The Report Committee for Diego Leonardo Funes
Certifies that this is the approved version of the following report:

Evaluation of Relational Database Implementation of Triple-Stores

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Suzanne Barber

Co-Supervisor:

Daniel Miranker

Evaluation of Relational Database Implementation of Triple-Stores

by

Diego Leonardo Funes, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2011

Dedication

To Roxanne, Emma and Baby Lili.

Abstract

Evaluation of Relational Database Implementation of Triple-Stores

Diego Leonardo Funes, M.S.E.

The University of Texas at Austin, 2011

Supervisors: Suzanne Barber and Daniel Miranker

The Resource Description Framework (RDF) is the logical data model of the Semantic Web. RDF encodes information as a directed graph using a set of labeled edges known formally as resource-property-value statements or, in common usage, as *RDF triples* or simply *triples*. Values recorded in RDF triple form are either Universal Resource Identifiers (URIs) or literals. The use of URIs allows links between distributed data sources, which enables a logical model of data as a graph spanning the Internet. SPARQL is a standard SQL-like query language on RDF triples.

This report describes the translation of SPARQL queries to equivalent SQL queries operating on a relational representation of RDF triples, and the physical optimization of that representation using the IBM DB2 relational database management system. Performance was evaluated using the Berlin SPARQL Benchmark. The results show that the implementation can perform well on certain queries, but more work is required to improved overall performance and scalability.

Table of Contents

List of Tables

List of Figures

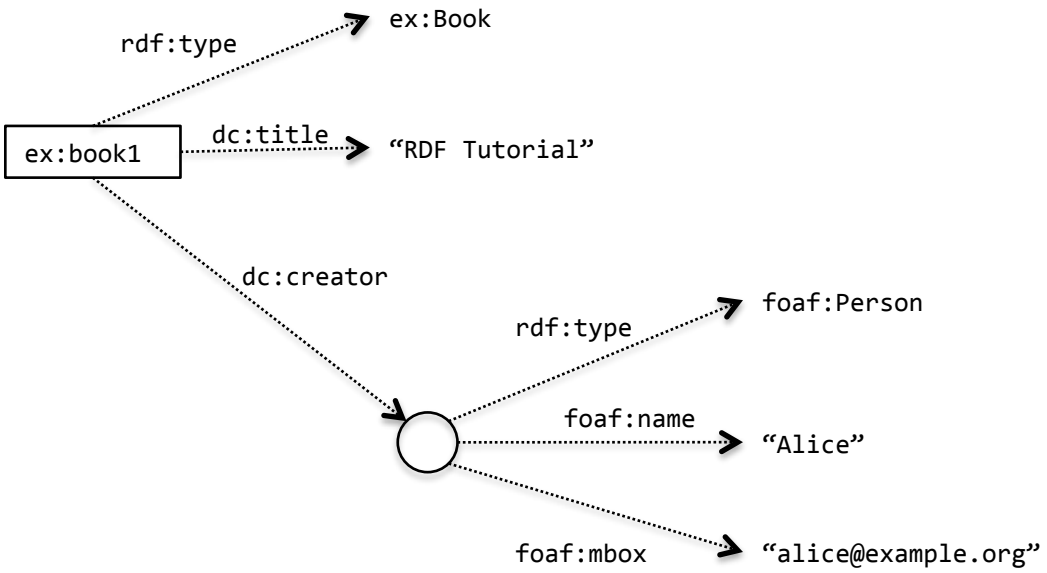
Chapter 1: Introduction

The Semantic Web is a technology for the evolution of the existing world wide web of documents (web pages) to a world wide web of *data*. Data is not only made available but also described by metadata represented in ontologies suitable for machine processing. This approach is expected to simplify existing services (web search, for example) or enable new services. The realization of the Semantic Web requires a standard and flexible data model that facilities the integration of data sources while allowing these data sources to evolve in a decentralized environment like the World Wide Web. This chapter briefly describes some of the standards that comprise the Semantic Web.

RESOURCE DESCRIPTION FRAMEWORK

The Resource Description Framework, or RDF, is a data model designed to facilitate data merging and evolution of the logical schema of the data. At its core RDF describes resources using subject-property-object statements, also referred to as RDF triples, or simply 'triples'. Because RDF information is encoded as set of triple statements, systems that store and manage RDF data are referred to as *triplestores*. The set of triples in a RFD dataset form a directed graph that encodes the properties of each resource and the relations between resources. Figure 1 shows an example RDF graph and its triple representation describing some properties of a book and its author

a) Graph representation



b) Triple representation

```
{
  (ex:book1, rdf:type, ex:Book)
  (ex:book1, dc:title, “RDF Tutorial”)
  (ex:book1, dc:creator, _:x)
  (_:x, rdf:type, foaf:Person)
  (_:x, foaf:name, “Alice”)
  (_:x, foaf:mbox, “alice@example.org)
}
```

Figure 1: Example RDF Graph.

All RDF features relevant to this report are shown in the Figure above. The notation *prefix:name* is a short-form used to represent universal resource identifiers, or URIs. This notation is the same used by XML namespace where the prefix expands to a proper URL, so *rdf:type* would expand to *http://www.w3.org/1999/02/22-rdf-syntax-ns#type*. The graph shows a resource *ex:book1* of type *ex:Book* with the title “RDF Tutorial” and a creator (author) described as another resource. Not all resources in a RDF graph need to be explicitly named since in many cases the identity of the resource is

not as relevant as the set of properties that resource possesses. In this example, the author of the book is represented by a blank node of type *foaf:Person* with name "Alice" and email alice@example.org. This is a clear example where the identifier used to represent a resource (the author) is likely to be irrelevant, but the properties associated to the resource are likely to be of interest.

There are several RDF serialization formats in use today. RDF/XML is typically the choice for machine-readable RDF documents since it allows the reuse of existing XML tools and expertise. However, RDF/XML tends to hide the intuitive graph structure of RDF data and as a consequence it is common to use other more human-readable formats, such N-Triples and Turtle. The specifics of these serialization formats will not be described in this report since they are not rrelevant to the discussion of RDF data management. It is important highlight that RDF should be viewed as a logical data model, and not confused with the language or format used for exchange (XML, for example).

RDF places some restrictions on the data types allowed by each term in a triple. Subject terms may only be named resources (URIs) or blank nodes[1], properties can only be URIs, and objects can be URIs or literals of any type. Note that there are no restrictions on what a property should refer to, or any other relations between resources and properties. It would still be a perfectly valid RDF graph if, for example, the *ex:book1* resource had an additional *rdf:type* property with a value *foaf:Person*. Encoding domain specific information (for example, that the set of resources of type Book and Person is disjoint) is not handled at the level of the data model and delegated to description languages like RDF Schema and OWL.

[1] Blank nodes are usually represent as *_:x* where *x* is an arbitrary identifier required to be locally unique.

RDF SCHEMA AND OWL

RDF Schema (RDFS) and the Web Ontology Language (OWL) provide the mechanisms to define the logical schema of the RDF data by specifying relations between resources and properties. RDF Schema defines a primitive modeling language capable of specifying the following constraints [3]:

1. Class and sub-class relations between resources.
2. Property and sub-property relations.
3. Property domain and range

Subclass relations refer to the well-known concepts used in many object-oriented languages, where instances belong to a class and all its parent classes. Property hierarchies are a less familiar concept, but it's the same idea of containment used for instances applied to properties. Finally, property domain and range allow the specification of constraints about the classes a property can be applied to and what values the property can take.

The set of data modeling features provided by RDF Schema is not expressive enough for modeling all the data domains required by the Semantic Web. OWL is the modeling language intended to fill this need. Some of the features provided by OWL not available in RDF Schema include [3]:

1. Define disjoint classes.
2. Combination of classes using set operations such as union, intersection and complement.
3. Cardinality restriction of class sets.
4. Additional property features like uniqueness, transitivity and inverse.

