk

Figure 16: Property mapping table pMap

Analogous to class identifiers, property identifiers also have a local name. In Figure 16 the local name of the mapped property associated to relational column *CUSTID* given the prefix *co:* is *CustID*.

The UPV itself is an RDFS description of the back-end relational database in terms of mapped classes and properties in *cMap* and *pMap*. Here the UPV, *Comp*, represented by the URL *http://udbl.it.uu.se/upv/comp/* identifies an RDFS description of the *CUSTOMER* and *ORDERS* tables in the example relational database *Company*.

Given *cMap* and *pMap*, the following command in SWARD automatically generates the UPV for the database:

```
ViewRDB('Comp')
```

Here, the procedure *ViewRDB* generates a UPV named *Comp* for the database named *Company*.

A generated UPV *U*, is defined as a union of two subviews, one representing the schema of the relational database, the *schema view S*, and one representing its contents, the *content view C*, i.e. $U=S \cup C$.

Given the above property mapping table, the extent of the content view *C* of the UPV *Comp*, will contain the triples in Figure 17. All values in the UPV are strings but for readability quotation marks are omitted.

S	**P**	**V**
co:Customer/120	co:CustID	120
co:Customer/120	co:Market	AUTOMOBILE
co:Orders/1	co:OrderID	1
co:Orders/1	co:OrderCustomer	120
co:Orders/1	co:Clerk	Wesson
co:Orders/2	co:OrderID	2
co:Orders/2	co:OrderCustomer	120
co:Orders/2	co:Clerk	Doe

Figure 17: Content view for Company database

The schema view of a UPV, *S*, views relational database tables and columns as mapped classes and properties, respectively. Mapped classes are represented as instances of the RDFS meta-class *rdfs:Class* while mapped properties belong to meta-class *rdf:Property*. In SWARD the range of a mapped property is always a simple RDFS literal (RDFS class *rdfs:Literal).* URIs are treated as strings.

Given the property and class mapping tables in the example, Figure 18 shows the extent of the schema view *S* for table *CUSTOMER*.

S	**P**	**V**
co:Customer	rdf:type	rdfs:Class
co:CustID	rdf:type	rdf:Property
co:CustID	rdfs:domain	co:Customer
co:CustID	rdfs:range	rdfs:Literal
co:Market	rdf:type	rdf:Property
co:Market	rdf:domain	co:Customer
co:Market	rdfs:range	rdfs:Literal

Figure 18: Schema view for the CUSTOMER table in the Company database

The meta-properties needed in the schema view to define mapped classes and properties with their domains and ranges are *rdf:type, rdfs:domain*, and *rdfs:range*. In this Thesis they are referred to as *schema property identifiers* representing *schema properties*.

There are several other meta-classes and meta-properties in the RDFS specification [11] which are not needed for the mapping of relational databases to complete RDFS views. For example, the RDFS meta-properties *rdfs:subClassOf* and *rdfs:subPropertyOf* used to represent subsumption relationships between ontology classes and properties are not used in SWARD since there is no natural representation of such relationships in a relational database.

Notice that the user has to specify only the class and property mapping tables; the schema view is automatically generated in terms of these tables.

The user-defined tables *cMap* and *pMap* are small and stored in the main memory of SWARD. Furthermore, the view *S* is also small and is materialized in main memory to speed up query processing. Our partial evaluation algorithm will access these main memory tables intensively. It does not access the physical database at all.

Figure 19 shows how the elements of the relational data model are represented in UPVs. The handling of foreign keys through class relationship properties is explained later in Chapter 9.

Relational database element	**UPV representation**
Table	Mapped class
Column	Mapped property
Row	Mapped instance
Domain	Range of mapped property
Key	Mapped property
Foreign key	Class relationship property

Figure 19:Representation of RDF Schema elements in UPVs

3.3 Query Processor

SWARD transforms SPARQL queries into algebra expressions containing one or several calls to SQL for retrieving data from the relational database. Figure 20 illustrates the SWARD system query processing architecture.

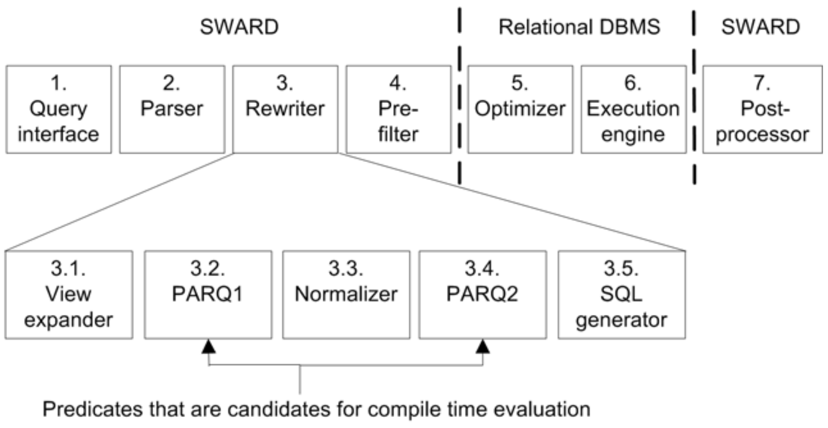


Figure 20: SWARD query processor

The user accesses SWARD through its *query interface*. The *parser* first translates the SPARQL query into ObjectLog [36].

The *view expander* recursively substitutes view references (ObjectLog rules/views cannot be recursive) with their definitions.

The steps *PARQ1* and *PARQ2* reduce the ObjectLog query by partial evaluation accessing the main memory tables *cMap*, *pMap*, and *S*. Depending on the strategy used *PARQ1* and/or *PARQ2* may not be executed.

The *normalizer* transforms the query to disjunctive normal form [3] (i.e., a union of conjunctive query fragments). Normalization improves query execution by combining in the same conjunctive query fragment predicates from the query and predicates from the property view definitions. Normalization produces efficient query execution plans but the cost for rewriting the resulting large expressions can be very high.

Finally, the *SQL generator* translates conjunctive query fragments in the normalized predicate into an algebra expression containing calls to SQL. The SQL calls are optionally preconditioned by a *pre-filter,* which is a predicate interpreted by SWARD that determines whether the SQL query should be selected for execution based on information stored in main memory system tables (*cMap*, *pMap*, and *S*) in SWARD. The SQL calls are shipped via JDBC to a commercial back-end relational database for cost-based optimization by its *optimizer* and evaluation by its *execution engine*. The algebra expression also contains a *post-processor* where such query fragments are evaluated by SWARD that cannot be handled by the relational back-end, e.g. construction of identifiers for mapped instances representing rows in relational tables.

The *query optimization phase* is defined as query rewriting (Step 3) plus the step to run the relational database optimizer (Step 5) in Figure 20. The *query execution phase* is defined as pre-filtering (Step 4), relational database query execution (Step 6), and doing post-processing (Step 7).

3.4 Universal Property View

The data handled by SWARD is the union of *content data* stored in the relational database and *schema data* that describe the contents of the relational database.

Assumption 1: The schema data and the content data do not overlap.

Assumption 1 is natural because of the separation of schema and data in relational databases.

We use ObjectLog to internally represent the UPV *U,* schema view *S* and content view *C.* As $U=S \cup C$ it has the definition

```
U(s,p,v) :- S(s,p,v) OR C(s,p,v)
```

3.4.1 Schema View

The schema view *S* in a UPV is defined as:

```
S(s,p,v) :- Classes(s,p,v) OR
            Domains(s,p,v) OR
            Ranges(s,p,v)
```

The *class view*, *Classes(s,p,v),* defines the mapped classes and properties in the UPV. The *domain view*, *Domains(s,p,v)* specifies for every mapped property as its domain the mapped class associated to the table owning the column associated to the mapped property. The *range view, Ranges(s,p,v),* specifies the values of mapped properties as always being literals.

Example 6 shows the class view definition for the example database. Lines 1-3 define all mapped classes for all viewed relational tables (*table*) in the class mapping table *cMap*. Lines 4-6 define all mapped properties for all viewed relational columns (*column*) in the property mapping table *pMap*.

```
Classes(s,p,v):-
1.(cMap(table,‘Comp’,s)        AND
2. p = rdf:type                AND
3. v = rdfs:Class)                 OR
4.(pMap(table,column,‘Comp’,s) AND
5. p = rdf:type                AND
6. v = rdf:Property)
```

Example 6: Class view definition

The domain view is defined as:

```
Domains(s,p,v):- pMap(table,column,’Comp’,s) AND
                 p=rdfs:domain               AND
                 cMap(table,’Comp’,v)
```

It states that the domain of a mapped property is the mapped class associated with the table in which the column associated to the mapped property exists.

Finally, the range of any property mapped to a relational database column is always a literal, i.e.:

```
Ranges(s,p,v):- pMap(table,column,’Comp’,s) AND
                p = rdfs:range              AND
                v = rdfs:Literal
```

The views defined above (*Classes*, *Domains,* and *Ranges*) are small, and do not access the back-end relational database. Furthermore, they do not change during the lifetime of the UPV. Therefore, they are materialized in SWARD during UPV generation into the main memory table *S* representing the schema view of the UPV.

3.4.2 Content View

The content view *C* of a relational database for a UPV is defined as a union of internal *property views* PV_p where one property view is generated for each mapped property *p* i.e.

$$C = \bigcup_p PV_p.$$

Example 7 shows the generated definition of *U* for the example UPV with *C* expanded on lines 3-7. Notice that the number of mapped properties will be large, since real-world relational databases contain many columns, so the disjunctive expression will be very large. The schema view is referenced on line 2. By convention, here, all property views are prefixed with '*P_*'.

```
1.U(s,p,v):-
2.S(s,p,v)                OR
3.P_CustID(s,p,v)         OR
4.P_MktSegment(s,p,v)     OR
5.P_OrderID(s,p,v)        OR
6.P_OCustID(s,p,v)        OR
7.P_Clerk(s,p,v)
```

Example 7: UPV definition

Example 8 shows the definition of the property view *P_MktSegment*.

```
1.P_MktSegment(s,p,v) :-
2.customer(custid,v)                        AND
3.cMap('CUSTOMER','Comp',cid)               AND
4.iMap(cid,custid,s)                        AND
5.pMap('CUSTOMER','MKTSEGMENT','Comp',p)
```

Example 8: Property view P_MktSegment

Line 2 accesses the relational table *CUSTOMER*. Line 3 accesses the class mapping table to get the class identifier *cid* for the mapped class associated to the table *CUSTOMER* and the UPV *Comp*. The external predicate *iMap* on line 4 generates a unique URI, *s*, representing a row in the table by string concatenation of the class identifier *cid* and a key *custid*, e.g. *co:Customer/120*. Such *instance identifiers* represent instances of mapped classes in the UPV. Line 5 retrieves the property identifier *p* representing the property named *Market* in *pMap* that is mapped to the column *MKTSEGMENT*.

An instance identifier is a special purpose URI constructed out of a prefix and a local name where the local name is the name of the mapped instance in the UPV, given the prefix. For example, the name of the mapped instance associated with the row in table *CUSTOMER* with primary key value '120' is given the prefix *co:* is *Customer/120*.

The external *iMap* predicate is invertible to be able to as well obtain the key for a given mapped instance identifier by parsing the identifier string.

In general, a property view has the structure in Figure 21 where variable *k* is bound to the primary key value of the relational table (*table*) and variable *v* is bound to values from the relational column (*column*) mapped by the property identifier *p* in the *pMap* table. Brackets [] are used to represent names substituted by the UPV generator. On line 2 the predicate [*table*] accesses the relational table to relate the value *v* of the table column representing property *p* to the primary key *k*. Table names substituted for predicate [*table*] accessing a relational database are always lowercased before substitution (line 2 Example 8).

```
1.P_[column](s,p,v) :-
2.[table](k,v)                                            AND
3.cMap([table],[upv],cid)                                 AND
4.iMap(cid,k,s)                                           AND
5.pMap([table],[column],[upv],p)
```

Figure 21: Property view definition

4 Query Classes

In this Chapter three classes of SPARQL queries to UPVs are defined together with examples of queries from each class. *Content queries* access only the database contents. *Schema queries* retrieve schema data without accessing the database contents. A *hybrid query* combines schema and content data. Queries that access data outside the relational database are not covered.

4.1 Content Queries

Definition 1: A *content query* is a query where the properties in all triple patterns are constant URIs that identifies mapped properties in the UPV. Such triple patterns are called *mapped property patterns*.

Content queries search the relational database contents. For example, the SPARQL query Q1 in Example 9 selects from the UPV for order number ‘1’ the market segment *mkt* of the customer *cust* placing the order.

```
SELECT ?cust ?mkt
WHERE {?order co:OrderID ‘1’ .
       ?order co:OrderCustomer ?ocust .
       ?cust co:CustID ?ocust .
       ?cust co:Market ?mkt .}
```

Example 9: Content query Q1

Query Q1 is a content query because all properties in the WHERE clauses are bound to constant identifiers of mapped properties. It will return the following result tuple when executed:

```
(co:Customer/120, ’AUTOMOBILE’)
```

co:Customer/120 is a system generated identifier (URI) representing a mapped instance of the mapped class named *Customer* by concatenating its class identifier *co:Customer* with the key value ‘120’ in table *CUSTOMER*.

SWARD answers content queries by generating SQL queries that searches the relational database.

4.2 Schema Queries

Definition 2: A *schema query* is a query where the properties in all triple patterns are constants that identify schema-properties in the UPV. Such triple patterns are called *schema property patterns*.

For example, SPARQL query Q2 in Example 10 is a schema query that selects from the UPV all mapped properties *prop* whose domains are *co:Customer*, except the property *co:CustID.* That is, Q1 finds the mapped property identifiers for the non-key columns in table *CUSTOMER.* It is a schema query because the only property identifier *rdfs:domain* in the WHERE clause is a schema property identifier.

```
SELECT ?prop
WHERE {?prop rdfs:domain co:Customer .
       FILTER (?prop != co:CustID) .}
```

Example 10: Schema query Q2

In SPARQL value constraints enclosed by '(' and ')' can be defined with the FILTER keyword. Q2 contains the filter **!=** (not equal).

The query returns the tuple:

```
(co:Market)
```

Since the schema property identifiers of schema queries do not represent any mapped property (Assumption 1), they can be answered by accessing only the small main memory table *S* representing the materialized schema view. We will show that partial evaluation removes the access to the content view *C* in the UPV definition.

4.3 Hybrid Queries

Definition 3: A *hybrid query* combines database schema and contents i.e. mixes schema property and mapped property patterns.

In this Thesis a large subclass of hybrid queries is investigated that dynamically selects some mapped properties from a class and access their values. Such queries are important since i) they allows for the user to query the viewed database without complete knowledge of the mapped classes and properties in the UPV and ii) they can be stated in a more compact way than their content queries counterparts. For example, query Q3 retrieves for a specific order all the mapped properties, *prop*, and values, *val*, of the customer placing the order except for the property *co:CustID*.

```
SELECT ?cust ?prop ?val
WHERE {?prop rdfs:domain co:Customer .
       ?order co:OrderID ‘1’.
       ?order co:OrderCustomer ?ocust .
       ?cust co:CustID ?ocust .
       ?cust ?prop ?val .
       FILTER (?prop != co:CustID) .}
```

Example 11: Hybrid query Q3

Query Q3 returns the tuple:

```
  (co:Customer/120, co:Market, ‘AUTOMOBILE’)
```

Notice that, given that the user has sufficient knowledge of the classes and properties in the UPV, query Q3 could also be stated as a content query but with additional triple patterns to get all mapped properties and their values from the mapped class *Customer*.

In Chapter 9 it is described how UPVs are augmented with class membership views to retain semantics from the relational database. Another subclass of hybrid queries dynamically selects mapped classes and accesses their mapped instances. Such hybrid queries are not investigated in this Thesis.

5 The PARQ Algorithm

A central technology used in the query processing of SWARD is partial evaluation. We have developed a new partial evaluation algorithm named PARtial evaluation of Queries (PARQ) based on evaluation of query fragments expressed in ObjectLog [36]. Our algorithm guarantees that the query never grows by partial evaluation but is often reduced in size. It is iterative and in each iteration it tries to reduce the query by compile evaluation of pre-specified primitive predicates. A primitive predicate can be a logical variable or constant, a table reference, or an external predicate reference. PARQ will stop when there are no more pre-specified primitive predicates left in the query to evaluate at compile time.

In the next chapters we illustrate how the algorithm enables efficient processing of queries to large disjunctive UPVs by applying it on conjunctive SPARQL queries parsed into ObjectLog. It is shown that PARQ reduces query expressions produced during the processing into much simpler expressions, which can be handled efficiently by a regular relational query optimizer. The algorithm is generally applicable on any query, but conjunctive SPARQL queries to UPVs are particularly suited since they are very complex with many embedded disjunctions that in turn contain further conjunctions. Furthermore, most subexpressions referenced in a query to a UPV can be eliminated by the partial evaluation of PARQ.

Recall that, in general, a partial evaluator [29] (or specializer) is a function *M* that takes two arguments, the source of a program *P* and a static (known) subset of the input *I*, and produces a specialized and more efficient program *PS*:

$M(P,I) = PS$

In the PARQ algorithm, *P* is a query fragment expressed as an ObjectLog predicate and *I* is a system table specifying the names of the primitive predicates that are candidates for evaluation at compile time. Initially *P* is the original query. In addition to *P* and *I* we provide as an extra argument to PARQ a list of the output (project) variables of the original query, *OV*, i.e:

$PARQ(P,I,OV) = PS$

If *OV* and *P* define a query to a UPV we will show that *PS* is substantially faster to evaluate than *P*.

PARQ specializes *P* by partial evaluating iteratively query fragments at compile time until no more reduction is possible. In SWARD, by evaluating at compile time only system predicates stored in main memory, the back-end relational database is not accessed during partial evaluation, so partial evaluation incur no extra disk accesses.