A full description of RDF Schema and OWL is out of the scope of this report[2]. Perhaps the most relevant aspect of these languages from our point of view is that the RDF Schema and OWL specifications are also encoded as RDF graphs. RDF Schema and OWL metadata are also first-class RDF data, and can evolve along side the data they describe. The importance of this feature from the RDF data management perspective is that any efficient design for the storage and query for RDF is automatically applicable to RDF Schema and OWL data.

SPARQL QUERY LANGUAGE

SPARQL [20] provides a query language at the level of the RDF data model. Queries are described as graph patterns that the RDF data must match to be returned as a result of the query. Graph patters are in turn composed of triple patterns. A triple pattern may specify fixed terms that a given triple must match and variable terms that can take any value. The simplest SPARQL query would have a single triple pattern as shown in Figure 2.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?x
WHERE {
    ?x rdf:type foaf:Person .
}
```

Figure 2: SPARQL query with a single triple pattern.

The SELECT clause of the query specifies what variables should be returned as results. The WHERE clause includes all the triple patterns the results must match. In this case the query will return the resource identifier of all resources of type *foaf:Person*.

[2] For details, refer to the w3c recommendations for RDFS [16] and OWL 2 [14].

However, in this case the resource identifier is not likely to be the information someone may want to ask about a person. Figure 3 expands the query to ask for more interesting properties about a person.

```
SELECT ?name ?email
WHERE {
    ?x rdf:type foaf:Person .
    ?x foaf:name ?name .
    ?x foaf:mbox ?email
}
```

Figure 3: SPARQL query with join variable.

Now the resource identifier is not used as result, it is only used to specify that the properties of interest should refer to the same resource. In this case the query is asking for the name and email of all resources of type *foaf:Person*. One feature of this query is that for a person to show up in the result it must have a name AND an email address. It is probably safe to assume that a person will always have a name, but there may be people without an email address that will not be included in the results. To handle this use case SPARQL defines the OPTIONAL operator to declare that certain triple patterns are optional. If an optional triple pattern is not matched, the results will indicate that the any optional projected variables are unbound. Figure 3 modifies the query to make email addresses optional.

```
SELECT ?name ?email
WHERE {
    ?x rdf:type foaf:Person .
    ?x foaf:name ?name .
    OPTIONAL { ?x foaf:mbox ?email }
}
```

Figure 3: SPARQL query with optional pattern.

Someone looking at the previous query may ask, "Who doesn't have an email address?" To answer this question SPARQL provides a FILTER operator that allows the specification of arbitrary expressions to determine what should be returned as a result. Only results that evaluate the expression to true are returned. Filter expression can include the familiar arithmetic and logical operators found in most programming languages, along with additional functions to support regular expression matching, for example. For the specific question on how to find results that do not match an optional graph pattern SPARQL defines a *bound(x)* function that returns *true* if the variable is bound to a value. Figure 4 shows how the *bound(x)* function may be used to filter out persons that have an email address and only return the name of people with no email addresses.

```
SELECT ?name
WHERE {
    ?x rdf:type foaf:Person .
    ?x foaf:name ?name .
    OPTIONAL { ?x foaf:mbox ?email }
    FILTER(!bound(?email))
}
```

Figure 4: SPARQL query with optional pattern.

Additional details about the SPARQL query language will be addressed in chapter 3 when the translation from SPARQL to SQL is described. However, for a full description of SPARQL and its formal semantics refer to the W3C recommendation document [20].

Chapter 2: Database Design

The design of an RDF triplestore on top of a relational database management system (RDMS) is primarily about the design of the relational physical schema of the database. The goal is to provide the query engine the appropriate access paths to the triple data so that efficient query plans can be generated. The physical design is based on assumptions about what algorithms are considered most efficient for RDF data and how the query planner may select physical operators.

Several approaches have been used to map RDF data to the relational domain. These may be roughly categorized as follows:

1. *Clustered property tables*. This is a data driven approach where the system attempts to derive a relational schema for an instance of RDF data. The idea is to identify cluster of properties that resources have a tendency to posses and group them in a single relational table. The relational schema extraction may be done online or offline, but it can become a complicated task given the flexibility of the RDF data model. This system also favors fixed workloads since it is not possible to index all possible property table column combinations. This method has been used, in part, by Jena [23].
2. *Vertical partioning on properties*. Proposed by [2], this is also a data driven approach but does not require sophisticated analysis of the dataset. RDF triples are partitioned by property and each property is assigned a table with two columns, subject and object. The main contribution of this technique is identifying that join performance is a significant factor in RDF database scalability. For typical queries where the property is fixed and joins are specified on the subject of triples, the database can use efficient merge join to

run the query. This system was implemented on a column-oriented database, but all of its characteristic features apply to row-oriented databases.

3. *'Giant' triple table*. Conceptually this is the simplest approach since it's a direct mapping of the RDF data model to a relation model. It has the advantage that the relational schema is not driven by specific RDF data, which means that, in principle; any performance characteristics of the database should apply any dataset. This is the approach used by many RDF triplestore systems including Jena [10], Sesame [6] and RDF-3X[11].

The RDF triplestore implementation described here is based on the giant triple table approach, partitioning the data by object data type. The primary design constraint is the definition of indexes and table clustering.

TRIPLE TABLE

Conceptually the definition of a triple table is very simple, as shown in Figure 5.

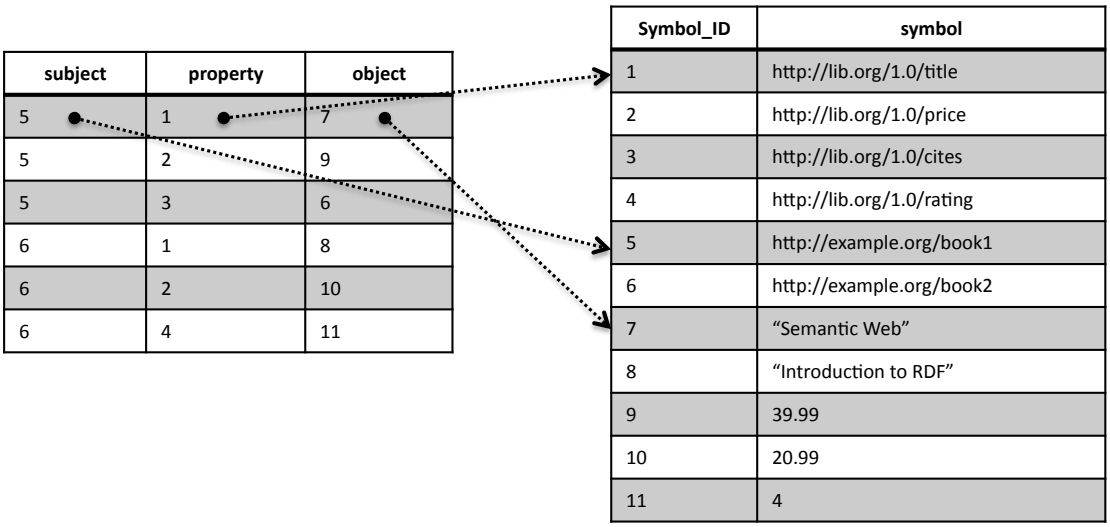
subject	property	object
http://example.org/book1	http://lib.org/1.0/title	"Semantic Web"
http://example.org/book1	http://lib.org/1.0/price	39.99
http://example.org/book1	http://lib.org/1.0/cites	http://example.org/book2
http://example.org/book2	http://lib.org/1.0/title	"Introduction to RDF"
http://example.org/book2	http://lib.org/1.0/price	20.99
http://example.org/book2	http://lib.org/1.0/rating	4

Figure 5. Triple table

However this table definition is problematic in several ways. First, it is very likely that many URLs, particularly for properties, will be used many times, which suggests the use of some form of compression. Second, join processing on string may

not be as efficient as using integers and in some systems may force the query planner to only use nested-loop joins instead of more efficient merge or hash joins. Finally, all object values must be serialized to string data, which make the evaluation of filter conditions difficult.

To address the first two problems it is common to employ a symbol dictionary that maps strings to unique identifiers. The triple table is only required to hold integers that refer to entries in the dictionary table. Evaluation of a query begins by looking up the symbol identifiers for the fixed terms in the query. Join between triples operates on the integer identifier values. In many cases a join variable may not be projected as a query result, which eliminates the need to ever lookup the string in the dictionary. Finally, mapping the identifiers to strings generates the query results. Figure 6 shows a possible implementation of a symbol dictionary.



subject	property	object
5	1	7
5	2	9
5	3	6
6	1	8
6	2	10
6	4	11

Symbol_ID	symbol
1	http://lib.org/1.0/title
2	http://lib.org/1.0/price
3	http://lib.org/1.0/cites
4	http://lib.org/1.0/rating
5	http://example.org/book1
6	http://example.org/book2
7	“Semantic Web”
8	“Introduction to RDF”
9	39.99
10	20.99
11	4

Figure 6. Triple table with dictionaries

PARTITION ON OBJECT DATATYPE

The dictionary configuration just described does not address the problem of serializing all object values to strings. The dictionary table would need to not only hold string values, but also numeric values. This suggested that an ideal configuration would use a dictionary when the object value is variable in size, like a string, and use an inline the value when it is fixed size. One approach used in systems like Jena/TDB [21] is to encode a key in the object value that the system can use to identify if the value is inlined or a dictionary key. However, this technique would complicate the evaluation of filter conditions by the SQL database engine. To avoid this complication data is partitioning on multiple tables based on object datatype, where the object column of a table is defined to a native SQL datatype that maps to the RDF data types. String and URIs values are mapped to separate dictionaries, while fixed-sized types like *ints*, *floats* and *date* are stored inlined. The string dictionary includes an additional column to store the language tag of the string literal. Figure 7 shows the final configuration of the triple store using dictionaries and typed tables.

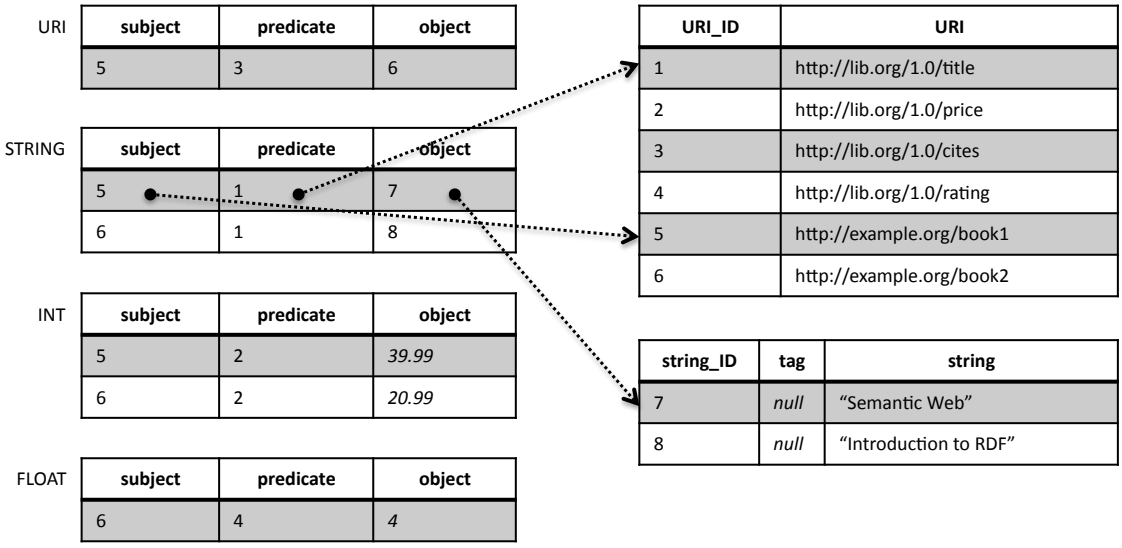


Figure 7. Table partitioning based on object data type

Dictionary tables must be indexed on both key and values to speed up lookups of strings and identifiers, respectively. Dictionary keys and values are constraint to be unique and not null, and the key of the dictionary is defined as the primary key of the table since it will serve as a foreign key to all other tables using the dictionary. Columns of the triple tables are constraint to be not null and foreign keys of the appropriate dictionary. Note that the subject and properties of all tables are URIs and are mapped to the URI dictionary table. A triple table should not contain duplicate statements, which suggests that the subject-property-object triple should form the primary key of each triple table.

To eliminate most symbol to ID dictionary look-ups it is possible to use a cryptographic hash to generate symbol IDs. The probability of collisions between symbols is practically zero (a property of cryptography hashes) and the client application can deterministically resolve symbols to ID without using the dictionary. This technique is used by Jena/TDB and explored in this implementation.

Dataset partitioning is performed when the database is loaded. If the RDF dataset includes user-defined or other non-primitive (XML Schema) datatypes, additional information must be provided to map the datatype to the appropriate SQL datatype. This mapping is necessarily domain specific, since the selection of the SQL datatype will depend on how the object value is used. For example, a phone number may be considered a string literal since arithmetic operations on phone number make little sense, but the designer may choose to map those values to integers to store them inline and avoid dictionary look-ups. While not considered in this report, it is interesting to observe that this information could be encoded in RDF Schema or OWL, allowing the system to adapt to any dataset.

The triple table partitioning just described is limited to a few simple data types, but the concept can be beneficial for object datatypes that require specialized indexing. One example is geographic data, which can be efficiently indexed for range query using R-Trees. A database capable of spatial indexing would store geographic data in a separate table and could use a R-Tree index for the object column of the table.

The partitioned triple tables can be exposed as a single triple table using a SQL view that is defined as the union of all triple tables. Because of the requirements of the SQL UNION operator, all object values must be formatted to strings. Depending on the database system used, using the triple view as the target of user queries may complicate the evaluation of filter expressions. As will be described in chapter 3, the SPARQL to SQL translation will use information from the dataset to target queries to specific tables in the database.

INDEX CONFIGURATION