Figure 22 shows the pseudo code for the top level of PARQ. The algorithm is applied on an ObjectLog predicate *P,* which is a conjunction that can contain disjunctive expressions. The function *PC(P,I,OV)* partially evaluates conjunctions, while *PD(P,I,OV)* handles disjunctions. Line 1 in Figure 22 handles the case when *P* is an atom (logical variable or constant), line 2 when *P* is a conjunction, line 3 when *P* is a disjunction, and finally line 4 when *P* is a primitive (simple) predicate, other than a logical variable or constant, which is treated as a conjunction with one element.

```
function PARQ(P, S, OV)->PS
Input: P: a predicate
       I: a set of primitive predicate names being candidates
          for compile time evaluation.
       OV: output variables of the original query.
            These variables must remain in PS.
Output: Partially evaluated query PS
1.   if P is atomic then return P
2.   else if P is a conjunction then return PC(P, I, OV)
3.   else if P is a disjunction  then return PD(P, I, OV)
4.   else return PC(AND(P), I, OV) (a primitive predicate P is treated as a conjunction
                                     with one predicate)
```

Figure 22: PARQ algorithm

The pseudo code of the central iterative function *PC* is shown in Figure 23.

```
function PC(P, I, OV)->PS
Input: P: a conjunction
       I: a set of primitive predicates being candidates for
          compile time evaluation.
       OV: output variables of the conjunction
Output: Partially evaluated conjunction PS
1.   CHFLG := true
2.   while CHFLG is true
3.   do  if P is empty then return true
4.       CHFLG := false /* will be set to true if reduction made */
5.       for each conjunct C in P
6.       do  if C is a primitive predicate $c(a_1,\ldots,a_n)$ and $C \in I$
```

```
7.          then if all arguments $a_1,\dots,a_n$ are constants $c_1,\dots,c_n$
8.               then R := evaluate $c(c_1,\dots,c_n)$
9.                    if R = *false* then return *false*
10.                   else    remove C from P
11.              else if some $u_1,\dots u_k$ among $a_1,\dots,a_n$ are unknown .
12.                   then    try to execute the *probe query* { $u_1,\dots u_k$ | $c(a_1,\dots,a_n)$}
13.                           if the probe query succeeds
14.                           then    if no result is returned
15.                                   then return *false*
16.                                   else if the query returns
17.                                        exactly one tuple $v_1,\dots,v_k$
18.                                        then    remove C from P
19.                                                substitute in P all $u_i$ with $v_i$
20.                                                if $u_i \in$ OV
21.                                                then    add to P predicate $u_i$=$v_i$

22.                                        CHFLG := *true*
23.         else if   C is a primitive predicate $c(a_1,\dots,a_n)$ and there is another
                      predicate Q in P, $c(b_1,\dots,b_n)$, with equal key
24.              then substitute in P all non-key $b_i$ with $a_i$
25.                   remove Q from P
26.                   if $b_i \in$ OV then add to P predicate $b_i$=$a_i$
27.                   CHFLG = *true*
28.         else if   C is equality, c1=c2 where c1 and c2 are constants
29.              then if      c1≡c2
30.                   then    remove C from P
31.                           CHFLG = *true*
32.                   else    return *false*
33.         else if   C is a disjunction
34.              then C' := PD(C, I, OV') where OV' is
                          OV ∪ (freevars(C) ∩ freevars(P–C))
35.                   if C' = *false* then return *false*
36.                   else    if C' =/= C
37.                           then    replace C with C'
38.                                   CHFLG := *true*
39.         if CHFLG = *true* then leave for each /* reduction made */
40.  return P
```

Figure 23: Partial evaluation of conjunctions

The entire conjunction is replaced with symbol *false* if one of its predicates evaluates to *false* (line 9). The *probe query* on line 12 tries to evaluate at compile time a primitive predicate *C* where *name(C)* ∈ *I*. It fails if there are not enough known parameters to evaluate it (happens only for external predicates), in which case *C* cannot be compile time evaluated in the current iteration. If the probe query succeeds but returns no result (line 14) it means that the entire conjunctive predicate is *false*. If it returns exactly one result tuple (lines 16-17) the probe query is reduced by removing *C* (line 18) and substituting the variables in the probe result tuple (line 19). However, output variable assignments from the probe query are retained (lines 20-21). Important is that probe queries yielding more than one result tuple are *not* eliminated. This guarantees that the query is reduced in size in each round of the while loop on line 2, except for assigning the limited number of output variables in *OV*.

Lines 23-37 apply the following *key rewrite rule* [24]:

Key rewrite rule: In a conjunction *P*, two predicates *C* and *Q* with the same name and the same key parameters are equivalent. Therefore, they can be unified making all parameters of *C* and *Q* equal and *Q* can be removed from *P*.

On lines 28-32 equality predicates with both arguments bound to constants are specially treated. Equality is specially treated by SPARQ and equality cannot be in *I*.

Disjunctive subexpressions in a conjunction (lines 33-38) are handled by calling *PD* (Figure 24). When calling *PD*, the result variables in *OV* are augmented with the intersection of those free (unbound) variables in the disjunctive subexpression to partial evaluate, *freevars(C)*, that also exist in the rest of the query (*freevars (P-C)*). This is for avoiding elimination, by substitution, of variables that needs to be retained to produce the query result.

```
function PD(P, I, OV)->PS
Input: P: a disjunction
        I: a set of primitive predicates being candidates for partial
           evaluation.
        OV: output variables of the disjunction
Output: Partially evaluated disjunction PS
1.  for each disjunct D in P
2.  do  D’: = PARQ(D, I, OV)
3.        if D’=true then return true
4.        if D’=false then remove D from P
5.        else  if D’ =/= D
6.                then  replace D with D’
7.        if P empty then return false
8.  return P
```

Figure 24: Partial evaluation of disjunctions

Figure 24 shows the pseudo code for *PD*. The elements of the disjunctions are each partially evaluated by recursively invoking PARQ (line 2). If an element is partially evaluated to *true* the entire disjunction is reduced to *true*; if an element is reduced to *false*, it is removed from *P* (line 3-4). If an element is reduced to some other expression than *true* or *false*, it is replaced (lines 5-6).

The algorithm guarantees that the query never grows by compile time evaluating only primitive predicates producing no more than one result tuple. The execution is controlled by *I* to avoid compile time probing of expensive primitive predicates. The number of predicates that are tried for compile time evaluation by PARQ is therefore at most O(N^2), where *N* is the number of primitive predicates in *P* whose names are in *I.* This is because the iteration on line 5 in *PC* is restarted for every primitive predicate in *I* that is compile time evaluated and removed from *P*. Since compile time evaluation of primitive predicates in *P* never produces any new primitive predicates that are in *I* the number of predicates that are in *I* is reduced for each new iteration on line 5. In worst case every primitive predicate in *P* whose name is in *I* is compile time evaluated.

6 Query Processing

In this Chapter the need for partial evaluation when processing SPARQL queries to UPVs is demonstrated with a small example. This is followed by an overview of five different query processing strategies for answering SPARQL queries to UPVs.

6.1 Processing an Example Query

Example 12 shows how query Q1 in Example 9 is represented in ObjectLog.

```
1.query(cust,mkt) :-
2.U(order,co:OrderID,'1')           AND
3.U(order,co:OrderCustomer,ocust)   AND
4.U(cust,co:CustID,ocust)           AND
5.U(cust,co:Market,mkt)
```

Example 12: ObjectLog expression for Q1

Lines 5-10 in Example 13 show how line 5 in Example 12 is view expanded using the definition of *U* in Example 7.

```
1.query(cust,mkt) :-
2.U(order,co:OrderID,'1')             AND
3.U(order,co:OrderCustomer,ocust)     AND
4.U(cust,co:CustID,ocust)             AND
5.(S(cust,co:Market,mkt)            OR
6. P_CustID(cust,co:Market,mkt)     OR
7. P_MktSegment(cust,co:Market,mkt) OR
8. P_OrderID(cust,co:Market,mkt)    OR
9. P_OCustID(cust,co:Market,mkt)    OR
10.P_Clerk(cust,co:Market,mkt)
```

Example 13: Query Q1 after view expansion

After expanding the other three clauses in Q1 there will be 24 clauses, i.e. 20 references to property views and 4 references to the schema view. Expanding the property views generates 80 primitive predicates. With the 4 schema view references the total number of primitive predicates adds up to 84.

The SWARD query processor (Figure 20) then normalizes the view expanded queries before they are translated into algebra expressions by the SQL generator. Traditionally in database systems, normalization of queries is performed to simplify and make query processing more efficient [8][31][68][54]. A query could be normalized to either a disjunction of conjunctions (disjunctive normal form, or DNF) or a conjunction of disjunctions [3] (conjunctive formal form, or CNF). It will be shown that after view expanding and normalizing queries to UPVs, the number of primitive predicates is huge. Elimination of normalization is thus very important and is will be shown how partial evaluation by PARQ achieves this.

The normal form used in SWARD is DNF. Normalization of queries in SWARD, by combining in the same conjunctive query fragment predicates from the query and predicates from the property view definitions, improves query execution performance significantly, as will be shown. The reason is that DNF normalization enables only the relevant combined query fragment to be sent to the back-end relational DBMS.

Furthermore, when processing conjunctive SPARQL queries over disjunctive UPVs, normalization to DNF produces substantially smaller expressions than CNF, as explained below. Consider a conjunctive SPARQL query to a disjunctive UPV where Q is the number of clauses in the SPARQL query, UP is the number of property views in the UPV definition and CP is the number of primitive predicates in each property view. To simplify, the one reference to the primitive predicate S in each UPV definition is ignored (line 5 in Example 13). After view expansion an expression is produced on the following format:

K_1 *AND* $K_2,\ldots,K_q$ where

$K_i=P_1$ *OR* $P_2,\ldots,P_{up}$ and
$P_j=C_1$ *AND* $C_2,\ldots,C_{cp}$

Here, K_i is a clause in the query, P_j is a property view, and C_n is a primitive predicate.

The number of primitive predicates in the query after normalization to DNF is:

$UP^Q*CP*UP$

UP^Q is the number of conjuncts inside the generated disjunction and each conjunct contains $CP*UP$ primitive predicates.

The number of primitive predicates after normalization to CNF is:

$CP^{UP}*Q*UP$

$CP^{UP}*Q$ is the number of disjuncts inside the generated conjunctive expression and *UP* is the number of primitive predicates in each disjunct.

The expression on CNF thus grows exponentially over the number of property views in the UPV definition while the expression on DNF grows exponentially over the number of clauses in the query. Since the number of clauses in the query is normally much smaller than the number of relational columns viewed in RDF, normalization to DNF is a better strategy than transforming the query to CNF.

For example, normalization to DNF of query Q1 without partial evaluation, to the UPV in Example 7 (including the schema view *S*), produces 18144 primitive predicates while normalization to CNF would produce 24576 primitive predicates. Q1 is a very simple query, so normalization makes it impossible to process most real-world queries to large UPVs. We will next investigate query processing performance further and show that PARQ applied on straight-forward query processing methods achieves scalability.

6.2 BE: Naive Back End

The naive *Back-End* (BE) strategy represents the *UPV* entirely in the back-end relational database as an SQL view. All query optimization is done by the back-end DBMS and SWARD is not used. With BE *cMap*, *pMap* and *S* are tables in the relational database:

```
CMAP(TABLE,UPV,CLASSID)
PMAP(TABLE,COLUMN,UPV,PROPID)
S(S,P,V)
```

The columns {*TABLE*,*UPV*} and {*UPV,CLASSID*} are primary and secondary keys in *cMap*. In *pMap* {*TABLE*,*COLUMN*,*UPV*} and {*UPV*,*PROPID*} are primary and secondary keys. In *S* the primary key is made up of {*S,P,V*}.

Example 14 shows the definition in SQL of UPV *U* over the example database.

```
CREATE VIEW CUSTIDV(S,P,V)
 AS SELECT CM.CLASSID + '/' +
     CAST(C.C_CUSTKEY AS VARCHAR(25)),
      PM.PROPID, CAST(C.C_CUSTKEY AS VARCHAR(25))
    FROM CUSTOMER C, CMAP CM, PMAP PM
    WHERE PM.TABLE = 'CUSTOMER'   AND
          PM.COLUMN = 'C_CUSTKEY' AND
          PM.UPV = 'COMP'         AND
          CM.TABLE = 'CUSTOMER'   AND
          CM.UPV = 'COMP'
…
CREATE VIEW CLERKV(S,P,V)
  AS SELECT CM.CLASSID + '/' +
      CAST(O.O_ORDERKEY AS VARCHAR(25)),
       PM.PROPID, CAST(O.O_CLERK AS VARCHAR(25))
     FROM ORDERS O, CMAP CM, PMAP PM
     WHERE PM.TABLE = 'ORDERS'   AND
           PM.COLUMN = 'O_CLERK' AND
           PM.UPV = 'COMP'       AND
           CM.TABLE= 'ORDERS'    AND
           CM.UPV = 'COMP'

CREATE VIEW U(S,P,V)
  AS (SELECT * FROM CUSTID)     UNION ALL
     (SELECT * FROM MKTSEGMENT) UNION ALL
     (SELECT * FROM ORDERID)    UNION ALL
     (SELECT * FROM OCUSTID)    UNION ALL
     (SELECT * FROM CLERK)      UNION ALL
     (SELECT * FROM S)
```

Example 14: UPV in SQL

Our performance measurements will show that the BE strategy, using the commercial DBMS, does not scale when the size of the database grows. This is because no normalization is done during query processing resulting in very poor execution plans containing SQL joins of unions of many subplans for each property view.

6.3 END: Expand-Normalize-Decompose

The straight-forward END strategy uses SWARD to pre-process the queries before sending SQL statements to the relational database for cost-based query optimization. As classical query processing, the END strategy does view expansion and normalization before generating SQL expressions. No predicate is partially evaluated.

The major problem here is that the normalized query becomes huge even for this simple example. Real world databases will have large UPVs and the SPARQL queries will contain many self joins over these UPVs, *UPV joins*, so END will have unacceptable performance.

However, each disjunct in the expression on DNF contains a pre-filter executed in SWARD that use *cMap*, *pMap,* and *S* and therefore, after normalization to DNF, only the single SQL expression (shown below) is sent to the relational DBMS:

```
SELECT C.CUSTID,C.MKTSEGMENT
FROM ORDERS O,CUSTOMER C
WHERE O.ORDERID = '1'       AND
      C.CUSTID = O.OCUSTID
```

All filtered-out SQL expressions select combinations of the query triple patterns and mapped properties not relevant to answer the query and are therefore not sent to the back-end DBMS.

Since this is the only SQL query executed by the back-end relational database, the produced plans actually scale with the size of the database, unlike BE. Thus END produces a scalable plan but with very high query processing cost that grows exponentially with the size of the query because of normalization to DNF.

Next, we show that by applying PARQ on END the query is substantially reduced and normalization always eliminated for conjunctive content queries, which dramatically improves the query processing time.

6.4 END-P: END with Partial Evaluation

As will be shown, by applying PARQ on END with *I = {cMap,pMap,S}* the size of the UPV is dramatically reduced as no normalization is needed for conjunctive content queries. Since these predicates are stored in main memory, PARQ processing is very fast. We illustrate this by going through how PARQ reduces query Q1.

First, consider the view expansion of line 9 in Example 13 (*P_OCustID*) producing the expression in Example 15 (_ denotes dummy variables).

```
1.orders(orderid,mkt,_)              AND
2.cMap(‘ORDERS’,’Comp’,cid)          AND
3.iMap(cid,orderid,cust)             AND
4.pMap(‘ORDERS’,’OCUSTID’,’Comp’,co:Market)
```

Example 15: Expanded property view P_OCustID

Partial evaluation by step *PARQ1* of predicate *pMap* on line 4 will yield *false*. Thus, the entire view expanded expression on line 9 in Example 13 is eliminated (line 9 of the PARQ algorithm in Figure 23). Analogously lines 6, 8, and 10 in Example 13 will be eliminated by PARQ since the property identifier *co:Market* does not represent those columns and therefore *pMap* in the expanded column view definitions will be compile time evaluated to *false*. The schema view reference *S* (line 5 in Example 13) is also compile time evaluated to *false* (Assumption 1). Thus, step *PARQ1* replaces the entire disjunction on lines 5-10 in Example 13 with the conjunction from view expanding line 7, i.e. the query fragment in Example 16 representing column *MKTSEGMENT* in table *CUSTOMER*.

```
1.customer(custid,mkt)              AND
2.cMap(‘CUSTOMER’,’Comp’,cid)       AND
3.iMap(cid,custid,cust)             AND
4.pMap(‘CUSTOMER’,’MKTSEGMENT’,’Comp’,co:Market)
```

Example 16: Expanded property view P_MktSegment

Here the call to *pMap* on line 4 is evaluated at compile time to *true* and can be eliminated (line 10 Figure 23). Furthermore, compile time evaluation of the call to *cMap* on line 2 in Example 16 substitutes variable *cid* with *co:Customer* (lines 16-19 in Figure 23). Since *cid* is not among the output variables of Q1, the test on line 20 in Figure 23 is *false*. Thus PARQ replaces the disjunctive expression in the expanded *U* on line 5 in Example 12 with the following expression producing the desired property identifier (*co:Market*):

```
customer(custid,mkt)         AND
iMap(co:Customer,custid,cust)
```

Analogously, lines 2-4 in Example 12 are also view expanded and partially evaluated to different conjunctive single property view expressions, producing the query in Example 17.

```
1. query(cust,mkt) :-
2. orders('1',_,_)                        AND
3. iMap(co:Orders,'1',order)              AND
4. orders(orderid,ocust,_)                AND
5. iMap(co:Orders,orderid,order)          AND
6. customer(ocust,_)                      AND
7. iMap(co:Customer,ocust,cust)           AND
8. customer(custid,mkt)                   AND
9. iMap(co:Customer,custid,cust)
```

Example 17: Expanded and partially evaluated Q1

We see that there are four references to predicates representing relational tables, *ORDERS* and *CUSTOMER*, one for each conjunct in query Q1. Since *iMap* generates a unique mapped instance for each row in the relational table (i.e. the last parameter of *iMap* is a key) the system can infer from lines 7 and 9 that *ocust = custid* and then, using the key rewrite rule (lines 23-25 in Figure 23), substitute *custid* with *ocust* in the query and remove line 9 in Example 17. This makes *ocust* be the key in both calls to *customer* on lines 6 and 8, so these calls have equal keys and can be combined into a single *customer(ocust,mkt)*.

Analogously, the calls to *iMap* on lines 3 and 5 implies that *orderid = '1'* so *orders* on lines 2 and 4 can be combined into one call, *orders('1',ocust,_)*.

The key rewrite rule combines *iMap* and relational database calls to the same table before generating the SQL. It produces the fully reduced query in Example 18:

```
1. query(cust,mkt) :-
2. orders('1',ocust,_)             AND
3. iMap(co:Orders,'1',order)       AND
4. customer(ocust,mkt)             AND
5. iMap(co:Customer,ocust,cust)
```

Example 18: Fully reduced query Q1.