The triple tables by themselves will not provide efficient access paths for the query planner to use. Without indexes the only alternative is to perform costly table scans. One advantage of the RDF data model is that a triple table can be indexed in all its column permutations, which provides an efficient access path for any query pattern.

A B-Tree index in all three columns provides the database query planner an access path that avoid a table access altogether. All information in the table is also included in the index. By creating all index combinations the database may never access the base table if it can always find an index to satisfy the query. This is essentially the same technique used by current experimental RDF databases like RDF-3X and Jena/TDB.

The use of indexes can be viewed as an extension of the vertical partitioning approach described at the beginning of the chapter. By partitioning on property, the designer optimized queries with fixed properties and joins on subjects. The advantage in processing these queries is that the property can be used to select the appropriate tables and, since the subjected are ordered, efficient merge joins can be used to process the join. In principle, creating all index combinations provided similar access paths for *any* query pattern. Fixed terms of a triple pattern define the leading columns of an index and are used to find the leaf node in the B-Tree to start the index scan. From this term, join terms in the triple pattern can be found in sorted in the index. Figure 8 illustrates this process assuming a join on two subjects and fixed properties.

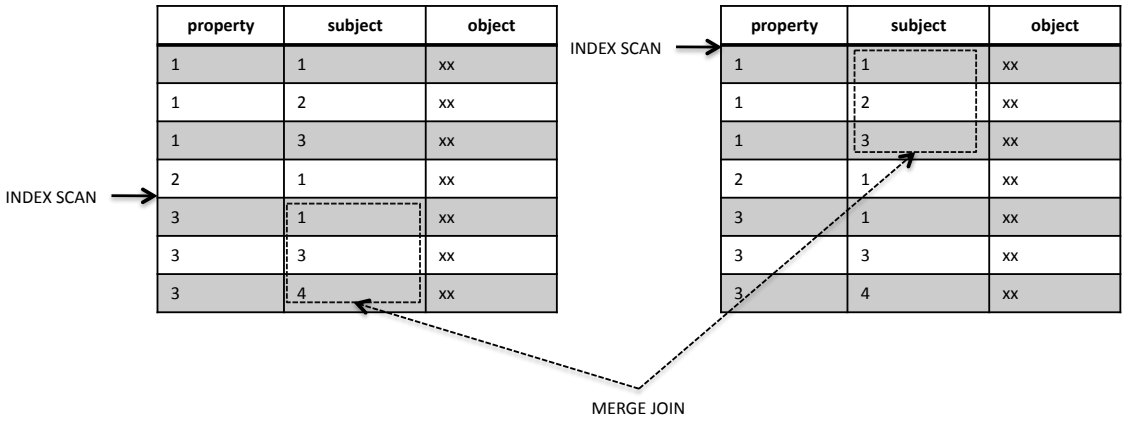


Figure 8. Expected index access for query execution

As noted in the beginning of the chapter, it is not possible to instruct the query planner to use a particular join algorithm or access path. By creating all index combinations the expectation is that the query planner will use one of these, along with other heuristics and statistics, to determine the best query plan.

CLUSTERED TABLES CONFIGURATION

The advantage of using a single table with all index combinations is that all decisions in the planning of the query are delegated to the database query optimizer. An alternative approach is to create multiple tables with a single clustering index. The criteria for table selection remains the same, to favor joins between triple terms, but now part of the query planning is done indirectly by selecting the table that is considered to be the best option based on the query join pattern. Join ordering is still left to the database to determine. Since the table selection scheme does not make use of any data statistics available to the query planner, it is possible to end up with worst performance compared to the indexed table configuration. The clustered table selection scheme is described in chapter 3.

DB2 SPECIFIC CONFIGURATION

The triplestore implementation just described was implemented on IBM DB2 9.7 Enterprise Server Edition. While all of the table schemas described apply to any relational database, modern databases are complicated systems with many configuration options. It is necessary to explore features or exploit behaviors specific to the database used to implement the RDF triplestore. This section describes some DB2 specific configuration options used.

Index with additional columns. DB2 allows the inclusion of additional columns in a unique index that will not be used as part of the key, but will only be included in the leaf nodes of the index. This feature is ideal for the symbol indexes, since the value can be included in the key index and vice versa. Including all the columns that satisfy a query in the index eliminates the need to fetch records from the table. An index scan is enough to get all the required information.

Query optimization level. By default, DB2 will attempt to find the optimal join order using dynamic programming enumeration. If the operation goes beyond an user configurable memory limit for query compilation, the query planner aborts and reverts to a greedy algorithm for join order selection. Most interesting SPARQL queries will result in SQL queries with many joins and compiling such queries will almost certainly overflow the memory limit. To avoid the cost of the fail compilation attempt it is desirable to lower the query optimization level to disable dynamic programming and used the greedy algorithm from the start.

Page size. To accommodate large strings (> 1000 characters) in the dictionary table the default page size of 4kbytes must be increased to allow indexing of the value column of the dictionary. However, the number of rows in a table to limited to 255 rows regardless of the row size. Increasing the result size will result in wasted disk space, and more importantly, inefficient use of IO buffers. Using a page size of 16kbytes, enough to index strings up to 4000 characters, will result in a page utilization of around 20% for triple tables. If this configuration is required, DB2 allows the definition of table spaces with independent page sizes. Dictionary tables and triple tables should be defined in separate table spaces. Strings larger than 8000 characters cannot be indexes by regular DB2 B-Tree indexes and should be considered special data types with special indexing requirements.

Bulk Loads. While not strictly part of the physical design of the database, bulk loading has important practical implications on the dataset sizes users are willing to store in a RDF triplestore. Transactional inserts are managed using 'instead of' triggers on triple views. Clients insert triples in their original form and the trigger map string symbols to the appropriate IDs, inserting new symbols in the dictionary when appropriate. This setup is convenient low rate updates on a running database, but the

amount of work per triple insert is too high for bulk loading. The triple insert with this configuration was measured at around 100 triples/second, which would result in a load time of 23 days for a dataset of 200 million triples, a relatively small dataset given today's scalability requirements into billions of triples.

To improve bulk load performance the dataset is preprocessed to build the symbol dictionaries, partition the dataset by object data type and join the triple symbols with the dictionary IDs. Preprocessing is done by a custom Java application that uses merge joins on disk. The result of the preprocessing step is one file per triple table laid out for fast loading using DB2's LOAD utility. Once data is loaded the database can then index and add the appropriate referential constraints to all triple tables.

Chapter 3: SPARQL to SQL Translation

SPARQL provides the user a high-level mechanism to get information from the RDF graph without dealing with the implementation details of the RDF triplestore. In the case of triplestores implemented on top of relational database management systems (RDMS), the interlaying storage system already provides a similar high-level mechanism in the form of SQL. However, in the context of RDF data, the physical schema of the database required to build SQL queries is an implementation detail that should not be directly exposed to users. The physical schema of the database will likely be designed for efficient querying, and may not necessarily represent the RDF data model. As a consequence, the physical organization is both subject to change and inadequate to reason about RDF queries. For these reasons the triplestore should provide a SPARQL interface and translate RDF queries to SQL.

Many RDF storage systems that use relational databases limit the role of the database as a persistent and reliable storage mechanism. Complex SPARQL queries may be executed by issuing simpler SQL queries to the database to fetch triples, which are then filtered or joined outside the database. In contrast, for the triplestore system described in this report the intention is to offload all query operations to the database. This requirement imposes two high-level design principles for the SPARQL to SQL translation process:

1. *Coverage*. The translation should cover a reasonable subset of the features provided by the SPARQL query language.
2. *Fast and Naïve*. The translation should not attempt complex query optimizations. Any advance optimization techniques are delegated to the query optimizer of the RDMS.

Regarding design principle 1, the reasonable subset is defined in terms of what are typical queries used in benchmarks and where the semantic mismatch between SPARQL and SQL does not make the translation too complex. Some limitations of the translation algorithm implemented are described later in this chapter.

The translation process follows a classic compiler structure: a parser converts the SPARQL query text to an abstract syntax tree. The syntax tree is then used to build an intermediate representation of the query. Finally, the intermediate representation is translated to SQL. The first two phases of this process are handled using the open-source Jena ARQ framework [4], which converts the SPARQL query text to the SPARQL algebra described in the specification document [20]. Figure 10 shows a text representation of the SPARQL algebra for the SPARQL query shown in Figure 9.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name ?email
WHERE {
    ?x rdf:type foaf:Person .
    ?x foaf:name ?name .
    OPTIONAL { ?x foaf:mbox ?email }
}
```