Finally, the SQL generator produces the following single SQL statement from lines 2 and 4 in the reduced query. The statement is sent to the back-end relational DBMS for cost-based optimization and execution. Notice that this is the same SQL statement as produced with the END strategy.

```
SELECT C.CUSTID,C.MKTSEGMENT
FROM ORDERS O,CUSTOMER C
WHERE O.ORDERID = '1'       AND
      C.CUSTID = O.OCUSTID
```

In this example, there is no pre-filter after PARQ has reduced the query. The execution plan will contain post-processing to construct two instance identifiers from the result of the SQL query (lines 3 and 5 in Example 18). The call to *iMap* on line 3 is actually not needed here and could be removed as explained in [24], but the cost of constructing instance identifiers is very cheap so the current implementation keeps this in the post-filter.

The example shows that PARQ applied on END significantly reduces the query and eliminates normalization. A single SQL statement is sent to the back-end database.

In general the following holds for END-P:

Theorem: For conjunctive content queries, the content view *C* is always reduced to a single conjunction after applying PARQ on *C* after view expansion.

Proof: All identifiers p_i in a conjunctive content query with condition *AND* $(C(s_i,p_i,v_i))$ are constants representing mapped properties. Every property view PV_p in the content view *C* contains a test to decide if p_i is identifier for the mapped property *p*. This test is made by calling *pMap* to see if p_i is mapped to the relational column viewed by PV_p (e.g. line 4 in Example 15). After view expansion and partial evaluation by PARQ (step *PARQ1* in Figure 20*)* of each query clause $C(s_i,p_i,v_i)$, *C* is thus replaced with the single conjunctive view expanded property view $PV_p(s_i,p_i,v_i)$ where p_i is an identifier for mapped property *p* (e.g. Example 16).

Corollary: For conjunctive content queries, each UPV reference, *U*, is always replaced by END-P with a single conjunction after applying PARQ on view expanded *U*.

Proof: Follows directly from Theorem and Assumption 1.

Disjunctive SPARQL queries are treated in SWARD as unions of conjunctive subqueries where END-P is applied on each subquery. This is not covered by this Thesis.

6.5 DPS: Dynamic Plan Selection

END-P becomes slower when the number of mapped properties increases because of the cost of view expansion and partial evaluation of the larger expanded query. The naive DPS (Dynamic Plan Selection) strategy eliminates view expansion by defining the UPV as selecting from a system table *pView* precompiled subplans for each property view:

```
U(s,p,v):- pView(p,'Comp',pvd) AND
           applyView(pvd,s,p,v)
```

The system table *pView(PropID,UPV,PropViewDef)* stores, for a given mapped property identifier (*PropID*), and UPV, the definition of the corresponding property view definition (*PropViewDef*) including a precompiled execution plan to retrieve the extent of *PropViewDef* from the database. To handle schema and hybrid queries, precompiled subplans are stored also for the RDF Schema meta-property identifiers *rdf:type*, *rdfs:domain,* and *rdfs:range*. Such subplans are defined in terms of the materialized schema view, *S*, and does not access the back-end relational database. These property views are referred to as *meta-property views*.

The system predicate *applyView(PropViewDef,S,P,V)* dynamically invokes the precompiled view definition *PropViewDef* yielding the RDF triples <*S,P,V*>.

After view expansion of the reference to *co:Market* in line 5 in Example 12 we get the expanded query in Example 19.

On line 5 the external predicate *pView* retrieves the property view definition *pvd* for the property identifier *co:Market* and the UPV *Comp*. Then *applyView* invokes the precompiled query plan *pvd* to retrieve the entire property view extent. Analogously the other references to *U* in Example 12 also retrieve entire property view extents. All selections are done as post-processing in SWARD after all referenced property views are downloaded, clearly a very inefficient execution strategy.

```
1.query(cust,mkt) :-
2.U(order,co:OrderID,'1')            AND
3.U(order,co:OrderCustomer,ocust)    AND
4.U(cust,co:CustID,ocust)            AND
5.pView(co:Market,’Comp’,pvd)        AND
6.applyView(pvd,cust,co:Market,mkt)
```

Example 19: View expanded Q1 using DPS

DPS decreases the query processing time by dynamically selecting at execution time only those precompiled property views used in the query. No normalization is needed but query execution is very slow since entire extents of all property views referenced in the query are shipped to SWARD and joined there. No predicates are partially evaluated.

6.6 DVS-P: Dynamic View Selection with Partial Evaluation

View expansion cannot be done with DPS since the actual property views are retrieved at run time. DVS-P (Dynamic View Selection with Partial evaluation) applies PARQ on DPS to select and expand those (meta-) property views referenced in the query. Normalization is not needed here either since the view expanded query is always conjunctive.

DVS-P is defined as DPS partially evaluated with *I = {cMap,pMap,S,pView}*. Furthermore, PARQ is extended with a special rule to handle delayed expansion of views, as will be explained.

Partial evaluation of *pView* in step *PARQ1* first selects the (meta-) property view. For example, on line 12 in Example 20 PARQ will probe *pView* to retrieve the property view *P_MktSegment* denoted *pView:MktSegment* in Example 20.

```
1.query(cust,mkt) :-
2.U(order,co:OrderID,'1')                          AND
3.U(order,co:OrderCustomer,ocust)                  AND
4.U(cust,co:CustID,ocust)                          AND
5.applyView(pView:MktSegment,cust,co:Market,mkt)
```

Example 20: Partially evaluated Q1 using DVS-P

Here the first argument of *applyView* (line 5) is the picked property view definition. If the first parameter of *applyView* is known at compile time, as in this case, the view could be expanded by the query processor. We therefore add to PARQ the following rule for *applyView* expansion after line 32 in Figure 23:

View expansion rule: If predicate *C* is *applyView(PropViewDef,S,P,V)* and *PropViewDef* is a constant then replace *applyView(PropViewDef,S,P,V)* with view expanded *PropViewDef(S,P,V)*.

Line 5 in Example 12 is thus replaced with the predicate *P_MktSegment(cust,co:Market,mkt)*, producing exactly the same query fragment as in Example 16 after view expansion.

The same substitutions of *applyView* are done for the other query clauses. The final steps, i.e. further query reduction and SQL generation are the same as for END-P. The difference between END-P and DVS-P is that with END-P we first view expand a large UPV definition and then apply PARQ to reduce it. With DVS-P we apply PARQ to pick and view expand only the pieces of the UPV referenced in the query before reducing it.

Our performance measurements verify that DVS-P scales excellently for content queries both with the size of the relational database, as well as with the size of the SPARQL query and the UPV definition. The back-end relational DBMS will execute the same single SQL query as END and END-P.

7 Performance Measurements

We measured the performance of BE, END, END-P, DPS, and DVS-P for content queries while scaling database size as well as query and UPV sizes.

The *query optimization time* is defined as the time to rewrite the query (Step 3 in Figure 20) plus the time to run the relational database optimizer (Step 5). The *query execution time* is the time for doing pre-filtering (Step 4), relational database query execution (Step 6), and executing the post-processing (Step 7).

The experiments were run on a DELL Optiplex GX270 with 2.2 GHz CPU, 512 MB main memory, and Windows XP Professional OS. We used a commercial relational database with 100 MB buffer size and the TPC-H [64] benchmark database generator for the data scalability tests. The profiling tool of the DBMS was used to measure how much time was spent in back-end query optimization and execution, respectively.

7.1 Scaling the Database Size

Figure 25 shows the times to execute query Q1 for the different strategies and a cold database while scaling the database size according to the TPC-H benchmark.

The execution plan produced by the relational DBMS in the BE strategy was examined using the query inspection tool of the commercial DBMS. It revealed that no normalization at all had been made and the large plan contained SQL joins of unions of a subplan for each property view in the UPV. Such an approach does not scale when the database size is increased (notice the logarithmic scale for the y-axis) since all property view extents are retrieved. However, it avoids the high cost of normalization so it will be able to handle large queries.

With END, the execution time scales, as expected. However, the query optimization time with END was 30 sec compared to only about 0.3 sec with BE. END-P and DVS-P send exactly the same SQL query as END to the back-end relational DBMS, with the same execution times.

DPS scales somewhat better than BE. The reason is that the execution plan of BE was very complex containing 48 joins accessing all property views, while DPS contains only 13 joins executed in SWARD to access only those property view extents required to answer the query. However, DPS still does not scale well as it retrieves entire property view extents, like BE.

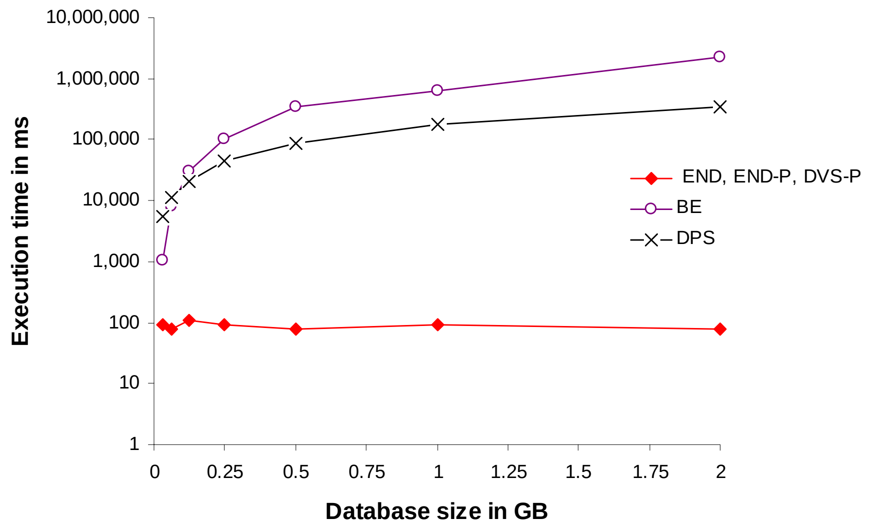


Figure 25: Execution times up to 2 GB (logarithmic scale)

7.2 Scaling the Query and Schema Sizes

In general, a SPARQL query in SWARD referencing *N* properties (in terms of *N* triple patterns) will have *N-1* self joins to the UPV. Since the UPV definition will usually contain a large number of mapped properties (relational databases usually have many columns), realistic size SPARQL queries will have many self joins of a complex UPV, and the system must be able to handle this efficiently. We measured the query optimization times as the sizes of the SPARQL query and the UPV definition were increased.

In order to scale the UPV, the size of a synthetic relational database schema was scaled by using a table generator function *createTbls(nt,nc)*, where *nt* is the number of generated tables and *nc* is the number of generated columns in each table. All tables T_i, $1 \leq i \leq nt$, have identical column names C_j, $1 \leq j \leq nc$ where C_1 is primary key.

For example, *createTbls(3,4)* generates these tables:

```
T1(C1,C2,C3,C4)
T2(C1,C2,C3,C4)
T3(C1,C2,C3,C4)
```

Every relational table T_i and column C_j is associated with a mapped class c_i and mapped property p_j in *cMap* and *pMap*, respectively.

The generated SPARQL queries are scaled by adding more triple patterns that retrieve an increasing number of properties from each mapped class.

```
SELECT ?v_{1,1},…,?v_{1,ppt},?v_{2,1},?v_{2,3},…,
        ?v_{2,ppt},?v_{3,1},?v_{3,3},…,?v_{nt,ppt}
WHERE {?s_{1} <p_{1,1}> ?v_{1,1} .
            …
       ?s_{1} <p_{1,ppt}> ?v_{1,ppt} .
       ?s_{2} <p_{2,1}> ?v_{2,1} .
       ?s_{2} <p_{2,2}> ?v_{1,1} .
       ?s_{2} <p_{2,3}> ?v_{2,3} .
            …
       ?s_{2} <p_{2,ppt}> ?v_{2,ppt} .
       ?s_{3} <p_{3,1}> ?v_{3,1} .
       ?s_{3} <p_{3,2}> ?v_{2,1} .
       ?s_{3} <p_{3,3}> ?v_{3,3} .
            …
       ?s_{nt} <p_{nt,ppt}> ?v_{nt,ppt} .}
```

Example 21: Scaling the SPARQL query size

ppt properties are extracted per mapped class c_i where $ppt \leq nc$. The property identifiers $p_{j,1}$ ... $p_{j,ppt}$ represent properties $p_{j,1}$ ... $p_{j,ppt}$ mapped to columns C_1 ... C_{ppt} of table T_i were $v_{i,1}$ is the value for each mapped property. The variable s_i holds mapped instances for the mapped class c_i. Identifiers for mapped properties are constants and enclosed with <...>. The scaled synthetic queries have the shape shown in Example 21.

Each table T_{i+1} is joined with table T_i with column C_2 in T_{i+1} equal to C_1 in T_i. Hence, our query is scaled up to *nt*ppt* SPARQL triple patterns producing *nt*ppt-1* UPV joins and *nt-1* SQL joins in the relational database extracting *nt**(*ppt-1*)+*1* mapped property values.

For example, *createTbls(2,4)* and *ppt=3* generates the SPARQL query:

```
SELECT ?v1_1, ?v1_2, ?v1_3, ?v2_1, ?v2_3
WHERE {?s1 <p1_1> ?v1_1 .
       ?s1 <p1_2> ?v1_2 .
       ?s1 <p1_3> ?v1_3 .
       ?s2 <p2_1> ?v2_1 .
       ?s2 <p2_2> ?v1_1 .
       ?s2 <p2_3> ?v2_3 .}
```

Figure 26 compares the optimization times for an increasing number of SPARQL triple patterns in the WHERE clause for a UPV over a single relational table with eight mapped properties (*nt=1, nc=8, ppt* varies from *1* to *8*). The optimization time for the BE strategy was measured by scaling the number of SQL self joins over a UPV definition of equal size in a synthetic SQL query constructed analogously to the SPARQL query in Example 21.

The SQL analogue using the BE strategy for the synthetic SPARQL query above is:

```
SELECT C1.V,C2.V,C3.V,C4.V,C6.V
FROM U C1,
     U C2,
     U C3,
     U C4,
     U C5,
     U C6
WHERE C1.P = <p1_1> AND
      C2.P = <p1_2> AND
      C3.P = <p1_3> AND
      C4.P = <p2_1> AND
      C5.P = <p2_2> AND
      C6.P = <p2_3> AND
      C1.S = C2.S   AND
      C2.S = C3.S   AND
      C4.S = C5.S   AND
      C5.V = C1.V   AND
      C5.S = C6.S
```

As expected, the optimization time with END deteriorates very fast with the number of triple patterns in the query.

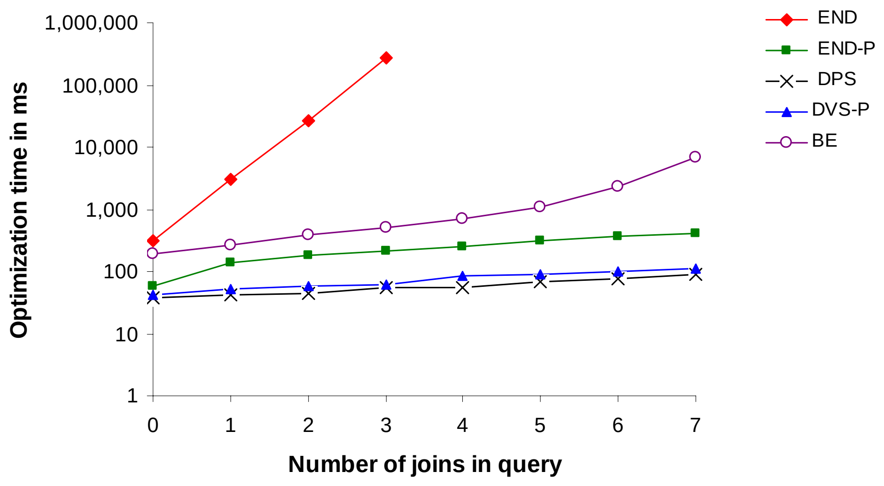


Figure 26: Optimization times up to 7 UPV joins in query

SWARD ran out of memory after four triple patterns i.e. three UPV joins. Logarithmic scale is used to be able to compare END and BE with the other strategies in the same diagram. END-P scales dramatically better than END since normalization is eliminated. DVS-P and DPS are faster than END-P by selective view expansion. The BE strategy scales better than END but is slower than the strategies based on PARQ.

To conclude, Figure 26 clearly shows that END produces unacceptable processing times for SPARQL queries of even very modest size (three UPV joins). However, by applying PARQ on END (END-P) the query processing scales well when the SPARQL query size is increased. Both DVS-P and DPS scale even better, but DPS has unacceptable execution performance. The END-P and DVS-P strategies outperform the naive BE with respect to both optimization and execution times.

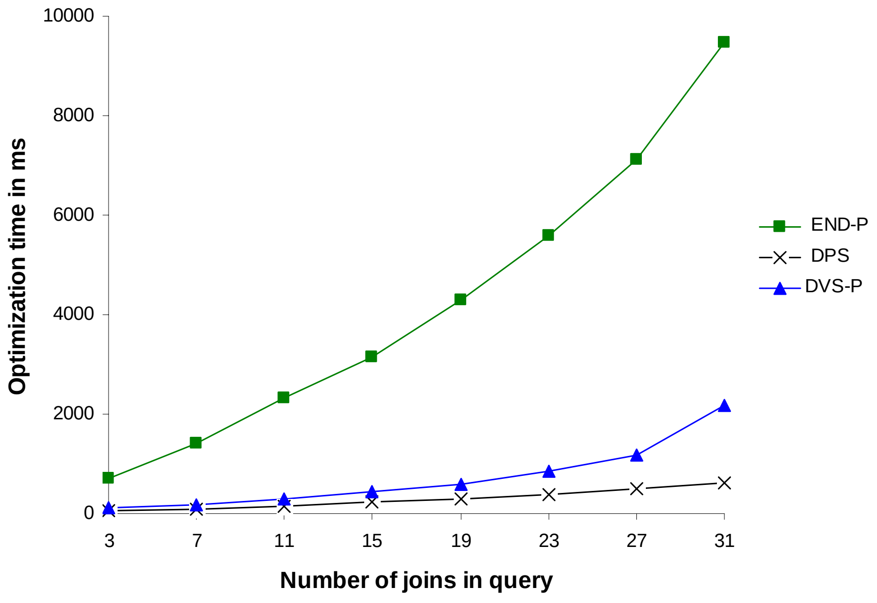


Figure 27: Optimization times up to 31UPV joins in query

To investigate query optimization performance for a very large UPV, Figure 27 compares END-P, DPS, and DVS-P for a database with *nt=8* and *nc=10* (i.e. 80 mapped properties). The query retrieves four mapped properties per mapped class (*ppt=4*) while scaling the number of joined tables in the back-end relational database to 8. Thus, there are up to 31 UPV joins in the WHERE clause of the SPARQL query.

The reason DVS-P is slower than DPS is the increasing cost of expanding many property views, while DPS simply selects precompiled execution plans without view expansion and application of PARQ.

In summary, our evaluations show that END-P and DVS-P produce efficient reduced conjunctive content queries while BE, END, and DPS do not scale. For content queries DVS-P scales best, since it further improves the query optimization time compared to END-P as the size of the query increases.

However, DVS-P requires PARQ to be able to infer exactly what views to expand, which is always possible for content queries where the properties in all triple patterns are constant URIs that identify mapped properties in the UPV (Definition 1), but not for hybrid queries, as will be shown in the next Chapter. There it is presented how END-P provides scalable query processing also for hybrid queries.

In Figure 28 a summary of all evaluated query processing strategies for content queries is presented. The optimization time of content queries for a strategy is characterized relatively to the other strategies. The execution of content queries is described as scalable or non scalable depending on the nature of the algebra expression produced in Step 3.5 in the SWARD query processor (Figure 20).

Strategy	**Optimization**	**Execution**	**Compile time candidates *I***
BE	Fast	Non scalable	{}
END	Very slow	Scales	{}
END-P	Fast	Scales	{*cMap*,*pMap*,*S*}
DPS	Very fast.	Non scalable	{}
DVS-P	Very fast	Scales	{*cMap*,*pMap*,*S*,*pView*}

Figure 28: Evaluation of query processing strategies for content queries

8 Processing Schema and Hybrid Queries

The processing of hybrid queries relies on how schema queries are processed. In this Chapter therefore, partial evaluation of schema queries is first described followed by how partial evaluation also improves processing hybrid queries dramatically.

8.1 Schema Queries

Expanding the WHERE clause of Q2 in Example 10 with the definition of *U* produces the following ObjectLog query:

```
query(prop):-
(S(prop,rdfs:domain,co:Customer) OR
 C(prop,rdfs:domain,co:Customer)     ) AND
prop != co:CustID
```

In this case *C* is eliminated by partial evaluation because for a schema query the meta-property identifier does not exist in *pMap* (Assumption 1) and therefore every property view PV_p in *C* evaluates to *false*.

The final query thus becomes:

```
query(prop) :-
S(prop,rdfs:domain,co:Customer) AND
prop != co:CustID}
```

Notice that *S* is not compile time evaluated here by PARQ since the mapped class identified by *co:Customer* has many properties and thus *S* yields more than one result binding for *prop*. The reduced query accesses only the materialized table *S* and does not access the back-end database at all.

8.2 Hybrid Queries

A hybrid query combines database schema and contents i.e. mix schema property and mapped property patterns (Definition 3).

Example 11 shows an example hybrid query Q3 that dynamically retrieves mapped properties from a class. Example 22 shows query Q3 in ObjectLog.

```
1.query(cust,prop,val) :-
2.U(prop,rdfs:domain,co:Customer)    AND
3.U(order,co:OrderID,'1')            AND
4.U(order,co:OrderCustomer,ocust)    AND
5.U(cust,co:CustID,ocust)            AND
6.U(cust,prop,val)                   AND
7.prop != co:CustID
```

Example 22: Hybrid query Q3 in ObjectLog

This is the kind of hybrid queries studied in this Thesis. Such a hybrid query is a conjunction between *schema clauses* selecting and binding variables to relational schema information (line 2), *content clauses* (lines 3-5) selecting content from the database, and *hybrid clauses* (line 6) dynamically retrieving mapped properties from a class by joining the schema and contents clauses.

Recall that content queries are defined to have all properties in the query bound to constant identifiers of mapped properties, and this is *not* the case for hybrid clauses. For example, on line 6 of Example 22, the variable *prop* is unknown. This prevents the reduction of *U(cust,prop,val)* to a conjunctive expression by PARQ.

However, as for query Q1, the schema clause on line 1 in Example 22 will be partial evaluated by step *PARQ1* into the single clause:

```
S(prop,rdfs:domain,co:Customer)
```

With the END-P strategy and *I = {cMap,pMap,S}* the content clauses on lines 3-5 in Example 22 are reduced by PARQ to simple conjunctive expressions (*Corollary*) and need no further discussion.

The hybrid clause on line 6 is view expanded in Example 23, lines 3-8, into a disjunction representing the entire UPV.

```
1.S(prop,rdfs:domain,co:Customer)     AND
2.//conjunctive expression from content clauses
3.(S(cust,prop,val)            OR
4. P_CustId(cust,prop,val)     OR
5. P_MktSegment(cust,prop,val) OR
6. P_OrderID(cust,prop,val)    OR
7. P_OCustID(cust,prop,val)    OR
8. P_Clerk(cust,prop,val)           ) AND
9. prop != co:CustID
```

Example 23: Expanded hybrid clause

Notice that property view definitions always contain a call to *pMap* that bind its property identifier. For example, in Example 8 that defines the property view *P_MktSegment*, line 5 binds *p* to *co:Market*. Therefore, every partial evaluation in step *PARQ1* of a property view in a hybrid clause explicitly binds *prop*.

We also notice that normalization combines the schema clause with the elements in expanded hybrid clauses. In particular, the schema clause of line 1 in Example 23 is conjuncted with each of the clauses on lines 4-8. Furthermore, the schema clause restricts the property identifier to those representing mapped properties from the class *Customer*.

The above observations implies that partial evaluation in step *PARQ2* eliminates the clauses on lines 4-8 that are not accessing mapped properties of class *Customer*, i.e. lines 6-8. The remaining clauses in the normalized expression are formed by combining lines 1, 2, and 9 with lines 3, 4, and 5.

At run time the conjunctive pre-filter formed by lines 1 and 3 evaluates to *false* since *prop* cannot be both a schema property identifier and a property identifier (Assumption 1). Also the pre-filter formed by lines 4 and 9 evaluates to *false* since the property identifier *co:CustID* should be excluded. The only executed SQL query is the one produced by lines 2 and 5:

```
SELECT C.MKTSEGMENT
FROM CUSTOMER C, ORDERS O
WHERE O.ORDERID = ‘1’       AND
      O.OCUSTID = C.CUSTID
```

In general, for such hybrid queries that dynamically retrieve mapped properties of a given class partial evaluation substantially reduces the queries by removing at compile time all query fragments for properties of other classes. The final expression is a disjunction if more than one mapped property is retrieved.

The reductions apply also for selecting properties from more than one class since such a query can be expressed as a union of queries accessing properties from single classes. This is not investigated in this Thesis.

8.3 Performance Measurements

We evaluated the query optimization time of the hybrid query Q3 by scaling the size of the UPV over the number of mapped properties of the class *Customer* extended with six more properties, starting with only one property and then successively increasing the number up to eight properties.

The optimization time of BE, END-P, DPS, and DVS-P for UPVs with increasing numbers of mapped properties in *pMap* is illustrated by Figure 29. Without partial evaluation (END) the SWARD optimizer runs out of memory even for the smallest UPV. The behaviour of the other strategies is similar as for content query Q1.

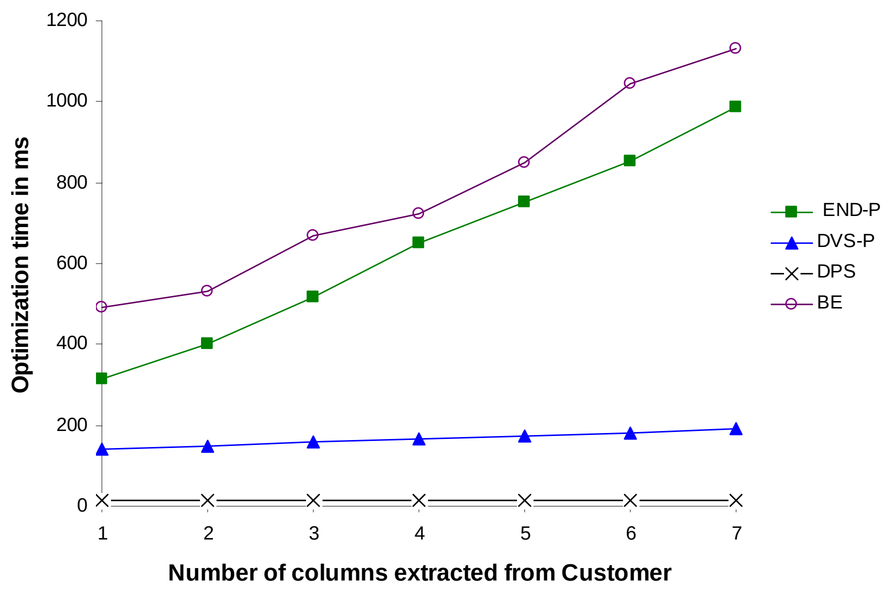


Figure 29: Optimization time for hybrid query

We also evaluated the query execution time for the hybrid query using TPC-H.

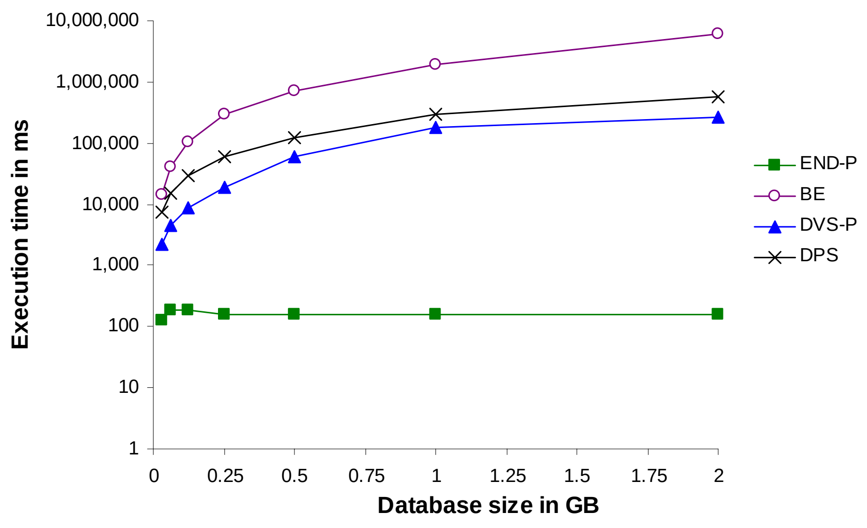


Figure 30: Execution time for hybrid query.

Figure 30 compares the performance of BE, END-P, DPS, and DVS-P for a UPV with 8 mapped properties and an increasing database size.

As expected BE and DPS proved to have the same bad performance as for content query Q1. For hybrid queries DVS-P does not scale because the partial evaluator is not able to uniquely identify the argument to *applyView* required by DVS-P so no view expansion of *applyView* can be made at compile time. DVS-P is faster than DPS with a factor two since partial evaluation by step *PARQ1* avoids selecting entire property views from *ORDERS*.

END-P is the only strategy that shows good performance for both content queries and hybrid queries. To conclude, this experiment shows that partial evaluation with strategy END-P enables transformation of hybrid queries into efficient SQL queries while DVS-P does not scale with the size of the database.

In Figure 31 a summary of all evaluated query processing strategies for hybrid queries is presented. The performance of the different strategies in terms of optimization and execution of hybrid queries are characterized the same way as for content queries. Since strategy END-P shows good performance for all three query classes it is chosen as default when processing SPARQL queries to UPVs in SWARD.

Strategy	**Optimization**	**Execution**	**Compile time candidates *I***
BE	Fast	Non scalable	{}
END	Very slow	Scales	{}
END-P	Fast	Scales	{*cMap,pMap,S*}
DPS	Very fast.	Non scalable	{}
DVS-P	Very fast	Non scalable	{*cMap,pMap,S,pView*}

Figure 31: Evaluation of query processing strategies for hybrid queries

9 Augmented UPVs

So far it has been described how UPVs provide access to content and schema information in relational databases. However, for three reasons the UPVs described so far do not yet qualify as complete RDFS views over relational databases:

- Every ER entity must be associated with an entity type. It should be possible to find all instances of a mapped class or the class of a mapped instance. However, so far, it has not been described how to encode class memberships of mapped instances.
- ER relationship types that model relationships between entity types should be defined in the UPV. ER relationships between mapped classes should be represented in the UPV. This is important for the same reasons as with class memberships. As of yet, it has not been described how to make ER relationships explicit in UPVs.
- So far, it has not been described how to define mapped properties over columns in relational tables with composite primary keys. This must be allowed in order to generate UPVs over arbitrary relational databases.

This Chapter extends the basic UPVs in order to represent the above three kinds of semantic information. Section 9.1 presents how mapped instances are associated with their mapped classes in the UPV. Section 9.2 shows how ER relationships implicitly represented by foreign keys in the relational model are made explicit in the UPV by encoding them as RDF properties relating mapped classes. Section 9.3 shows how relational tables with composite primary keys are represented in a UPV. Finally, in Section 9.4 it is proven that conjunctive content queries to augmented UPVs always are reduced to a simple conjunction by PARQ.

9.1 Class Membership

So far, the content view in a UPV has been defined as the union of property views that define values of mapped properties for mapped instances. However, there is so far nothing in the UPV that states explicitly the class membership of a mapped instance. This means that queries, such as, give me all instances of the class *co:Customer*, must be written as illustrated by query Q4 in Example 24.

```
SELECT ?cust
WHERE {?cust co:CustID ?custid .}
```

Example 24: Content query Q4 finding the instances of a class.

The result of Q4 is:

```
(co:Customer/120)
```

The result is correct since it is here assumed that a mapped property defined over a mapped class is instanciated for every member of that class. However, it is quite unnatural to state the query to find all instances of a class as finding all the mapped instances of some property in *pMap* such that the domain of the property is that particular class.

The RDFS meta-property *rdf:type* defines the class to which a URI belongs. So far we have used it (in Chapter 3) only to define the mapped class and mapped property meta-objects themselves. Thus, the meta-property *rdf:type* has not been used to state the class of a mapped instance as is required for complete RDFS views over relational databases. Because of this one cannot directly find all members of a class by using the *rdf:type* meta-property as in Example 25, and instead one has to use the unnatural query in Example 24. Query triple patterns that retrieve for a given mapped class its instances as in Example 25 are called *class membership patterns*.

```
SELECT ?cust
WHERE {?cust rdf:type co:Customer .}
```

Example 25: Content query Q5, finding all instances of a class.

To incorporate statements that define the classes of mapped instances in the UPV the definition of the content view is augmented to include also a *class membership view,* CM_c, for each mapped class in *cMap* that represents the extent of class *c*.

$$C = \bigcup_p PV_p \; OR \; \bigcup_c CM_c$$

In the example *Company* database the following three RDF triples encode the class membership of the mapped instances in Figure 17:

<co:Customer/120,rdf:type,co:Customer>
<co:Orders/1,rdf:type,co:Orders>
<co:Orders/2,rdf:type,co:Orders>

The augmented UPV definition with class membership views is shown in Example 26.

```
1.U(s,p,v):-
2.S(s,p,v)                  OR
3.P_CustID(s,p,v)           OR
4.P_MktSegment(s,p,v)       OR
5.P_OrderID(s,p,v)          OR
6.P_OCustID(s,p,v)          OR
7.P_Clerk(s,p,v)            OR
8.CM_Customer(s,p,v)        OR
9.CM_Orders(s,p,v)
```

Example 26: UPV definition with class membership view.

The class membership views *CM_Customer* and *CM_Orders* classify the mapped instances from relational tables *CUSTOMER* and *ORDERS* as members of the mapped classes *Customer* and *Orders,* respectively. By convention, here, all class membership views are prefixed with 'CM_'.

Example 27 shows the definition of the class membership view *CM_Customer*.

```
1.CM_Customer(s,p,v):-
2.customer(custid,_)                         AND
3.cMap('CUSTOMER','Comp',v)                  AND
4.iMap(v,custid,s)                           AND
5.p = rdf:type
```

Example 27: Definition of class membership view CM_Customer.

Class membership views associate mapped instances from a relational table with mapped classes and access *cMap* to do so.

In general, a class membership view in a UPV has the structure in Figure 32 where variables *k* is bound to the primary key value of the relational table (*table*). The predicate on line 2 accesses the table to bind *k*.

```
1.CM_[table](s,p,v) :-
2.[table](k)                        AND
3.cMap([table],[upv],v)             AND
4.iMap(v,k,s)                       AND
5.p = rdf:type
```

Figure 32: Definition of class membership view

Even though query Q5 in Example 25 accesses values from the *CUSTID* column in table *CUSTOMER* Q5 is not categorized as a content query by Definition 1 where a content query is defined to be a query where all properties are constants and bound to identifiers for mapped properties. Therefore, a more general definition of content queries is needed, *extended content queries*.

The RDFS meta-property *rdf:type* in a UPV defines either i) the meta-class of a mapped property (class *rdf:Property*) or the meta-class of a mapped class (class *rdfs:Class*) or ii) the class of a mapped instance. A query using *rdf:type* to reference a meta-class (i.e. the first role) is a schema query while a *extended content query* may use *rdf:type* in the second role. Since class membership patterns define the class of only mapped instances we get the following definition:

Definition 4: An *extended content query* is a conjunctive query where all triple patterns are either mapped property patterns, or class membership patterns.

It is now shown that applying PARQ on END with *I = {cMap,pMap,S}* for a extended conjunctive content query substantially reduces the query expression sizes and no normalization is needed for such queries either.

In Example 28 the extended content query Q5 is presented in ObjectLog.

```
1.query(cust):-
2.U(cust,rdf:type,co:Customer)
```

Example 28: ObjectLog expression for Q5.

Example 29 shows how line 2 is view expanded using the UPV definition from Example 26.

```
1.query(cust):-
2.S(cust,rdf:type,co:Customer)            OR
3.P_CustID(cust,rdf:type,co:Customer)     OR
4.P_MktSegment(cust,rdf:type,co:Customer) OR
5.P_OrderID(cust,rdf:type,co:Customer)    OR
6.P_OCustID(cust,rdf:type,co:Customer)    OR
7.P_Clerk(cust,rdf:type,co:Customer)      OR
8.CM_Customer(cust,rdf:type,co:Customer)  OR
9.CM_Orders(cust,rdf:type,co:Customer)
```

Example 29: Query Q5 after view expansion.