Figure 9: SPARQL query.

```
(project (?name ?email)
  (leftjoin
    (bgp
      (triple ?x <rdf:type> <foaf:Person>)
      (triple ?x <foaf:name> ?name)
    )
    (bgp (triple ?x <foaf:mbox> ?email))))
```

Figure 10: SPARQL query algebra.

Translating a full SPARQL query to SQL is reduced to traversing the algebra tree and translating each algebra element separately. A basic graph pattern (BGP) is considered a leaf of the tree and processed as a single unit. In other words, individual triples nodes have no direct translation to SQL, only the combination of triple patterns can generate a meaningful SQL query. Translated BGP SQL queries are nested in other SQL queries to implement other operations of the SPARQL algebra, including OPTIONAL and UNION.

BASIC GRAPH PATTERNS

A Basic Graph Pattern, or BGP, is the simplest SPARQL query, where it only includes triple patterns and variables joining nodes of those triple patterns. Figure 11 show a SPARQL query to find the 'name' of all RDF items of type 'Person'.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE {
    ?x rdf:type foaf:Person .
    ?x foaf:name ?name . }
```

Figure 11: SPARQL basic graph pattern.

The first step of the translation to SQL assumes the database includes a single triple view with the following columns:

subject	subject_id	property	property_id	object	object_id

Figure 12: Triple view assumed by query translation.

The columns *subject*, *property* and *object* contain the value of the subject, property and object of a triple statement, respectively. The *subject_id*, *property_id* and *object_id* contain the dictionary ID associated with its corresponding value, if the type of the value uses a dictionary. The SQL query is later refined to take into account the datatypes of the objects and target the real tables of the database.

Each triple in the BGP is assigned a unique identifier from a counter starting from 1. The identifier is used to alias the database triple table for each triple. The group of all aliased table references form the FROM clause of the SQL query. The sample query includes two triple pattern, which result in two table references:

```
SELECT ...
FROM triples t1, triples t2
WHERE ...
```

Figure 13: SQL statement FROM clause.

Each variable in the BGP is tracked to resolve the variable name to the appropriate column of the database triple table. As shown in Figure 14, the projected variables in the SQL query are only determined from the projected variables of the SPARQL query. As will be described later in this chapter, a BGP query may also project variables requested by other operators. If projected variable is used in a join (shared

among several triple patterns), the variable is resolved to the first triple in the BGP that references the variable.

```
SELECT t2.obj AS name
FROM triples t1, triples t2
WHERE ...
```

Figure 14: SQL statement SELECT clause.

The WHERE clause encodes the fixed triples nodes and join variables found in the SPARQL graph pattern. Since the result of the query must satisfy all the constraints found in the BGP, the SQL query predicate will be the conjunction of all the fixed node and join variable constraints. Fixed nodes are translated as equalities between the value of the node specified in the query and the appropriate *value* column of the triple view. Join constraints are translated as equalities between the *IDs* of two triples. The full translation is show in Figure 15.

```
SELECT t2.obj AS name
FROM triples t1, triples t2
WHERE t1.prp = 'rdf:type' AND
      t1.obj = 'foaf:Person' AND
      t2.prp = 'foaf:mbox' AND
      t1.sub_id = t2._sub_id
```

Figure 15: Full SQL statement from SPARQL query.

It is important to highlight that the joins occur between IDs and not the values. In many cases, as is the case with the sample query, join variables are not projected as a query result. The actual value of the node is irrelevant to satisfy the query, only equality between nodes. If the join involves a datatype that uses a dictionary, which is always the

case when joining on the subject of a triple, the query engine does not need to look up the value in the dictionary to satisfy the query.

An alternative translation used by other systems, including sparql2sql [7] and Jena/SDB [18], where each triple is translated to a SQL select and the query is the join of all these selects, as shown in Figure 16.

```
SELECT t2.obj AS name
FROM
  (SELECT * FROM triples WHERE prp = 'rdf:type' AND
                               obj = 'foaf:Person') AS t1
INNER JOIN
  (SELECT * FROM triples WHERE prp = 'foaf:mbox') AS t2
ON t1.sub_id = t2._sub_id
```

Figure 16: Alternative SQL query using explicit joins.

The SELECT-FROM-WHERE approached is favored in this implementation for the following reasons:

1. The SELECT-FROM-WHERE query is more declarative and does not suggest any particular join order, which is delegated to the query optimizer to determine. A good query optimizer will most likely generate the same access plan for both queries, but this approach highlights the principle of deferring as much as possible to the database optimizer.
2. The BGP translation does not need to handle the special case when there are no join variables between triple patterns, which should be translated as a cross join
3. No need to track the variable scope. Using explicit joins, the join predicate can only refer to variables that have introduced by a previous triple pattern. Some SPARQL queries may force a reordering of triples to satisfy this requirement.

4. Filters are easier to translate since they are just added as additional constraints of the where clause. Filters are described in more detail later in this chapter.

Regarding point 2, it is common to find SPARQL queries where the join node between two triples is not defined explicitly using a variable, and instead use the same fixed node on multiple triples, as shown in Figure 17 with its corresponding translation Figure 18.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.org>

SELECT ?label ?comment
WHERE {
   ex:item1 rdfs:label ?label .
   ex:item1 rdfs:comment ?comment }
```

Figure 17: Query without explicit join variable.

Any translation using explicit joins will find that the set of variables shared between the two triple patterns is empty. Since the inner join predicate cannot be empty, the translation must either use a cross join, or find common fixed nodes to generate a join predicate. SPARQL allows queries where the triple patterns do not share any variables or fixed nodes. In this case using a cross join is the only option. In any case, the explicit join requires additional complexity to identify and handle these cases. The SELECT-FROM-WHERE translation handles all these cases automatically.

```
SELECT t1.obj AS label,
       t2.obj AS comment
FROM triples t1,
     triples t2
WHERE t1.sub = 'http://example.orgitem1' AND
      t1.prp = 'http://www.w3.org/2000/01/rdf-schema#label' AND
      t2.sub = 'http://example.orgitem1' AND
      t2.prp = 'http://www.w3.org/2000/01/rdf-schema#comment'
```

Figure 18: SQL translation of implicit join terms.

UNIONS

SPARQL unions are used to match pattern alternatives. The UNION operator maps directly to the SQL UNION operator. The only requirement is that the projected variables (columns) in the BGPs of the union match. How this condition is met for UNIONs illustrates how projected variables are handled in general. Project variable requests are propagated top-down. The top-level projection is driven by the variable list SELECT clause of the SPARQL query. The project operator requests what variables the SQL query should project. In this case, where SQL query is the UNION operator, the translation will add those variables its SQL SELECT clause and propagate the project request to its operand sub-queries. The request will ultimately be received by a BGP, which will resolve the variable name to the appropriate triple table column. If a variable cannot be resolved it is projected as NULL, which matches the expected semantics of the UNION operator.

The UNION operation is translated as a SQL union of two sub-queries. The sub-queries can be arbitrarily complex, since the translation occurs bottom-up, starting from BGP translations. Any translated SQL as a result of a SPARQL operator can be used as a sub-query for another SQL operator, including the UNION operator itself. To illustrate this process, consider the simple query in Figure 19 and its translation in Figure 20.

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>
SELECT ?x ?y
WHERE  {
  { ?book dc10:title ?x } UNION { ?book dc11:title  ?y }
}
```

Figure 19: SPARQL UNION operator.