By Assumption 1 line 2 is compile time evaluated to *false* and eliminated. Lines 3-7 are also eliminated by partial evaluation because of Assumption 1 since no mapped property identifier can be named *rdf:type*. Example 30 shows the expanded class member view *CM_Orders* from line 9 in Example 29.

```
1.orders(custid,_,_)                  AND
2.iMap(co:Customer,custid,cust)       AND
3.cMap('ORDERS','Comp',co:Customer)   AND
4.rdf:type=rdf:type
```

Example 30: Expanded class membership view CM_Orders.

The call to *cMap* on line 3 is compile time evaluated to *false* and the entire view expanded expression on line 9 in Example 29 is therefore eliminated by PARQ.

Thus, after partial evaluation only line 8 remains in Example 29. After view expansion it becomes the expression in Example 31.

```
1.customer(custid,_)                    AND
2.iMap(co:Customer,custid,cust)         AND
3.cMap('CUSTOMER','Comp',co:Customer)  AND
4.rdf:type=rdf:type
```

Example 31: Expanded class membership view CM_Customer.

The call to *cMap* on line 3 is compile time evaluated to *true* and removed. Line 4 is removed by line 30 in Figure 23. Example 32 shows the fully reduced query Q5.

```
1.query(cust) :-
2.customer(custid,_)             AND
3.iMap(co:CustID,custid,cust)
```

Example 32 Fully reduced query Q5.

Finally, the SQL generator produces the following single SQL statement from line 2 of the reduced query. The statement is sent to the back-end relational DBMS for cost-based optimization and execution.

```
SELECT C.CUSTID
FROM CUSTOMER C
```

The same SQL is generated also from query Q4 that finds all members of a mapped class by accessing all mapped instances with a property that has as domain that particular class.

So far it was shown how to find the extent of a mapped class by queries to UPVs using the property *rdf:type*. The same property *rdf:type* can also be used for finding the type of a given mapped instance.

```
SELECT ?class
WHERE {co:Customer/120 rdf:type ?class .}
```

Example 33: Content query Q6, finding the class membership of an instance.

For example, query Q6 in Example 33 retrieves the class for a given instance of that class, *co:Customer/120*.

The result of Q6 is:

```
(co:Customer)
```

PARQ applied on query Q6 with *I = {cMap,pMap,S,iMap}* will be shown to reduce the query to a simple conjunctive expression before SQL generation.

In Example 34 the extended content query Q6 is presented in ObjectLog.

```
1.query(class):-
2.U(co:Customer/120,rdf:type,class)
```

Example 34: ObjectLog expression for Q6.

Example 35 shows how line 2 is view expanded using the UPV definition from Example 26.

```
1.query(class):-
2.S(co:Customer/120,rdf:type,class)            OR
3.P_CustID(co:Customer/120,rdf:type,class)     OR
4.P_MktSegment(co:Customer/120,rdf:type,class)  OR
5.P_OrderID(co:Customer/120,rdf:type,class)    OR
6.P_OCustID(co:Customer/120,rdf:type,class)    OR
7.P_Clerk(co:Customer/120,rdf:type,class)      OR
8.CM_Customer(co:Customer/120,rdf:type,class)   OR
9.CM_Orders(co:Customer/120,rdf:type,class)
```

Example 35: Query Q6 after view expansion.

Analogously to Example 29 lines 2-7 are eliminated by partial evaluation. The expanded class membership view *CM_Orders* is shown in Example 36.

```
1.orders(custid,_,_)                            AND
2.iMap(cid,custid,co:Customer/120)              AND
3.cMap('ORDERS','Comp',cid)                     AND
4.rdf:type=rdf:type
```

Example 36: Expanded class membership view CM_Orders revisited.

The call to *cMap* on line 3 is compile time evaluated and the class membership view in Example 36 is partial evaluated by binding variable *cid* to the constant mapped class identifier *co:Orders*. Line 4 is removed by line 30 in Figure 23.

The reduced expression is shown in Example 37.

```
1.orders(custid,_,_)                            AND
2.iMap(co:Orders,custid,co:Customer/120)
```

Example 37: Reduced class membership view CM_Orders.

Recall that the external *iMap* predicate is invertible (Chapter 3) to be able to obtain the key for a given mapped instance identifier by parsing the identifier string. In order to eliminate the class membership view *CM_Orders* from the expanded query the external predicate *iMap* is here compile time evaluated in the backward direction to *false*.

Thus, lines 2-9 are replaced with the expression produced from line 8 in Example 35 with the calls to *cMap* and *iMap* compile time evaluated.

```
customer('120',_)
```

After SQL generation the following statement is sent to the back-end relational database for cost-based optimization and execution.

```
SELECT 1
FROM CUSTOMER C
WHERE C.CUSTID = '120'
```

The SQL statement is an existence check for the primary key value '120' in the column *CUSTID* in table *CUSTOMER*.

The external predicate *iMap* is cheap since it does not access the back-end relational but does only do simple string handling.

The possibility to define inverses of an external predicates is critical for the query processing performance in SWARD. Consider a scenario when *iMap* was not invertible. Instead of the SQL statement above, the code in Example 37 would remain and the following two statements would be sent to the back-end DBMS for execution.

```
SELECT C.CUSTID
FROM CUSTOMER C

SELECT O.ORDERID
FROM ORDERS O
```

Since here the key for the given mapped instance in query Q6 cannot be obtained by compile time evaluation the whole table *CUSTOMER* in the back-end relational DBMS is scanned. Also, the code in Example 37 would remain, since the class identifier for the mapped instance in query Q6 cannot be accessed, which produces an additional scan of the *ORDERS* table. This is clearly a non-scalable strategy for evaluating query Q6.

With the introduction of class membership properties in UPVs queries could be stated that dynamically selects mapped classes and accesses their members. Such a subclass of hybrid queries is not investigated in this Thesis.

In summary, in this Section it was shown by examples how ER entity information was preserved in UPVs through the class membership views. It was also shown how class membership views where eliminated from the extended content queries (Definition 4) using partial evaluation in the same manor as for ordinary content queries.

Next, it is shown how to further augment UPVs to encode relationships between mapped classes using foreign key information in the relational database. This is required in order to explicitly model relationships among mapped classes, which in turn enables more natural queries to the UPV.

9.2 Class Relationships

Binary ER relationships are supported implicitly in relational databases through foreign keys. Such relationships are modeled in UPVs as special mapped properties that relate mapped instances from two mapped classes.

For example, in the relational database, *Company*, the relationship between entity types *ORDERS* and *CUSTOMER* in Figure 13 is represented by the fact that column *OCUSTID* is a foreign key in table *ORDERS* that references column *CUSTID* in table *CUSTOMER*.

In the basic UPV, this is encoded implicitly by the three RDF triples:

<*co:Orders/1,co:OrderCustomer,120*>
<*co:Orders/2,co:OrderCustomer,120*>
<*co:Customer/120,co:CustID,120*>

To get the market segment of a customer placing an order and the clerk filing the order, one has to join the values of the foreign keys on lines 3 and 4 in Example 38.

```
1.SELECT ?cust ?mkt ?clerk
2.WHERE {?order co:Clerk ?clerk .
3.       ?order co:OrderCust ?ocust .
4.       ?cust co:CustID ?ocust .
5.       ?cust co:Market ?mkt .}
```

Example 38: Content query Q7

By including these additional two triples in the UPV the class relationship is made explicit through a new property *co:OrderedBy*.

<co:Orders/1,co:OrderedBy,co:Customer/120>
<co:Orders/2,co:OrderedBy,co:Customer/120>

Here, *co:Orders/1* is a mapped instance of the class *co:Orders* and *co:Customer/120* is a mapped instance of the class *co:Customer*. Properties that relate instances from the two mapped classes, such as *co:OrderedBy,* are called *class relationship properties*. Triple patterns where the property is constant and identifies a class relationship property are called *class relationship patterns*. Example 39 shows a SPARQL query with a class relationship pattern. The definition of a extended content query is genralized even further to incorporate also class relationship patterns.

Definition 5: A *generalized content query* is a conjunctive query where all triple patterns are either mapped property patterns, class membership patterns, or class relationship patterns.

```
SELECT ?cust ?mkt ?clerk
WHERE {?order co:Clerk ?clerk .
       ?order co:OrderedBy ?cust .
       ?cust co:Market ?mkt .}
```

Example 39: Generalized content query Q7

Using class relationship patterns, retrieving the market segment of a customer placing an order can be expressed in a more natural and simple way as shown in Example 39.

Table	**OTable**	**FkColumn**	**UPV**	**CRID**
ORDERS	CUSTOMER	ORDERID	Comp	co:OrderedBy

Figure 33: Relationship mapping table for Company

To represent mapped class relationships in the UPV SWARD requires a user-defined *relationship mapping table, rMap(Table,OTable,FkColumn,UPV,CRID)* (Figure 33), that maps 1:1 for a given UPV between a foreign key column (*FkColumn*) in a table (*Table*) that refers to the primary key column in some other table (*OTable*) into a class relationship identifier (*CRID*).

SWARD handles class relationships by generating additional views, class *relationship views* CR_{crid}, for the class relationships identifiers in *rMap*. The content view definition is generalized to the union of property views, class membership views, and class relationship views:

$$C = \bigcup_{p} PV_p \; OR \bigcup_{c} CM_c \; OR \bigcup_{crid} CR_{crid}$$

The further augmented UPV definition with class relationship views is shown in Example 40. By convention, here, all class relationship view names are prefixed with ‘CR_’.

```
1.U(s,p,v):-
2.S(s,p,v)                  OR
3.P_CustID(s,p,v)           OR
4.P_MktSegment(s,p,v)       OR
5.P_OrderID(s,p,v)          OR
6.P_OCustID(s,p,v)          OR
7.P_Clerk(s,p,v)            OR
8.CM_Customer(s,p,v)        OR
9.CM_Orders(s,p,v)          OR
10.CR_OcustID(s,p,v)
```

Example 40: UPV definition with class relationship view

Example 41 shows the class relationship view definition for the class relationship identifier *co:OrderedBy* in Figure 33 (line 10 in Example 40).

```
1.CR_OCustID(s,p,v) :-
2.orders(orderid,ocustid,_)                             AND
3.iMap(cid,orderid,s)                                   AND
4.cMap(‘CUSTOMER’,’Comp’,otcid)                         AND
5.iMap(otcid,ocustid,v)                                 AND
6.cMap(‘ORDERS’,‘Comp’,cid)                             AND
7.rMap(‘ORDERS’,’CUSTOMER’,‘OCUSTID’,‘Comp’,p)
```

Example 41: Class relationship view CR_OcustID

Notice how additional calls to *cMap* and *iMap* on lines 4 and 5, respectively, are needed to construct the mapped instance *co:Customer/120*. On line 7 *rMap* is accessed to get the class relationship identifier.

In general, a class relationship view in a UPV has the structure in Figure 34 where variable *k* is bound to values from the primary key column of the relational table (*table*) and *fv* is bound to values from the foreign key column (*column*) of the table mapped by the class relationship identifier *p* in *rMap*, referencing the relational table (*otable*).

```
1.CR_[column](s,p,v) :-
2.[table](k,fv)                          AND
3.iMap(cid,k,s)                          AND
3.cMap([otable],[upv],otcid)             AND
4.iMap(otcid,fv,v)                       AND
6.cMap([table],[upv],cid)                AND
7.rMap([table],[otable],[column],[upv],p)
```

Figure 34: Class relationship view definition

In addition, the class view (Example 6), domain view, and range view, which define the schema view *S*, are extended to define the class relationship property *co:OrderedBy* along with its domain and range through the triples below:

<co:OrderedBy, rdf:type,rdfs:Property>
<co:OrderedBy, rdfs:domain,co:Orders>
<co:OrderedBy,rdfs:range,co:Customer>

The extended class, domain and range view are presented in Example 42.

```
Classes(s,p,v):- (cMap(table,‘Comp’,s)              AND
                  p = rdf:type                      AND
                  v = rdfs:Class)                        OR
                 (pMap(table,column,‘Comp’,s)       AND
                  p = rdf:type                      AND
                  v = rdf:Property)                      OR
                 (rMap(table,otable,column,‘Comp’,s)AND
                  p = rdf:type                      AND
                  v = rdf:Property)

Domains(s,p,v):- (pMap(table,column,’Comp’,s)       AND
                  p=rdfs:domain                     AND
                  cMap(table,’Comp’,v))                  OR
                 (rMap(table,otable,column,’Comp’,s)AND
                  p=rdfs:domain                     AND
                  cMap(table,’Comp’,v))

Ranges(s,p,v):- (pMap(table,column,’Comp’,s)        AND
                 p = rdfs:range                     AND
                 v = rdfs:Literal)                       OR
                (rMap(table,otable,column,’Comp’,s) AND
                 p = rdfs:range                     AND
                 cmap(otable,’Comp’,v))
```

Example 42: Extended schema view definition

Now the market segment of a customer placing an order can be selected in a more natural and simple way by using the class relationship identifier *co:OrderedBy* as shown in Example 43.

```
SELECT ?cust ?mkt ?clerk
WHERE {?order co:Clerk ?clerk .
       ?order co:OrderedBy ?cust .
       ?cust co:Market ?mkt .}
```

Example 43: Content query Q8

Query Q8 produces the result:

```
(co:Customer/120,‘AUTOMOBLE’,’Doe’)
(co:Customer/120,‘AUTOMOBILE’,’Wesson’)
```

Next it is shown how applying PARQ on END with *I = {cMap,pMap,S,rMap}* the size of UPVs with class relationship views are substantially reduced and no normalization is needed for conjunctive content queries. Analogous to *cMap*, *pMap* and *S* the primitive predicate *rMap* is also small and stored in main memory of SWARD. It does not access the back-end relational database.

Example 44 shows query Q7 in ObjectLog.

```
1.query(cust,mkt,clerk) :-
2.U(order,co:Clerk,clerk)           AND
3.U(order,co:OrderedBy,cust)        AND
4.U(cust,co:Market,mkt)
```

Example 44: ObjectLog expression for Q8

Example 45 shows how line 2 in Example 44 is view expanded using the UPV definition augmented with class membership views and relationship views.

```
1.query(cust,mkt,clerk) :-
2.(S(order,co:Clerk,clerk)            OR
3. P_CustID(order,co:Clerk,clerk)     OR
4. P_MktSegment(order,co:Clerk,clerk) OR
5. P_OrderID(order,co:Clerk,clerk)    OR
6. P_OCustID(order,co:Clerk,clerk)    OR
7. P_Clerk(order,co:Clerk,clerk)      OR
8. CM_Orders(order,co:Clerk,clerk)    OR
9. CM_Customer(order,co:Clerk,clerk)  OR
10.CR_OCustID(order,co:Clerk,clerk))         AND
11.U(order,co:OrderedBy,cust)                AND
12.U(cust,co:Market,mkt)
```

Example 45: Query Q8 after view expansion

Notice how the UPV definition is augmented with the relationship view *CR_OCustID* on line 10 and class membership views *CM_Orders* and *CM_Customer* on lines 8 and 9, respectively. Example 46 shows the expanded relationship view *CR_OCustID*.

```
1.orders(orderid,ocustid,_)                        AND
2.iMap(cid,orderid,order)                          AND
3.cMap(‘CUSTOMER’,’Comp’,otcid)                    AND
4.iMap(otcid,ocustid,clerk)                        AND
5.cMap(‘ORDERS’,’Comp’,cid)                        AND
6.rMap(‘ORDERS’,’CUSTOMER’,’OCUSTID’,’Comp’,co:Clerk)
```

Example 46: Expanded relationship view CR_OCustID.

In the example, compile time evaluation of *rMap* (in step *PARQ1*) on line 6 is evaluated to *false* and the entire view expanded expression on line 10 in Example 45 is eliminated.

Lines 2-6 and 8-9 in Example 45 are also partial evaluated to *false* and removed reducing the disjunction to the conjunctive query fragment representing the property view *P_Clerk* shown in Example 47.

```
1.orders(orderid,_,clerk)                       AND
2.iMap(cid,orderid,order)                       AND
3.cMap('ORDERS','Comp',cid)                     AND
4.pMap('ORDERS','CLERK','Comp',co:Clerk)
```

Example 47: Expanded property view P_Clerk.

The call to *pMap* on line 4 is compile time evaluated to *true* and removed. On line 3 compile time evaluation of *cMap* partial evaluates the property view *P_Clerk* in Example 47 by substituting variable *cid* for *co:Orders*.

Thus partial evaluation replaces the disjunctive expression in the expanded *U* on line 2 in Example 44 with the following expression representing the desired property identifier (*co:Clerk*):

```
orders(orderid,_,clerk)              AND
iMap(co:Orders,orderid,order)
```

Analogously, line 4 in Example 44 is replaced with the conjunctive expression representing the property identifier *co:Market*:

```
customer(custid,mkt)
iMap(co:Customer,custid,cust)
```

The UPV call on line 3 in Example 44 is view expanded producing the disjunction in Example 48.

```
1.S(order,co:OrderedBy,cust)            OR
2.P_CustID(order,co:OrderedBy,cust)     OR
3.P_MktSegment(order,co:OrderedBy,cust) OR
4.P_OrderID(order,co:OrderedBy,cust)    OR
5.P_OCustID(order,co:OrderedBy,cust)    OR
6.P_Clerk(order,co:OrderedBy,cust)      OR
7.CM_Orders(order,co:OrderedBy,cust)    OR
8.CM_Customer(order,co:OrderedBy,cust)  OR
9.CR_OCustID(order,co:OrderedBy,cust)   OR
```

Example 48: Expanded UPV call

Lines 1-6 are eliminated by partial evaluation.

Example 49 shows the expanded class membership view *CM_Customer*.

```
1.customer(custid,_)                    AND
2.cMap(‘CUSTOMER’,’Comp’,cust)          AND
3.iMap(cust,custid,order)               AND
4.co:OrderedBy=rdf:type
```

Example 49: Expanded class membership view CM_Customer

In the example, the equality on line 4 is replaced with *false* (line 32 in Figure 23) and the entire view expanded expression on line 8 in Example 48 is eliminated. Analogously line 7 in Example 48 is also eliminated by partial evaluation reducing the disjunction the conjunctive query fragment representing the property view *CR_OCustID* shown in Example 50.

```
1.orders(orderid,ocustid,_)                                 AND
2.iMap(cid,orderid,order)                                   AND
3.cMap(‘CUSTOMER’,’Comp’,otcid)                             AND
4.iMap(otcid,ocustid,cust)                                  AND
5.cMap(‘ORDERS’,’Comp’,cid)                                 AND
6.rMap(‘ORDERS’,’CUSTOMER’,’OCUSTID’,’Comp’,co:OrderedBy)
```

Example 50: Expanded relationship view CR_OCustID.

The call to *rMap* on line 6 is compile time evaluated to *true* and removed. On lines 3 and 5 evaluation of *cMap* substitutes variables *otcid* and *cid* for property identifiers *co:Customer* and *co:Orders,* respectively

Thus partial evaluation replaces the disjunctive expression in the expanded *U* on line 3 in Example 44 with the following expression representing the desired property identifier (*co:OrderedBy*):

```
orders(orderid,ocustid,_)               AND
iMap(co:Orders,orderid,order)           AND
iMap(co:Customer,ocustid,cust)
```

The key rewrite rule (Figure 23) combines the produced *iMap* and relational database calls to the same table before generating the SQL. It produces the fully reduced query as shown in Example 51:

```
1.query(cust,mkt,clerk) :-
2.orders(orderid,ocustid,clerk)           AND
3.iMap(co:Orders,orderid,order)           AND
4.customer(ocustid,mkt)                   AND
5.iMap(co:Customer,ocustid,cust)
```

Example 51: Fully reduced query Q6.

Finally, the SQL generator produces the following single SQL statement from the reduced query. The statement is sent to the back-end relational DBMS for cost-based optimization and execution:

```
SELECT O.ORDERID,O.OCUSTID,O.CLERK,C.MKTSEGMENT
FROM ORDERS O,CUSTOMER C
WHERE C.CUSTID = O.OCUSTID
```

In this section ER relationships, implicitly represented by foreign keys in the relational model, where defined explicitly by augmenting the UPV definition with class relationship views. It was also exemplified how relationship views where eliminated from the generalized content query (Definition 5) using partial evaluation in the same manor as ordinary property views.

Handling of binary M:N ER relationship types and N-ary ER relationship types is not investigated in this Thesis.

9.3 Representation of Composite Keys in RDF Schema

Since composite primary keys are common in relational databases it is very important that they are handled in UPVs so that columns in tables containing such keys can also be viewed in RDF.

For example, consider the addition of the weak entity type *LINEITEM* to the ER diagram in Figure 13 as shown in Figure 35.

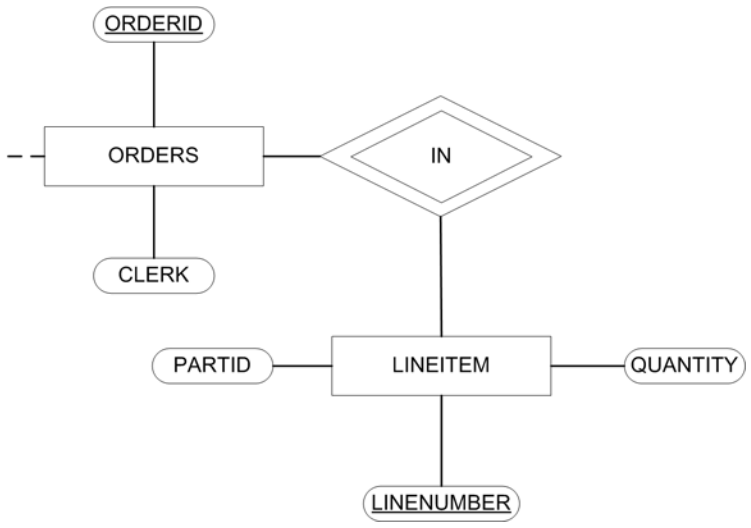


Figure 35: ER diagram of extended Company database.

The weak entity type *LINEITEM* is translated to the additional *LINEITEM* table in our example *Company* database, providing information about the quantity of parts from a product line in an order placed by a customer. The columns *LINENUMBER* and *LORDERID* are the composite primary key in *LINEITEM*.

LINEITEM	LINENUMBER	LORDERID	PARTID	QUANTITY
	12345	2	Semiconductor	150

In a UPV identifiers are constructed to represent mapped instances of classes in *cMap*. An instance identifier consists of the identifier representing the class in *cMap* mapped to the table concatenated with a key value from its primary key (Section 3.3) and corresponds to a row in that table. For a table with a composite primary key the instance identifier must contain key values from *all* the primary key columns. Such instance identifiers are called *composite instance identifiers*.

Figure 36 shows the class mapping for table *LINEITEM*.

Table	**UPV**	**ClassID**
LINEITEM	Comp	co:LineItem

Figure 36: Additional class mappings for LINEITEM table

Figure 37 shows the property mappings in *pMap* for the columns in table *LINEITEM*.

Table	**Column**	**UPV**	**PropID**
LINEITEM	LINENUMBER	Comp	co:LineNumber
LINEITEM	LORDERID	Comp	co:LineNumberOrderID
LINEITEM	PARTID	Comp	co:PartID
LINEITEM	QUANTITY	Comp	co:Amount

Figure 37: Additional property mappings for LINEITEM table

Since column *LORDERID* in table *LINEITEM* is a foreign key referencing column *ORDERID* in table *ORDERS* the relationship mapping table *rMap* contains the mapping shown in Figure 38.

Table	**OTable**	**Column**	**UPV**	**CRID**
LINEITEM	ORDERS	LORDERID	Comp	co:BelongsTo

Figure 38: Additional relationship mappings for LINEITEM table

Given the above class relationship and property mapping tables the content view of the UPV *Comp* over the extended *Company* database will contain the additional triples shown in Figure 39. The composite instance identifiers are constructed by concatenating the primary key values in a row. We use the separator ‘@’ when concatenating the primary key values in composite instance identifiers to make it possible to inversely reconstruct the keys from the instance identifier.

S	P	V
co:LineItem/12345@2	co:LineNumber	12345
co:LineItem/12345@2	co:LineNumberOrderID	2
co:LineItem/12345@2	co:PartID	Semiconductor
co:LineItem/12345@2	co:Amount	150
co:LineItem/12345@2	co:BelongsTo	co:Orders/2
co:LineItem/12345@2	rdf:type	co:LineItem

Figure 39: Content view for LINEITEM table

Example 52 shows the UPV definition generated by SWARD for the extended *Company* database where *P*, *CR,* and *CM* on lines 3-5 are the unions of property, class membership, and class relationship views, respectively. Lines 6-9 show the property views over composite primary key table *LINEITEM*.

```
1.U(s,p,v):-
2.S(s,p,v)                OR
3.P(s,p,v)                OR
4.CR(s,p,v)               OR
5.CM(s,p,v)               OR
6.P_LineNumID(s,p,v)      OR
7.P_LOrderID(s,p,v)       OR
8.P_PartID(s,p,v)         OR
9.P_Quantity(s,p,v)
```

Example 52: UPV definition over extended Company database with support for property views over columns in tables with composite primary keys.

Example 53 shows the property view *P_Quantity* representing column *QUANTITY* in table *LINEITEM*.

```
1.P_Quantity(s,p,v) :-
2.lineitem(lorderid,linenumber,lpartid,v)          AND
3.cMap('LINEITEM','Comp',cid)                      AND
4.pMap('LINEITEM','QUANTITY','Comp',p)             AND
5.KC_LineItem(linenumber,lorderid,ck)              AND
6.iMap(cid,ck,s)
```

Example 53: Property view P_Quantity

In order to concatenate composite primary key values to construct composite mapped instance identifiers, the general definition of a property view in Figure 21 is augmented with a predicate generated by SWARD for each table associated with a class in *cMap*, the *key constructor,* prefixed with 'KC_', *KC_LineItem* in Example 53. A key constructor takes as its first arguments the primary keys values of a row and returns the concatenated key string of a composite instance identifier as the last argument. For example, on line 5 in Example 53 the key constructor *KC_LineItem* takes as input values from the composite primary key columns *LINENUMBER* and *LORDERID* and concatenates them into a key string. If *LINENUMBER* is '12345' and *LORDERID* is '2' the concatenated key string becomes '12345@2'. With the addition of the key constructor, property views are generalized to view columns in relational tables with composite primary keys.

In general, a generalized property view in a UPV able to represent composite primary keys has the structure in Figure 34 where variables $k_1,\ldots,k_n$ are bound to key values in a composite primary key of the relational table (*table*) and *v* is a row value from the relational column (*column*) represented by the mapped property identifier *p* in *pMap*.

```
P_[column](s,p,v) :-
[table](k_1,…,k_p,v)                AND
cMap([table],[upv],cid)             AND
pMap([table],[column],[upv],p)      AND
KC_[table](k_1,…,k_p,ck)            AND
iMap(cid,ck,s)
```

Figure 40: Composite property view definition

The key constructors are implemented as external predicates with variable number of arguments to be able to construct a compound instance identifier from several key values. The predicate is invertible to be able to obtain the keys for a given composite mapped instance identifier by parsing the identifier string.

Templates for generalized class membership views and relationship views able to handle composite primary keys are shown in Figure 41 and Figure 42, respectively.

```
CM_[table](s,p,v) :-
[table](k_1,…,k_n)                  AND
cMap([table],[upv],v)               AND
KC_[table](k_1,…,k_n,ck)            AND
iMap(v,ck,s)                        AND
p = rdf:type
```

Figure 41: Generalized definition of class membership view

Variables $k_1,\ldots,k_n$ are bound to key values in a composite primary key of the relational table (*table*).

```
1.CR_[column](s,p,v) :-
2.[table]_[column](k_1,…,k_n,v)          AND
3.iMap(cid,ck,s)                          AND
4.cMap([otable],[upv],otcid)              AND
5.iMap(otcid,v,v)                         AND
6.cMap([table],[upv],cid)                 AND
7.KC_[table](k_1,…,k_n,ck)                AND
8.rMap([table],[otable],[column],[upv],p)
```

Figure 42:Generalized definition of class relationship view

Notice how calls to the key constructors are added on lines 4 and 7 in Figure 41 and Figure 42, respectively, to construct compound instance identifiers from key values.

It is now shown by an example that by applying PARQ on END with *I = {cMap,pMap,S,rMap}* the size of UPVs able to handle composite primary keys are substantially reduced without need for normalization for conjunctive content queries.

Example 54 shows the content query Q9 that returns the quantity of every part in each order and the name of the clerk handling that order.

```
SELECT ?part ?qty ?clerk
WHERE {?lineitem co:PartID ?part .
       ?lineitem co:Amount ?qty .
       ?lineitem co:BelongsTo ?order .
       ?order co:Clerk ?clerk .}
```

Example 54: Content query Q9

Query Q9 produces the result:

```
('Semiconductors','150','Doe')
```

Example 55 shows query Q9 in ObjectLog.

```
1.query(part,qty,clerk) :-
2.U(lineitem,co:PartID,part)              AND
3.U(lineitem,co:Amount,qty)               AND
4.U(lineitem,co:BelongsTo,order)          AND
5.U(order,co:Clerk,clerk)
```

Example 55: ObjectLog expression for Q9

Example 56 shows how line 3 in Example 55 is view expanded using the UPV definition in Example 52.

```
U(lineitem,co:Amount,qty):-
S(lineitem,co:Amount,qty)                    OR
P(lineitem,co:Amount,qty)                    OR
CR(lineitem,co:Amount,qty)                   OR
CM(lineitem,co:Amount,qty)                   OR
P_LineNumID(lineitem,co:Amount,qty)          OR
P_LOrderID(lineitem,co:Amount,qty)           OR
P_PartID(lineitem,co:Amount,qty)             OR
P_Quantity(lineitem,co:Amount,qty)
```

Example 56: UPV definition over extended Company with generalized property views

Example 57 shows the view expanded composite property view *P_Quantity*.

```
1.lineitem(linenumber,lorderid,_,qty)           AND
2.cMap('LINEITEM','Comp',cid)                   AND
3.pMap('LINEITEM','QUANTITY','Comp',co:Amount)  AND
4.KC_LineItem(linenumber,lorderid,ck)           AND
5.iMap(cid,ck,lineitem)
```

Example 57: View expanded generalized property view P_Quantity

On line 2 the call to *cMap* is compile time evaluated and *cid* is substituted for *co:LineItem*. The call to *pMap* on line 3 is evaluated to *true* and is eliminated. The reduced query fragment is shown in Example 58.

```
1.lineitem(linenumber,lorderid,_,qty)    AND
2.KC_LineItem(linenumber,lorderid,ck)    AND
3.iMap(co:LineItem,ck,lineitem)
```

Example 58: View expanded and partial evaluated generalized property view P_Quantity

The following SQL query is generated (lines 2-5 in Example 55) and sent to the back-end relational DBMS for cost-based optimization and execution.

```
SELECT .LORDERID,L.LPARTID,L.LINENUMBER,L.QUANTITY,O.CLERK
FROM LINEITEM L,ORDERS O
WHERE O.ORDERID = L.LORDERID
```

Calls to the external predicates *KC_LineItem* and *iMap* are post-processed in SWARD to construct the result of the query.

Handling of relational composite *foreign* keys in UPVs is not investigated in this Thesis.

9.4 Proving Reduction of Generalized Content Queries

In the previous sections it was shown how generalized content queries to augmented UPVs were reduced to simple conjunctions. A generalized content query is a query where all triple patterns can be (Definition 5) i) a mapped property pattern, ii) a class membership pattern, or iii) a class relationship pattern. It will now be proven that generalized content queries always are reduced to a simple conjunction by PARQ.

To prove this the following definition is needed:

Assumption 2: A property can never identify both a mapped property and a class relationship property.

Assumption 2 is natural because no row identifiers are stored in the back-end relational database.

Example 59 shows a content query, Q10, to the augmented UPV *U* of the *Company* relational database, containing all three types of triple patterns. The query fetches, for every order, the parts that constitute the order. Notice that the class membership pattern on line 2 could be omitted here but is retained since it is necessary for general applicability of the discussion.

```
1.query(order,part) :-
2.U(lineitem,rdf:type,co:LineItem)         AND
3.U(lineitem,co:PartID,part)               AND
4.U(lineitem,co:BelongsTo,order)
```

Example 59: ObjectLog expression for Q10

In order to prove that generalized content queries to augmented UPVs by application of PARQ always are reduced to simple conjunctions, it is sufficient to show that each type of triple pattern referenced in the query is reduced to a simple conjunction.

Recall from Section 9.2 that an augmented UPV is defined as the union of a schema view, property views, class membership views, and class relationship views.

9.4.1 Mapped Property Patterns

Consider the mapped property pattern on line 3 in Example 59. In general such a pattern has the form:

U(s,p,v)

Variable *p* is a constant and bound to an identifier for a mapped property in the UPV and *s* and *v* are variables representing the subject and value of the triple, respectively.

Example 60 shows the view expanded line 3 from Example 59.

```
1.U(lineitem,co:PartID,part) :-
2.S(lineitem,co:PartID,part)                      OR
3.P(lineitem,co:PartID,part)                      OR
4.CR(lineitem,co:PartID,part)                     OR
5.CM(lineitem,co:PartID,part)
```

Example 60: View expanded property pattern UPV reference

For the union of property views (line 3) and the schema view (line 2) it has already been shown that only one property view remains after view expansion and partial evaluation (Corollary in Chapter 6). What remains for mapped property patterns is to prove that all expanded class membership views (line 5) and class relationship views (line 4) are eliminated during partial evaluation.

Class membership views are eliminated because an identifier can never represent both a mapped property and a schema property (Assumption 1). In general, compile time evaluation of the equality predicate in line 5 in Figure 32 will eliminate all class membership views from a view expanded property pattern UPV reference. In Example 60 line 5 is eliminated.

Class relationship views are eliminated because an identifier can never represent both a mapped property and a class relationship property (Assumption 2). In general, compile time evaluation of *rMap* in line 7 in Figure 34 will eliminate all class relationship views from the view expanded property pattern UPV reference. In Example 60 line 4 is eliminated.

Thus, with *I* = *{pMap,S,rMap}*, mapped property patterns are always reduced to a simple conjunction after view expansion and partial evaluation.

9.4.2 Class Membership Patterns

Consider the class membership pattern on line 2 in Example 59. Such a pattern has the general form:

U(s,membership,v)

Variable *membership* is constant and bound to the schema property identifier *rdf:type* and either *s* is constant and bound to an identifier for a mapped instance identifier or *v* is constant and bound to an identifier for a mapped class.

First, assume that *v* is bound to a mapped class identifier. Example 61 shows the view expanded line 2 from Example 59.

```
1.U(lineitem,rdf:type,co:LineItem) :-
1.S(lineitem,rdf:type,co:LineItem)                      OR
2.P(lineitem,rdf:type,co:LineItem)                      OR
3.CR(lineitem,rdf:type,co:LineItem)                     OR
4.CM(lineitem,rdf:type,co:LineItem)
```

Example 61: View expanded class membership pattern UPV reference

Schema views are eliminated because of Assumption 1 and line 2 in Example 61 is removed.

Property views are eliminated because of Assumption 1. In general, compile time evaluating the *pMap* predicate on line 5 in Figure 21 will eliminate all property views from a view expanded class membership pattern UPV reference. For example, in Example 61 line 3 is removed.

Class relationship views are eliminated because of Assumption 1. In general, compile time evaluating the *rMap* predicate on line 7 in Figure 34 will eliminate all class relationship views from a view expanded class membership pattern UPV reference. For example, in Example 61 line 4 is removed.

In general, for a UPV with more than one class membership view the inverse call to *iMap* on line 4 in Figure 32 will evaluate to *false* in every class membership view except for the class membership producing mapped instances from the searched mapped class.

Now assume that *v* is unbound and *s* is bound to a mapped instance identifier. Analogously to the case when *v* was bound to a mapped class identifier the schema, property, and class relationship views are eliminated.

For UPVs with more than one class membership view the call to *cMap* on line 3 in Figure 32 will evaluate to *false* in every class membership view except for the class membership producing mapped instances from the mapped class identified by *v*.

Thus, with *I = {cMap,pMap,S,rMap,iMap}*, class membership patterns are always reduced to a simple conjunction after view expansion and partial evaluation.

9.4.3 Class Relationship Patterns

Consider the class relationship pattern on line 4 in Example 59. Such a pattern has the general form:

U(s,relationship,v)

The variable *relationship* is a constant bound to a class relationship identifier in the UPV.

Example 62 shows the view expanded line 4 from Example 59.

```
1.U(lineitem,co:BelongsTo,order) :-
1.S(lineitem,co:BelongsTo,order)                    OR
2.P(lineitem,co:BelongsTo,order)                    OR
3.CR(lineitem,co:BelongsTo,order)                   OR
4.CM(lineitem,co:BelongsTo,order)
```

Example 62: View expanded class relationship UPV reference

Schema views are eliminated because of Assumption 1. In Example 62 line 2 is removed.