```
SELECT x, y
FROM
   (SELECT t1.obj AS x, null AS y
   FROM triples_all_pos t1
   WHERE t1.prp = 'http://purl.org/dc/elements/1.0/title')
UNION
   (SELECT null AS x, t2.obj AS y
   FROM triples_str_tag_pos t2
   WHERE t2.prp = 'http://purl.org/dc/elements/1.1/title')
```

Figure 20: SQL translation of UNION operator.

Each BGP is translated in isolation and then combined as sub-queries using the UNION operator. In this specific example, taken from the SPARQL specification, the set of variables of the two BGPs is disjoint. To build a valid SQL UNION operator each BGP query must project the same variables, which is accomplished by projecting NULL values for unknown variables.

OPTIONALS

The SPARQL optional operator presents the most challenges in the translation of SPARQL to SQL, especially when using filters, multiple or nested optionals. The translation scheme shown in here only considers the subset of *well-designed* queries as

defined in [15]. This well-defined condition and other restrictions are described later in this chapter. With these restrictions the SPARQL OPTIONAL operator maps to the SQL LEFT OUTER JOIN operator.

The translation of the OPTIONAL operator is similar to the union operator. Each operand of LEFT OUTER JOIN operator is treated as an opaque sub-query. The join predicate is built by finding the intersection of the sets of variables from both sub-queries. Even though certain SPARQL queries may result in an empty set of variables, this condition is not allowed by this translation. Such queries will be rejected. The set of join variables is used to project those variables in each sub-query. Join variable projections differ from the projections used for unions in that joins project the IDs of triple nodes, not their values.

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>
SELECT ?x ?y
WHERE  {
  ?book dc10:title ?x OPTIONAL { ?book dc11:title ?y }
}
```

Figure 21: SPARQL OPTIONAL operator.

Figure 21 shows a simple query involving an optional. The query will return all books with a title specified using the title property version 1.0 and, if available, also return its title as specified by the title property version 1.1. As was the case with unions , each BGP is translated in isolation and the optional is translated to a SQL LEFT OUTER JOIN. The *?book* join variable provides the join condition. The translation result is shown in Figure 22.

```
SELECT bgp1.x AS x, bgp2.y AS y
FROM
 (SELECT t1.obj AS x, t1.sub_id AS book_id
 FROM triples t1
 WHERE t1.prp = 'http://purl.org/dc/elements/1.0/title') AS bgp1
LEFT OUTER JOIN
 (SELECT t2.obj AS y, t2.sub_id AS book_id
 FROM triples t2
 WHERE t2.prp = 'http://purl.org/dc/elements/1.1/title') AS bgp2
ON bgp1.book_id = bgp2.book_id
```

Figure 22: SQL translation of OPTIONAL using LEFT OUTER JOIN.

FILTERS

Filters provide a mechanism to provide query constraints in addition to triple patterns and join variables. Translating filter expressions within a BGP is straightforward. The expression is first translated to SQL mapping XQuery operators to SQL operators. Variable names are resolved to value columns of the triple table. The translated expression is added as an additional constraint of the WHERE clause of the SQL statement.

Figure 23 and Figure 24 show a SPARQL query and its translation to SQL, respectively. Note that variables in filter expression must resolve to the value columns, not the identifiers.

```
PREFIX dc:  <http://purl.org/dc/elements/1.0/>
PREFIX ex:  <http://example.org/>
SELECT ?title ?price
WHERE  {
  ?book dc:title ?x
  ?book ex:price ?price
   FILTER(?price < 20)
}
```

Figure 23: SPARQL FILTER operator.

```
SELECT t1.obj AS title,
       t2.obj AS price
FROM triples t1, triples t2
WHERE t1.prp = 'http://purl.org/dc/elements/1.0/title' AND
      t2.prp = 'http://example.org/price' AND
      t1.sub_id = t2.sub_id AND
      (t2.obj < 20)
```

Figure 24: SQL translation of FILTER operator.

An intuitive way to translate filter expressions inside an OPTIONAL is to associate the expression with the BGP inside the OPTIONAL. In this case, the filter expression would be part of the WHERE clause of the right sub-query. An equivalent translation is to include the filter expression as part of the LEFT OUTER JOIN predicate. This second alternative has the advantage that variables in the filter expression can refer to variables in the BGP containing the OPTIONAL, even if that variable is not contained by the OPTIONAL. This case is illustrated by the query in Figure 25 and its translation in Figure 26.

```
PREFIX dc:  <http://purl.org/dc/elements/1.0/>
PREFIX ex:  <http://example.org/>
SELECT ?title ?price
WHERE  {
  ?book dc:title ?title .
  OPTIONAL { ?book ex:price ?price
  FILTER(?title = 'xyz') }
}
```

Figure 25: SPARQL query algebra.

```
SELECT bgp1.title AS title,
       bgp2.price AS price
FROM
   (SELECT t1.obj AS title,
           t1.sub_id AS book_id
   FROM triples t1
   WHERE t1.prp = 'dc:title') AS bgp1
LEFT OUTER JOIN
   (SELECT t2.obj AS price,
           t2.sub_id AS book_id
   FROM triples t2
   WHERE t2.prp = ‘ex:price') AS bgp2
ON bgp1.book_id = bgp2.book_id AND
   (bgp1.title = 'xyz')
```

Figure 26: SPARQL query algebra.

JOINS

SPARQL queries can contain multiple basic graphs patterns. This is feature is not commonly used in benchmarks queries, but it is easily translated as an INNER JOIN between sub-queries. If the graph patterns do not share any variable, a CROSS JOIN should be used instead. Filter within groups are translated with the associated BGP.

DATATYPE TABLE SELECTION

The translation algorithm so far has targeted a single triple table in the database. For the query to be valid each table reference must be resolved to a specific object datatype table. Some queries may provide enough information to resolve the table datatypes using type inference. Consider a likely analysis of the query in Figure 27: variable *?x* is used as a subject, so it must be a resource, which means the first triple pattern should resolve to the triple table with URI objects. Variable *?y* is used in an expression involving an integer, so *?y* must be an integer as well for the expression to be well-typed. Since *?y* is used as the object of the second triple pattern the table can be resolve to the integer table.

```
SELECT ?x ?y
WHERE  {
  ex:sub ex:p ?x .
  ?x ex:q ?y FILTER ( ?y > 10 )
}
```

Figure 27: Infer table data type from query.

However, in many cases type information is not available. The most basic query is to request the value of a set of properties for a resource (subject). This query would involve multiple triple patterns with fixed properties, joined by a single subject variable and free object variables. There is no way to determine what object datatype will satisfy the query since any type is allowed. Without any additional information there are two ways to check all triples across tables that satisfy the query, both of which have undesirable properties in terms of efficiency and complexity

1. Use a triple that is defined as the union of all triple tables. All object values are formatted as strings to satisfy the SQL UNION operator requirements

2. Generate a SQL query for all permutations of table datatypes and triple patterns. The final result is the union of all queries.

This problem can be resolved for most queries if some information about the dataset is included in the translation process. While RDF allows complete flexibility in terms of the datatypes a property may refer to, most real-world data will have some structure that limits the range for a property. For example, a *name* property will likely point to a string literal, while an *age* property will point to a numeric value. This information is easy to extract from the dataset even no RDF Schema or OWL metadata is available.

During the database load process, the system builds a map of properties to object datatype. With the information the resolution of table types during the translation process is done as follows:

1. If the property term of a triple pattern is fixed, lookup the datatype for the property and use the database table associated with that datatype.
2. If the object term of a triple pattern is fixed, use its datatype information to select the appropriate database table.
3. If both property and object terms are fixed, ensure that the datatype information from steps 1 and 2 matches. Otherwise the query result is empty.
4. If both property and object terms are variable, use a triple view defined as the union of all triple tables.

The triple view union of all triple tables is still required, but its use is limited to a very specific triple pattern.

This table selection scheme is clearly effective for datasets where each property refers to a single value datatype, as is the case in many real-world and benchmark datasets. For properties that refer to multiple datatypes the system could automatically

create a view defined as the union of these datatypes. There are certainly pathological datasets that cannot be handled efficiently, but they are considered out of the scope of this report since their efficient storage may require completely different techniques.

If using the multiple indexes database configuration, datatype information is enough to generate a valid SQL query. The query optimizer will generate the access plan using its internal statistics and available indexes.

CLUSTERED TABLE SELECTION

When using the explicit clustered table configuration there is no way to tell the database query optimizer that the tables refer to the same data and can be used interchangeably. This is accomplished by the use of indexes on the triple tables. Using clustered table is then a way to force the database to use a given index. Cluster selection is added to the table after the table datatype has been resolved. In accordance to the design principles mentioned in the beginning of the chapter, cluster selection is done as simple as possible, using only information about fixed triple terms and join variables. No dataset statistics or global optimization techniques are used.

Each triple is modeled as a list of three items (subject, proprety and object) that can be reordered. The ordering in the list determines the clustering of the target table. If the list is {property, subject, object} the compiler will use the PSO clustering for that table reference. During the translation process each term is assigned a priority. When a full BGP is read, the list is sorted according to their priority. The problem is then reduced to assigning the appropriate priority to each term.

The desired clustering is determined by the fixed terms and joined variables. Consequently free variables are the lowest priority and will move to the right of the cluster ordering. Fixed terms should be used to locate the group of clustered rows that

share the same property, so they are given the highest priority. Joined variables will be placed in between fixed terms to the left and free variables to the right. Ties between terms of the same type (fixed, joined or free) are broken by assuming that object terms are more selective than subject terms, and subject terms are more selective that property. The preference is to move more selective terms to the left in order to reduce the number of rows to scan. As a consequence objects are given the highest priority and properties the lowest priority.

To encode these ordering constraints term priority is defined using a high digit that encodes the term type priority (fixed, join or free) and the term position in the triple (subject, property or object). Figure 28 illustrates the clustered table selection using a simple example query.

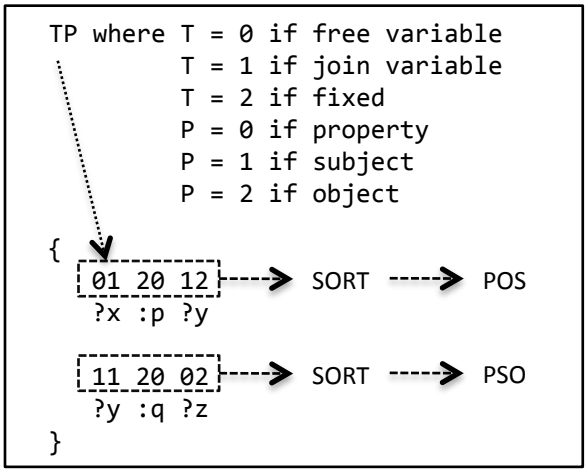


Figure 28: Clustered table selection based on term priority.

The implicit assumption made by this clustered table selection scheme is that join between triple tables will dominate the execution time of the query. The determination of join order is left to the database query optimizer.

SPARLQ TO SQL TRANSLATION RESTRICTIONS

Many features of the SPARQL language are left out of the translation process just described. Named graphs (GRAPH operator), projection expressions, ASK and CONSTRUCT query types, and most XQuery functions available in filter expressions, to name a few, were considered out of scope for this implementation and will not be described in detail. Instead, this section enumerates SPARQL queries that highlight a semantic mismatch between SPARQL and SQL, or SPARQL featuresthat someone may expect to work given the subset of features implemented by the current compiler.

Optionals with no explicit join variable

It is not uncommon to implicit define a join condition using fixed node values. For example, consider the query in Figure 29.

```
PREFIX ex:  <http://example.org/>
SELECT *
WHERE  {
  ex:sub ex:p ?w .
  OPTIONAL { ex:sub ex:q ?y }
}
```

Figure 29: OPTIONAL without explicit join variable.

The subject node on both graphs patterns is fixed to the value *ex:sub*, which constitutes a valid join condition for the join operation. However, the implicit join condition is not detected by the SPARQL to SQL translation and the query is rejected. Detecting such condition would require inspecting the fixed nodes of the graph patterns operands of the OPTINAL operator, detecting at least one join condition and promoting it to the ON clause of the SQL LEFT OUTER JOIN operator. Note that queries such as the

one used in the example can be rewritten to an equivalent form that uses explicit join variables, as shown in Figure 30.

```
PREFIX ex:  <http://example.org/>
SELECT *
WHERE  {
  FILTER(?s = ex:sub)
  ?s ex:p ?w .
  OPTIONAL { ?s ex:q ?y }
}
```

Figure 30: Equivalent query with explicit join variable.

Implicit join conditions only affect optional graph patterns. The translation of a basic graph pattern automatically handles implicit joins by including each triple pattern as a table reference in the FROM clause and fixed node conditions in the WHERE clause of the SQL statement.

Nested Optionals

SPARQL variable scoping allows a variable introduced in a graph pattern to be referenced by nested optional patterns, even if the variable is not referenced by all the patterns in between. Figure 31 shows an example of such queries.

```
PREFIX ex:  <http://example.org/>
SELECT *
WHERE  {
  ?x ex:p ?w .
  OPTIONAL {
      ?y ex:q ?y .
      OPTIONAL { ?x ex:r ?z }
} }
```

Figure 31: Nested optionals without connecting join variables.

These queries are rejected for the same reason optional without explicit join variables are rejected: it is not possible to build the LEFT OUTER JOIN predicate. This restriction matches the *well-designed* criteria defined in [15], where join variables present between nested optional graphs patterns must also appear in all optional patterns in between. With this restriction SPARQL queries have well defined properties in terms of associativity and commutativity of optional operators. Given that this particular query pattern has been found problematic in the definition of SPARQL semantics it is not considered a severe limitation for this implementation.

Chapter 4: Benchmark Results

SYSTEMS UNDER TEST

The triplestores used in the evaluation were selected based on ease of use, popularity in the Semantic Web community or due to an interesting implementation. The two systems used are briefly described below:

Jena/TDB [21]. Jena is an open-source Semantic Web framework developed by HP Labs with support for RDF, SPARQL, RDFS and OWL inference. RDF triplestores can be instantiated over most relational databases (SDB) or over a custom native RDF store (TDB). The evaluation presented here used the native version of the RDF store, Jena/TDB, which defines three B+Tree triple indexes on SPO, POS and OSP triples. The triple table is not materialized since the indexes include all necessary information about the dataset. Triples are formed from integer IDs that map to symbols in a dictionary. The dictionary is a table is indexes by a single B+Trees that map symbol to ID. The converse ID to symbol mapping is done using a sequential scan of the symbol table. Node IDs are generated from the md5 hash of the symbol. Fixed-sized values like intergers and floats are inlined in the B+Tree triple indexes. The bit pattern of the index value is used to encode whether the value is an entry in the dictionary or an inlined value.

Sesame OpenRDF [13]. Sesame is also a popular Semantic Web framework with RDFS and OWL inference capabilities, and support for multiple backend store options. For this evaluation two storage options were used: native RDF on disk and relational database store using Postgresql 9.0.3. The default configuration for the native store uses two B+Tree indexes over SPOC and POSC, where C refers to an additional context term that identifies the dataset that the triple belongs to. The database backend configuration is also based on a single triple table without the use of a dictionary.

DATASET

The *Berlin SPARQL Benchmark V3.0* [5] was used for the evaluation of the triplestores. The benchmark is based on a synthetic dataset designed around an e-commerce use case with products from different vendors and user reviews. The benchmarks distribution includes a dataset generator and testdriver with queries for various use cases. Typical dataset sizes used for triplestore evaluation are 1M, 25M, 100M and 200M triples (M = million). The test results presented were generated using the 'explore' use case query mix. The specific SPARQL queries used for evaluation are listed in the appendix.