Property views are eliminated because of Assumption 2. In general, compile time evaluation of *pMap* on line5 in Figure 21 will eliminate all property views in the view expanded triple pattern. In Example 62 line 3 is removed.

Class membership views are eliminated because of Assumption 1. In general, compile time evaluation of the equality predicate in line 5 in Figure 32 will eliminate all class membership views from the view expanded triple pattern. In Example 62, line 5 is removed.

In general, for a given UPV with more than one class relationship view the call to *rMap* on line 7 in Figure 34 will evaluate to *false* in all class relationship views except for the single one represented by the *relationship* parameter.

Thus, with *I = {pMap,S,rMap}*, class relationship patterns are always reduced to a simple conjunction after view expansion and partial evaluation.

Since all three types of patterns are shown to be reduced to simple conjunctions it can be concluded that content queries to augmented UPVs, with *I = {cMap,pMap,S,rMap,iMap}*, are always are reduced to simple conjunctions by application of the partial evaluation algorithm PARQ.

10 Related Work

This Chapter presents an overview of research projects and techniques related to this Thesis.

10.1 RDF Repository Systems

RDF repository systems [4][12][15][67] are systems for storing and searching large volumes of RDF statements. Most RDF repository systems [12][15][67] use relational databases internally. Data in the repository can be either directly stored RDF statements or statements downloaded and converted to RDF from some relational database. The internal database is fully managed by the repository system.

For RDF repositories with a single table storing all the triples such as in [15][67], in general, every two path expressions in the semantic web query will be translated into a triple table join in SQL. This means that SPARQL queries over such relational RDF repositories can be very slow to execute since when the number of triples in the triple table is increased the table may not fit in main memory any more meaning that each triple table join in the SPARQL query requires several disk accesses.

Another problem is that it is hard to access proper information about the distribution of values for different properties in an RDF repository with only one table storing all the triples. Not having enough information about the data will prevent the cost-based optimizer of doing a good job during query processing.

Because of no information about the characteristics of data needed by the applications it is difficult for the administrator of the repository to know how to cluster data and which indexes to create since. Usually indexes are defined on all three columns of the triple table.

To improve performance the RDF repositories [15][67] use property tables to cluster properties often accessed together. However, the generated SQL queries to search these property tables become complex when data has to be combined from several tables. Another problem is the many NULL values in property tables because of the unstructured nature of RDF data.

A better alternative is to use column databases such as e.g. [1][60][38] as back-ends for RDF repositories. This way the size of the triple store can be kept smaller when data is highly unstructured. The column-based approach will require more SQL joins compared to the property-table based one but efficient join algorithms can be used for this [1].

The AllegroGraph RDF repository [4] uses a native object store for storage of RDF.

Rather than storing RDF data in dedicated RDF-repositories SWARD automatically generates general UPV views over any relational database, given information how to perform the mappings between RDFS classes and properties and relational database tables and columns, respectively. By keeping the data in the relational database SWARD utilizes that relational databases are optimized for handling very large data volumes. Furthermore, by defining RDF views over relational databases they are queryable without having to be copied to some RDF repository and the UPV will always reflect any changes in the relational database.

10.2 RDF View Systems

SWIM [16] provides RDF views over relational databases. In SWIM, RQL [16] queries are internally represented as Datalog programs similar to SWARD. Partial evaluation is there also proposed as a way to reduce the size of RQL queries but we are not aware of any results or application of the technique to relational databases. RQL queries in SWIM are minimized using the *Chase* and *BackChase* algorithm [56][55] from [21] where a query is chased to a universal query plan, which is then minimized.

In SWARD this corresponds to the view expansion of UPV references in the query producing an analog to the universal query plan in [21], followed by reduction of the query by partial evaluation. However, the work in [56] is theoretical and we are not aware of any implementation thereof. Furthermore, the proposed *BackChase* algorithm is NP-complete [56] while PARQ reduces the query fast.

D2R Server [10] also provides RDF views over relational databases. In D2R Server the user explicitly specifies SQL fragments to fetch values of RDF properties from the underlying relational database, using user defined mappings between RDFS ontology elements and SQL fragments in the D2RQ mapping language [9]. D2R Server then combines the SQL fragments into complete queries. The user is responsible for the specialization of queries in the D2R Server i.e. the user must write a new specialized translator for each RDFS description to be used to map the relational database to RDF. Optionally, the D2R Server can automatically produce the D2RQ mappings needed according to some default scheme provided by the system. However, such automatic generation of mappings is inflexible and prevents the user from specifying mappings needed by a particular application.

By contrast, SWARD provides a general representation of RDF views over relational databases, UPVs, given two very simple user provided mapping tables. The user does not have to spend time learning a new language (D2RQ) and manually specializing the system, which is bound to introduce errors, but can instead focus on defining the proper mappings between RDFS elements and relational database constructs on a high level. Queries to the generated UPV are automatically specialized i.e. substantially reduced and dynamically translated to SQL during query processing using partial evaluation. Furthermore, we are not aware of any scalability experiments conducted with D2R Server.

SquirrelRDF [59] is a tool that enables relational databases to be queried in SPARQL through RDF views. This is done by translating SPARQL queries, given some user defined mappings between RDFS elements and relational database constructs, to SQL statements sent to the underlying relational database for execution.

However, in contrast to SWARD, SquirrelRDF does not support querying the RDFS ontology data such as for example the classes or properties in the relational database. Also, no information about relationships among RDFS classes is revealed. As opposed to in SWARD, domains and ranges for properties have to be manually added by the administrator. Furthermore, we are not aware of any scalability experiments conducted or any optimization at all being done during query processing in SquirrelRDF.

10.3 Partial Evaluation

Application of partial evaluation can be found in several areas such as for example automatic compiler generation [6][7], operating systems [46], programming languages [6][7][33][53] and computer graphics [5].

In this Thesis partial evaluation is applied on database queries by the PARQ algorithm. For database queries partial evaluation has been used mainly for optimizing queries over SQL views [26], optimizing distributed XPath queries [13] and translating object queries to SQL [47].

In [26] the authors propose a mechanism to avoid evaluating parts of SQL queries stated in terms of previously defined and materialized views, the motivation for this being improved performance in terms of reduced execution time or security reasons.

In contrast to the work in [26], application of the PARQ algorithm on entire SPARQL queries to UPVs reduce these queries to much simple expressions during compile time and provides substantial improvement of both optimization time and execution time.

In [13] XPath queries are executed over an XML trees fragmented over a number of sites on the web. To minimize the response time of a query no more computation than what is strictly necessary at each site to answer the query should be done. The main idea is to divide the query into pieces and send each piece of the query independently and in parallel to each site where the original query is partially answered and then let a coordinator site combine these partial results to the final result. Each XML tree fragment is visited only once.

In contrast to [13], that use partial evaluation in the sense of executing pieces of the query in parallel, the PARQ algorithm in SWARD reduces (specializes) queries to general UPVs during compile time into smaller and faster queries producing substantial improvements in optimization as well as query execution time.

In [47] an algorithm for translation of object queries to SQL is presented. An application of the algorithm could be object-oriented interfaces to relational databases. Object queries are first transformed into canonical queries expressed in a deductive database. These queries can contain class variables and attribute variables. Since schema information is usually not accessible in SQL, partial evaluation is used to instanciate class and attribute variables in the object queries before translation of them to SQL.

Similar to in [47], partial evaluation in SWARD also access schema information for translation of SPARQL queries to SQL. However, in SWARD schema information is used to reduce the size of the query shown to make queries to UPVs significantly more efficient.

10.4 Preservation of Foreign Key Information in RDF Schema

In [35] the authors propose a way to explicitly state foreign key information in RDFS. This is done by extending the RDFS standard with a new meta-class for representing foreign keys. The work is purely theoretical.

SWARD makes foreign key information in the back-end DBMS explicit in UPVs by representing them as properties that relate instances of classes in an RDFS description. Instead of introducing new RDFS meta-classes that are not a part of the current RDFS standard, as in [35] SWARD use existing RDFS meta-classes for representing foreign key information.

10.5 Disjunctive Query Optimization

In SWARD view expansion of UPVs produces conjunctions of large disjunctions. SWARD thus processes a class of very large disjunctive queries.

In [39][17] special approaches for dealing with disjunctions in queries are proposed. In [39], the query is first transformed to an expression on DNF. Multiple selection conditions on the same table, but perhaps in different branches of the disjunction, are combined and then optimized together reducing the number of SQL table scans and joins in the produced query plan. In [17], instead of normalizing the query, the characteristic of the given query is utilized for it to be optimized for better performance by using special purpose algebraic operators.

In contrast to [39] [17], in SWARD partial evaluation of declarative query fragments is used to systematically reduce disjunctive expressions to simple conjunctions, thus totally eliminating the need for normalisation. This reduction dramatically improves total query processing times for SPARQL queries to UPVs.

10.6 Property Table Representation of Data

In [2] it is demonstrated that relational databases where data is stored in a conventional horizontal scheme is not a realistic alternative for storage of constantly evolving, sparsely populated data, such as e-commerce data.

Instead a new vertical scheme for storage of e-commerce data is proposed where each row in the horizontal table is divided into several rows (one for each column in the horizontal table) on the format:

```
<Oid(object identifier), Key(column name), Val(column value)>
```

The object identifier is the value of the key column of the horizontal table and is used for associating rows in the vertical table representation with each other. The only schema information stored in the vertical table is column names. This is similar to the storage schemes used in column oriented databases [60].

UPVs in SWARD can be seen as a property table view of the entire relational database contents *including* its schema. Unlike [2], SWARD addresses the challenges to efficiently optimize queries over very large property tables that also include schema data. PARQ significantly improve query processing for such queries without modifying the DBMS kernel.

The *unpivot* algebra operator in [19] transforms a regular horizontal table into a property table by removing a number of columns. Extra rows are added to preserve the column names and values from the wide representation. By unpivoting a wide table on the primary key column a property table similar to the vertical table in [2] is produced. Property tables may or may not store their result depending how they are used in queries.

In [19], the unpivot and pivot operators are used for data modelling and data analysis while UPVs in SWARD view entire relational databases as RDF. Unlike in SWARD that acts like a pre-processor to SQL, the approach proposed in [19] is intrusive in that the unpivot and pivot operators are developed for use inside the relational RDBM. Also, all optimizations presented are applied on the algebraic level to improve only execution time of queries. For queries to UPVs it is necessary to also optimize the optimization time itself and this is done by SWARD through systematic partial evaluation of query fragments before cost based optimization of the produced SQL in the back-end DBMS.

10.7 SchemaSQL Server

The work on SchemaSQL [34] supports querying both data and meta-data in a relational database by providing special syntax to query the schema rather than our uniform UPV. They do not use partial evaluation but an ad-hoc special implementation of the query processor. Meta-data is not stored in a main memory database as in SWARD but in a special purpose relational database internal to the SchemaSQL Server. SchemaSQL queries are cost based optimized first during rewriting of SchemaSQL to SQL in the SchemaSQL Server and later on again optimized in the local data source. In contrast, SWARD acts as a pre-processor of SQL queries leaving all the cost based optimization to the back-end relational database. Both schema and content are defined by the general UPV definition. Partial evaluation is used during query processing for systematic specialization of the general UPV to substantially reduce the query size before cost-based query optimization by the back-end relational database. PARQ enables clean query processing without any special purpose query transformations.

11 Summary and Future Work

A system, SWARD, has been implemented for scalable processing of conjunctive SPARQL queries over general RDF Schema based views of relational databases. Relational databases are viewed in terms of RDF Schema based universal property views (UPVs) representing both relational schema and data. A UPV is automatically generated, given a relational database and two mapping tables, *cMap* and *pMap*, specifying how to map tables and columns to RDF Schema classes and properties, respectively. For augmented UPVs a third table, *rMap*, also has to be specified to map special relationship identifiers to foreign key columns. A UPV definition is a large disjunctive view which requires substantial reduction for efficient query processing.

To speed up query processing of queries to UPVs a new general partial evaluation algorithm, the PARQ algorithm, was presented that does systematic compile time evaluation of specific primitive predicates to produce a reduced query. The algorithm is simple and efficient, but is yet a very powerful technique for reduction of any query. To guarantee that the query size is never increased by PARQ, the algorithm evaluates at compile time only primitive predicates that produce empty or single tuple results. In this Thesis PARQ was applied on conjunctive SPARQL queries over large disjunctive UPVs it was shown that application of PARQ on such queries guarantees to reduce them into conjunctions without need for normalization. PARQ thus provides simple and scalable processing of SPARQL queries to large UPVs without need for ad hoc optimization tricks.

The names of the primitive predicates to evaluate at compile time by PARQ are explicitly pre-specified to avoid evaluating expensive predicates at compile time. When optimizing SPARQL queries to the UPVs only predicates stored in main memory and the external predicate *iMap* are evaluated at compile time, so PARQ is not accessing the database.

Furthermore, it was also demonstrated how the PARQ algorithm enables the programmer to develop elegant and clean query processing mechanisms, which are automatically specialized by partial evaluation for efficiency.

It was shown that query processing based on traditional view expansion followed by normalization, END (Expand – Normalize – Decompose), generates scalable execution plans for queries to UPVs. However, the plans become huge when the size of the query or the UPV increases, due to generation of unreasonable large normalized expressions. The END strategy is therefore infeasible for real-world SPARQL queries and UPVs.

To improve the query processing scalability we applied the PARQ algorithm on END; the modified algorithm is called END-P (END with Partial Evaluation). It was shown that PARQ substantially reduces the query optimization time in END-P, since a conjunctive query over the large disjunctive UPV is reduced to a simple conjunctive query and no normalization is needed.

An alternative strategy DVS-P (Dynamic View Selection with Partial Evaluation) was defined by applying PARQ on a query to a UPV definition where precompiled property views are dynamically selected. In this case, PARQ determines at compile time all property views that must be selected to answer content queries. DVS-P produces the same reduced queries as END-P without any normalization. The difference between END-P and DVS-P is that DVS-P applies PARQ to select the views in the UPV definitions to expand, while END-P applies PARQ after full view expansion. Therefore, for realistic content queries to large UPV definitions, query processing with DVS-P is faster than END-P.

In this Thesis it was further presented how the basic UPV framework is generalized in SWARD to preserve information about primary as well as foreign and composite keys in the generated UPVs. ER relationships are made implicit when translated to the relational model using foreign keys. When relational data is viewed in RDFS these relationships are made explicit again through the addition of so called *class membership views* to the content view of the UPV. Composite keys are not supported by RDFS but needs to be handled since they are supported by the relational model. They are represented by concatenation of composite primary key attribute values.

Furthermore, it was demonstrated how to incorporate class memberships of mapped instances by adding *class membership views* to the of the UPV definition, which express the class memberships of all mapped instances from the back-end DBMS.

It this work, it was presented that END-P produces scalable query processing also for the important subclass of hybrid queries that dynamically retrieve mapped properties from a class, while DVS-P is less suitable there.

In the future it should be investigated for what further class of SPARQL queries the presented techniques are applicable, such as handling of hybrid queries dynamically selecting mapped classes and their mapped instances, disjunctive SPARQL queries, and SPARQL queries with OPTIONAL triple patterns.

It should also be investigated how an object representation of RDF data, where RDF resources become instances of type *Resource*, could be used to represent RDF Schema *typed* literals in UPVs.

The current implementation of SWARD runs as a pre-processor to SQL in the back-end relational DBMS. If the system was part of the database server, relational algebra could directly be generated from the reduced query.

An interesting issue is also how to provide mediation by combining UPVs over different databases.

Finally, we believe that there are many other opportunities for using partial evaluation in complex query processing. The PARQ algorithm is guaranteed to converge fast and offers many opportunities for substantial query reduction in a systematic and controlled fashion.

Summary in Swedish

Informationutsökning i RDF Schema vyer av relationsdatabaser

Mängden data på Internet idag och dess brist på semantik gör att det blir allt svårare att komma åt önskad information. Sökmotorer baserade på fritextutsökning av data genererar för många och irrelevanta resultat.

Ett annat problem uppstår då information skall kombineras på ett meningsfullt sätt. Till exempel, en applikation som hämtar data från databaser med olika struktur och innehåll måste kunna avgöra om data från en kolumn ’A’ i en databas har samma mening som data från en kolumn ’A’ i en annan databas. Detta är något som är utmanande då sådan information saknas i databasschemat.

Det är tydligt att det behövs ett entydigt sätt att beskriva information för att underlätta utsökning och kombinering av denna. M.h.a. Resource Description Framework (RDF) märks information upp med egenskaper (metadata) som beskriver dess mening.

RDF Schema (RDFS) är definerat i termer av RDF och används för att klassificera information och definiera egenskaper hos dessa klasser. Frågespråket SPARQL är standard för utsökning av RDF-data.

RDFS-data lagras och görs sökbart i speciella lagringssystem för RDF. Då stora mängder data idag fortfarande är lagrad i relationsdatabaser är det viktigt att den också görs sökbar i SPARQL. Detta sker genom att informationen i relationsdatabasen konverteras till RDF data och laddas ner till ett lagringssystem för RDF. Då stora mängder information lagras på flera ställen blir detta dock en kostsam lösning m.a.p. lagringsutrymme och hantering av data.

Ett bättre sätt att göra relationsdatabaser sökbara i SPARQL är att tillhandahålla ickematerialiserade RDFS-baserade representationer, s.k. RDFS-vyer av relationsdatabaser. En RDFS-vy genererar vid utsökningstillfället en tillfällig RDFS representation av en relationsdatabas. Denna RDFS mappning lagras inte i något system utan används bara för att bevara en given fråga. På så sätt lagras inte samma information på flera ställen.

RDFS-vyer bör vara generella och strukturerade för att kunna defineras över godtycklig relationsdatabas samt undvika fel som annars vanligen introduceras i ad-hoc lösningar.

Relationsdatabaser designas ofta med Entity Relationship (ER) modellen. ER modellen är ett högnivåspråk för att på en konceptuell nivå skapa ett diagram som specificerar en relationsdatabas m.h.a typer och relationer mellan dessa typer. ER diagrammet översätts sedan till ett relationsdatabasschema. Då ett sådant schema är mer implementationsspecifikt än ett ER diagram är vissa element i ER modellen, som exempelvis relationer mellan typer, implicit representerade i relationsdatabasschemat. Kompletta RDFS-vyer av relationsdatabaser gör sådana element explicita igen.

Att processera frågor mot RDFS-vyer är utmanande av två anledningar. För det första leder sättet att representera data i RDF till stora frågor mot komplexa vyer. Traditionell processering av sådana frågor genererar enorma uttryck internt och leder därför till orealistiskt långa processeringstider. För det andra är det av största vikt att frågor mot RDFS-vyer av relationsdatabaser noggrant optimeras innan exekvering, då frågorna körs över stora datamängder.

För att utreda dessa frågor har vi utvecklat systemet Semantic Web Abridged Relational Databases (SWARD) för effektiv processering av SPARQL-frågor mot kompletta RDFS-vyer av relationsdatabaser. Sådana vyer genereras automatiskt av SWARD med ett minimum av information från användaren.

I SWARD representeras både innehåll och schemainformation i en relationsdatabas som en enda stor disjunktiv vy. En sådan vy kallas en universiell egenskapsvy, eller UPV. En UPV är en RDFS-mappning av en relationsdatabas. Den definieras som unionen av en schemavy (schemainformationen i databasen) och en datavy (innehållet i databasen).

En UPV genereras automatisk av SWARD givet att användaren anger hur tabeller och kolumner mappas mot RDFS-klasser samt egenskaper hos dessa klasser. Under frågeprocesseringen i SWARD översätts SPARQL-frågor till interna uttryck. Sådana uttryck innehåller SQL-fragment som används för att hämta data från den underliggande relationsdatabasen.

SPARQL-frågor mot UPVer hämtar antingen schemainformation, innehåll eller både och från den underliggande relationsdatabasen. SWARD hanterar effektivt samtliga typer av SPARQL-frågor mot UPVer. Det är dock speciellt utmanande att processera frågor mot relationsdatabasens innehåll då detta ofta är väldigt stort.

Vi har utvecklat en allmän algoritm, PARtial evaluation of Queries (PARQ) för att förenkla frågor mot komplexa vyer. Algoritmen är baserad på en teknik som kallas partialevaluering. Partialevaluering möjliggör utveckling av eleganta och enkla program som sedan automatiskt specialiseras till effektiva (snabbare och/eller mindre) program. PARQ reducerar frågan genom att iterativt utvärdera delar av frågan till dess att uttrycket inte går att förenkla mer.

Våra experiment visar att partialevaluering av frågor mot UPVer leder till avsevärt enklare och mindre frågor som går betydligt snabbare att processera än orginalfrågan.

Acknowledgement

First and foremost I would like to thank my supervisor Tore Risch for sharing his knowledge and enthusiasm.

I would also like to thank my former and current colleagues Kjell, Timour, Milena, Ruslan, Sabesan, Erik, and Silvia for their support.

Finally, I would like to thank my friends and family for their great patience and for always being there to support me.

This work was supported in part by the EU project Advanced eGovernment Information Service Bus, FP6-IST-2004-26727.

References

[1] D. J. Abadi, A. Marcus S. R. Madden, and K. Hollenbach , Scalable Semantic Web Data Management Using Vertical Partitioning, Proc. *33rd Intl. Conf. on Very Large Databases (VLDB 2007)*, pp. 411-422, 2007.

[2] R. Agrawal, A. Somani, and Y. Xu: Storage and Querying of E-Commerce Data, Proc. *27th Intl. Conf. on Very Large Databases (VLDB 2001)*, pp 149-158, 2001.

[3] A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, W. H. Freeman and Co., New York, USA, 1995.

[4] AllegroGraph, http://agraph.franz.com/allegrograph/.

[5] P. H. Andersen: Partial Evaluation Applied to Ray Tracing, *Software Engineering in Scientific Computing*, 1996, http://repository.readscheme.org/ftp/papers/topps/D-289.pdf.

[6] L. Andersen: Partial Evaluation of C and Automatic Compiler Generation (extended abstract), Proc. *4th Intl. Conf. of Compiler Constructions (CC 1992)*, pp. 251-257, 1992.

[7] L. Beckman, A. Haraldson, O. Oskarsson, and E. Sandewall: A Partial Evaluator, and Its Use as a Programming Tool, *Artificial Intelligence*, 7(4):319-357, 1976.

[8] P. A Bernstein, N. Goodman, E. Wong, C. Reeve, and J. B. Rothnie: Query Processing in a System for Distributed Databases (SDD-1), *ACM Trans. on Database Systems*, 6(4), 1981.

[9] C. Bizer and A. Seaborne: D2RQ -Treating Non-RDF Databases as Virtual RDF Graphs (Poster). Proc. *3rd Intl. Semantic Web Conf. (ISWC 2004)*, 2004.

[10] C. Bizer and R. Cyganiak: D2R Server - Publishing Relational Databases on the Semantic Web (Poster). Proc. *5th Intl. Semantic Web Conf. (ISWC 2006)*, 2006.

[11] D. Brickley and R. V. Guha: RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation 10 February 2004, http://www.w3.org/TR/rdf-schema/.

[12] J. Broekstra, A. Kampman, and F. van Harmelen: Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema. Proc. *1st Intl. Semantic Web Conf. (ISWC 2002)*, 2002.

[13] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis: Using Partial Evaluation in Distributed Query Evaluation, Proc. *32nd Intl. Conf. on Very Large Databases (VLDB 2006),* pp 211-222, 2006.

[14] P. P. S. Chen: The entity-relationship model: towards a unified view of data, *ACM TODS*, 1(1): 9-36, 1976.

[15] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan: An Efficient SQL-based RDF Querying Scheme, Proc. *31st Intl. Conf. on Very Large Databases (VLDB 2005)*, pp 1216-1227, 2005.

[16] V. Christophides, G. Karvounarakis, A. Magkanaraki, D. Plexousakis, and V. Tannen: The ICS-FORTH Semantic Web Integration Middleware (SWIM), *IEEE Data Engineering Bulletin*, 26(4), 2003.

[17] J. Claußen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn: Optimization and Evaluation of Disjunctive Queries, *IEEE Trans. Knowledge and Data Eng.* 12(2): 238-260, 2000

[18] E. F. Codd, A relational model of data for large shared data banks, *Communications of the ACM*, 13(6):377-387, 1970.

[19] C. Cunningham, C. A.Galindo-Legaria, and G. Graefe: PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS, Proc. *30th Intl. Conf. on Very Large Databases (VLDB 2004)*, pp 998-1009, 2004.

[20] S. Decker, F. van Harmelen, J. Broekstra, M. Erdmann, D. Fensel, I. Horrocks, M. Klein, and S. Melnik: The Semantic Web - on the Roles of XML and RDF, *IEEE Internet Computing*, Sept./Oct. 2000.

[21] A. Deutsch, A. Popa and V. Tannen: Physical Data Independence, Constraints and Optimization with Universal Plans. Proc. *25th Intl. Conf. on Very Large Databases (VLDB 1999)*, pp 459-470, 1999.

[22] Dublin Core Meta-data Initiative, Dublin Core Metadata Element Set, V 1.1, http://dublincore.org/documents/dces/.

[23] R. Elmasri and S. B. Navathe: *Fundamentals of Database Systems*, Addison-Wesley, 5th edition, 2007.

[24] G. Fahl and T. Risch: Query Processing over Object Views of Relational Data, *VLDB Journal*, 6(4):261-281, 1997.

[25] Y. Futamura: Partial evaluation of Computation Process - an Approach to a Compiler-Compiler, *Systems Comput. Controls*. 25:45-50, 1971.

[26] P. Godfrey and J. Gryz: Partial Evaluation of Views, *Journal of Intelligent Information Systems*, 16(1): 21-39, 2001.

[27] K. Grant Clark, A. Schain, and B. Parsia: Semantic Web @ NASA, http://xtech06.usefulinc.com/schedule/paper/147.

[28] P. Hayes: RDF Semantics, W3C Recommendation 10 February 2004, http://www.w3.org/TR/rdf-mt/.

[29] N. D. Jones: An Introduction to Partial Evaluation, *ACM Computing Surveys*, 28(3), 1996.

[30] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl: RQL: A Declarative Query Language for RDF, Proc. *Intl. World Wide Web Conf. (WWW 2002)*, 2002.

[31] L. Kerschberg, P. D. Ting, and S. B. Yao: Query optimization in a star computer network, *ACM Trans on Database Systems*, 7(4), 1982.

[32] G. Klyne and J. J. Carroll: Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation 10 February 2004, http://www.w3.org/TR/rdf-concepts/.

[33] H. J. Komorowski: Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: a Theory and Implementation in the Case of Prolog, Proc. *9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1982)*, 1982.

[34] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian: SchemaSQL – A Language for Interoperability in Relational Multi-database Systems, Proc. *22nd Intl. Conf. on Very Large Database (VLDB 1996)*, pp 239-250, 1996.

[35] G. Lausen, M. Meier, and S. Schmidt: SPARQling Constraints for RDF, Proc. *11th Int. Conf. on Extending Database Technology (EDBT 2008)*, 2008.

[36] W. Litwin and T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. In *IEEE Transactions on Knowledge and Data Engineering*, 4(6):517-528, 1992.

[37] P. Lyngbaek: OSQL: A Language for Object Databases, *Tech. Report*, HP Labs, HP-DTD-91-4, 1997.

[38] S. Manegold, P. A. Boncz, and M. L. Kersten: Optimizing database architecture for the new bottleneck: memory access. *VLDB Journal*, 9(3):231-246, 2000.

[39] M. Muralikrishna: Optimization of Multiple-Disjunct Queries in a Relational Database System, *Technical Report no. 750*, Univ. of Wisconsin-Madison, Feb. 1988, http://www.cs.wisc.edu/techreports/1988/TR750.pdf.

[40] W. Neidl, B. Wolf, C. Qu, S. Decker, M. Sinek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch: EDUTELLA: A P2P Networking Infrastructure Based on RDF. Proc. *11th Intl. World Wide Web Conference (WWW 2002)*, http://user.it.uu.se/~torer/publ/WWW-Edutella.pdf, 2002.

[41] Open Directory RDF Dump, http://rdf.dmoz.org/

[42] OWL Web Ontology Language, http://www.w3.org/TR/owl-features/

[43] Partial Evaluation: http://partial-eval.org/.

[44] J. Petrini and T. Risch: SWARD: Semantic Web Abridged Relational Databases, Proc. *6th Intl. Workshop on Web Semantics (WEBS 2007)*, http://user.it.uu.se/~udbl/publ/WEBS07.pdf, 2007.

[45] J. Petrini and T. Risch: Processing Queries over RDF views of Wrapped Relational Databases, Proc. *1st Intl. Workshop on Wrapper Techniques for Legacy Systems (WRAP 2004)*, http://user.it.uu.se/~udbl/publ/WRAP04.pdf, 2004.

[46] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang: Optimistic Incremental Specialization Streamlining a Commercial Operating System, Proc. *15th ACM Symposium on Operating System Principles*, 1995

[47] X. Qian and L. Raschid: Query Interoperation Among Object-Oriented and Relational Databases, Proc. *11th Intl. Conf. on Data Engineering (ICDE 1995)*, pp 271-278, 1995.

[48] RDF Data Access Use Cases and Requirements, W3C Working Draft 25 March 2005, 2005, http://www.w3.org/TR/rdf-dawg-uc/.

[49] RDF Site Summary 1.0, http://web.resource.org/rss/1.0/

[50] RDQL - A Query Language for RDF, W3C Member Submission 9 January 2004, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[51] T. Risch: Functional Queries to Wrapped Educational Semantic Web Meta-Data, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, Springer, ISBN 3-540-00375-4, 2003.

[52] T. Risch, V. Josifovski, and T. Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, Springer, ISBN 3-540-00375-4, 2003.

[53] D. Sahlin: An Automatic Partial Evaluator for Full Prolog. *PhD thesis*, Swedish Institute of Computer Science, 1991, http://citeseer.ist.psu.edu/sahlin91automatic.html.

[54] P. Sellinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price: Access Path Selection in a Relational Database Management System, *Proc. ACM SIGMOD Conf. Management of Data*, pp. 23-34, 1979.

[55] G.Serfiotis: Optimizing and Reformulating RQL Queries on the Semantic Web. *Master Thesis*, university of Crete, 2005, http://139.91.183.30:9090/RDF/publications/serfiotis.pdf.

[56] G.Serfiotis, I.Koffina, V.Christophides and V.Tannen: Containment and Minimization of RDF/S Query Patterns, *Proc. 4th Intl. Semantic Web Conf (ISWC 2005)*, 2005.

[57] D. Shipman: The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140-173, 1981.

[58] SPARQL Query Language for RDF, W3C Recommendation 15 January 2008, http://www.w3.org/TR/rdf-sparql-query/.

[59] SquirrelRDF, http://jena.sourceforge.net/SquirrelRDF/.

[60] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik: C-Store: A Column-oriented DBMS, Proc. *31st Intl. Conf. on Very large Databases (VLDB 2005)*, pp 553-564, 2005.

[61] Semantic Web for Earth and Environmental Terminology, http://sweet.jpl.nasa.gov/ontology/.

[62] SWARD, http://user.it.uu.se/~udbl/sward.html.

[63] E. Tambouris, G. Kavadias, and E. Spanos: The Government Markup Language (GovML), *Journal of E.Government* 1(2), 2004.

[64] TPC-H benchmark, http://www.tpc.org/tpch/.

[65] J. D. Ullman, H. Garcia-Molina and J. Widom: *Database Systems: The Complete Book*, Prentice Hall, Upper Saddle River, NJ, USA, 2001.

[66] Uniprot RDF dataset , http://dev.isb-sib.ch/projects/uniprot-rdf/.

[67] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds: Efficient RDF Storage and Retrieval in Jena 2, Proc. *VLDB Workshop on Semantic Web and Databases (SWDB 2003)*, pp 131-150, 2003.

[68] E. Wong, K Youssefi: Decomposition – A Strategy for Query Optimization, *ACM Trans. on Database Systems*, 1(3):223-241, 1976.

[69] Wordnet RDF dataset, http://wordnet.princeton.edu/~agraves/index.htm.

Appendix: SWARD Query Interfaces

SWARD can be queried by application programs and users using the semantic web query languages SPARQL, RDQL, or a subset of SQL. Figure 43 shows the query interface of the SWARD system.

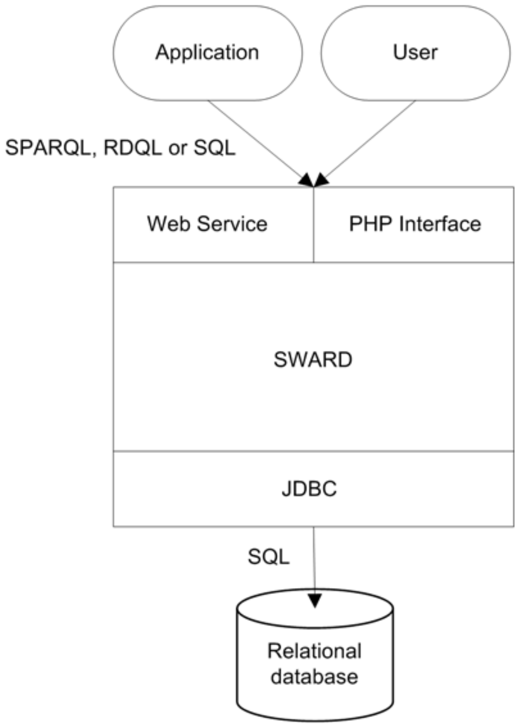


Figure 43: SWARD system query interface

SWARD provides a *PHP interface* allowing users to query SWARD through their web browser. SWARD can also be installed as a *web service* on a Windows server computer and called from clients through a web service based interface. A Java API is provided as a separate Java JAR file for transparently calling SWARD as a web service from Java applications.

The following interface classes are provided by the *SWARD Java API*:

```
class SWARD {

// Constructors
  public SWARD(String URL){}

// Execute SPARQL query over a UPV
  public RDFScan SPARQL(String SPARQLString){}

// Execute RDQL query over a UPV
  public RDFScan RDQL(String RDQLString){}

// Execute SQL query over a UPV
  public RDFScan SQL(String SQLString){}

}

class RDFScan {

//Get next result in RDFScan
  public Vector Next(Scan);

//Check for end of RDFScan
  public Boolean EOF(Scan);

}
```

A new instance of the class *SWARD* is constructed with the argument *URL* as the URL of an SWARD web service.

Once an SWARD object is constructed the application or user can issue queries in SPARQL, RDQL, or SQL through methods named *SPARQL*, *RDQL* or *SQL*, respectively.

The result of a query is an instance of class *RDFScan*. One can iterate through the scan by the method *Next* and test for end of scan with method *EOF*. Each call to *Next* returns a Java *Vector* object with structure

$\{\{VAR_1,VAL_1\},\ldots\{VAR_n,VAL_n\}\}$

where *n* is the width of the result tuples, VAR_i is the name of the *i*:th result variable, and VAL_i is its corresponding value.

In Example 63 the content query Q2 is presented in RDQL.

```
SELECT ?cust,?mkt
WHERE (?order,co:OrderID,'1'),
      (?order,co:OrderCustomer,?ocust),
      (?cust,co:CustID,?ocust),
      (?cust,co:Market, ?mkt)
```

Example 63: Content query Q1 in RDQL.

The triple patterns are specified in RDQL using the notation *(s,p,v)* where *s* (subject), *p* (property), and *v* (value) are constants or variables. In Example 64 the same query is expressed in SQL.

```
SELECT C4.S,C4.V
FROM COMP C1,
     COMP C2,
     COMP C3,
     COMP C4
WHERE C1.P = co:OrderID       AND
      C2.P = co:OrderCustomer AND
      C3.P = co:CustID        AND
      C4.P = co:Market        AND
      C1.V = '1'              AND
      C1.S = C2.S             AND
      C2.V = C3.V             AND
      C3.S = C4.S
```

Example 64: Content query Q1 in SQL.

The FROM clause in the SQL query specifies an identifier for the UPV to query. With SQL syntax, the UPV *Comp* is simply treated as a regular relational table and the FROM clause can not be omitted as in RDQL and SPARQL queries.

Notice the awkward form of the SQL (Example 64) query compared to the same query expressed in RDQL (Example 63) and in SPARQL (Example 9). The reason for this is that RDQL and SPARQL bear a resemblance to domain calculus [23], with all variables in the query implicitly existentially quantified and with variables substituted when possible. This enables a more compact representation of queries to UPVs with SPARQL and RDQL than with SQL which is based on tuple calculus [23] suited for queries to relational tables. In practice this means that an SQL query to a UPV will make self joins over many aliased row variables (*c1*, *c2*, *c3* and *c4*). In Example 64 the conditions on variables are defined in terms of one comparison with a constant and three UPV joins.