LOAD RESULTS

Load times are listed in *hours:minutes:seconds* format[3]. DB2 load time includes table loading, index creation, table reorganization and gathering of statistics. Dataset preprocessing is listed separately since it is required once per dataset, but should be considered part of the DB2 loading time.

	1M	25M	100M	200M
Jena/TDB	00:00:23.26	00:09:57.90	00:53:04.70	01:54:32.50
Sesame/Native	00:01:00.80	01:10:02.95	54:35:20.04	---
Sesame/DB	00:05:55.39	08:31:56.57	---	---
DB2/Preprocess	00:01:37.42	00:52:15.75	03:58:43.84	09:02:39.96
DB2/Index	00:00:57.89	00:22:53.89	02:07:41.71	07:09:01.04
DB2/Clustered	00:03:17.38	01:18:02.39	04:30:59.32	---

Table 1. Berlin SPARQL Benchmark load results

[3] Missing entries indicates that the dataset was not loaded due to time constraints.

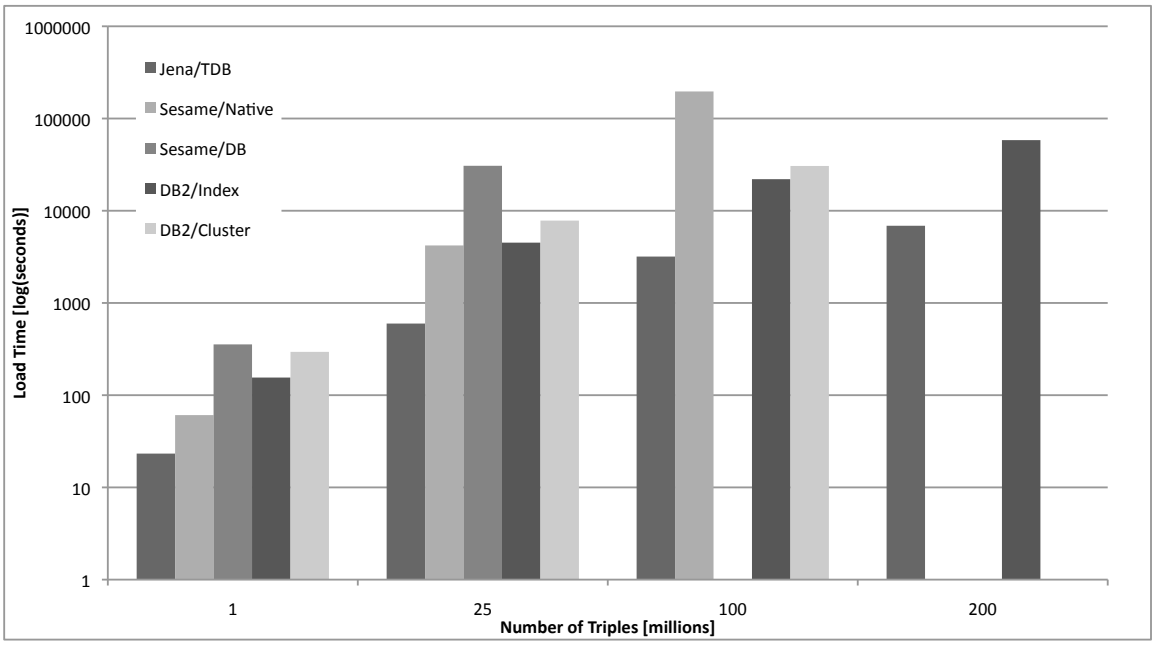


Figure 32. Berlin SPARLQ Benchmark load times

Figure 32 shows the load time for each system ($\log_{10}$ scale). DB2 results configurations include data preprocessing time. The triple tables loaded to DB2 used hashed values as symbol ID. The query processor can be configured to use symbols or hash values during SQL query generation.

QUERY RESULTS

All query results are listed as average query execution time in seconds. Four DB2 configuration were tested: Index on all 6 column combinations using symbol dictionaries (DB2/IDX); index on all 6 column combinations using hashing (DB2/IDX-HS); and clustered triple tables using symbol dictionaries (DB2/CL). Entries marked with '---' indicate that the query was not executed because the dataset was not loaded. Entries marked with '*>300*' indicate that the query took longer than 5 minutes and the test driver timed out.

	1	2	3	4	5	7	8	10	11
Jena/TDB	0.023	0.049	0.024	0.026	0.049	**0.035**	0.034	0.027	**0.038**
Sesame/Native	**0.004**	**0.006**	**0.005**	**0.005**	0.082	0.036	**0.008**	**0.006**	0.209
Sesame/DB	0.020	0.097	0.029	0.067	0.099	0.141	0.087	0.125	0.081
DB2/IDX	0.103	0.860	0.101	0.229	0.074	0.781	0.314	0.125	0.057
DB2/IDX-HS	0.048	0.305	0.026	0.060	**0.045**	0.051	0.074	0.097	0.103
DB2/CL	0.080	0.352	0.088	0.105	0.076	0.664	0.320	0.124	0.082

Table 2. Berlin SPARQL Benchmark query results (1M).

	1	2	3	4	5	7	8	10	11
Jena/TDB	0.096	0.113	0.116	0.103	**0.458**	0.300	0.222	0.183	**0.135**
Sesame/Native	**0.057**	**0.021**	**0.032**	**0.043**	1.817	**0.262**	**0.127**	**0.146**	5.128
Sesame/DB	0.065	0.148	0.116	0.146	1.111	0.412	0.607	0.425	8.553
DB2/IDX	1.370	29.80	1.874	3.497	2.167	37.53	8.442	6.553	0.498
DB2/IDX-HS	0.258	1.071	0.671	0.319	10.15	14.03	0.399	0.636	0.155
DB2/CL	0.766	6.162	1.327	0.766	1.895	37.33	10.08	6.647	0.225

Table 3. Berlin SPARQL Benchmark query results (25M).

	1	2	3	4	5	7	8	10	11
Jena/TDB	**0.255**	0.254	**0.260**	**0.268**	**1.471**	**0.828**	**0.557**	**0.431**	0.295
Sesame/Native	3.760	**0.189**	2.927	1.310	108.68	2.371	0.606	1.628	217.442
Sesame/DB	---	---	---	---	---	---	---	---	---
DB2/IDX	4.762	251.8	7.056	13.50	11.35	*>300*	59.97	83.30	0.847
DB2/IDX-HS	0.673	5.341	0.808	1.267	43.74	*>300*	1.495	0.658	**0.138**
DB2/CL	3.548	45.90	6.828	8.127	11.37	*>300*	124.7	69.73	0.222

Table 4. Berlin SPARQL Benchmark query results (100M).

	1	2	3	4	5	7	8	10	11
Jena/TDB	**0.260**	**0.245**	**0.255**	**0.302**	**18.488**	**1.036**	**0.595**	**0.450**	**0.266**
Sesame/Native	---	---	---	---	---	---	---	---	---
Sesame/DB	---	---	---	---	---	---	---	---	---
DB2/IDX[4]	---	---	---	---	---	---	---	---	---
DB2/IDX-HS[4]	---	---	---	---	---	---	---	---	---
DB2/CL	---	---	---	---	---	---	---	---	---

Table 5. Berlin SPARQL Benchmark query results (200M).

[4] Dataset was loaded but test driver did not complete after 6 hours of operation. Results for benchmark systems are shown for completeness.

DB2 QUERY ACCESS PLANS

This section will present a few access plans generated from the test queries to briefly show some of the query plan decisions made by DB2. In general, the query plans generated by SPARQL queries are complex, involving many joins. It is not possible to describe every access plan used in the evaluation.

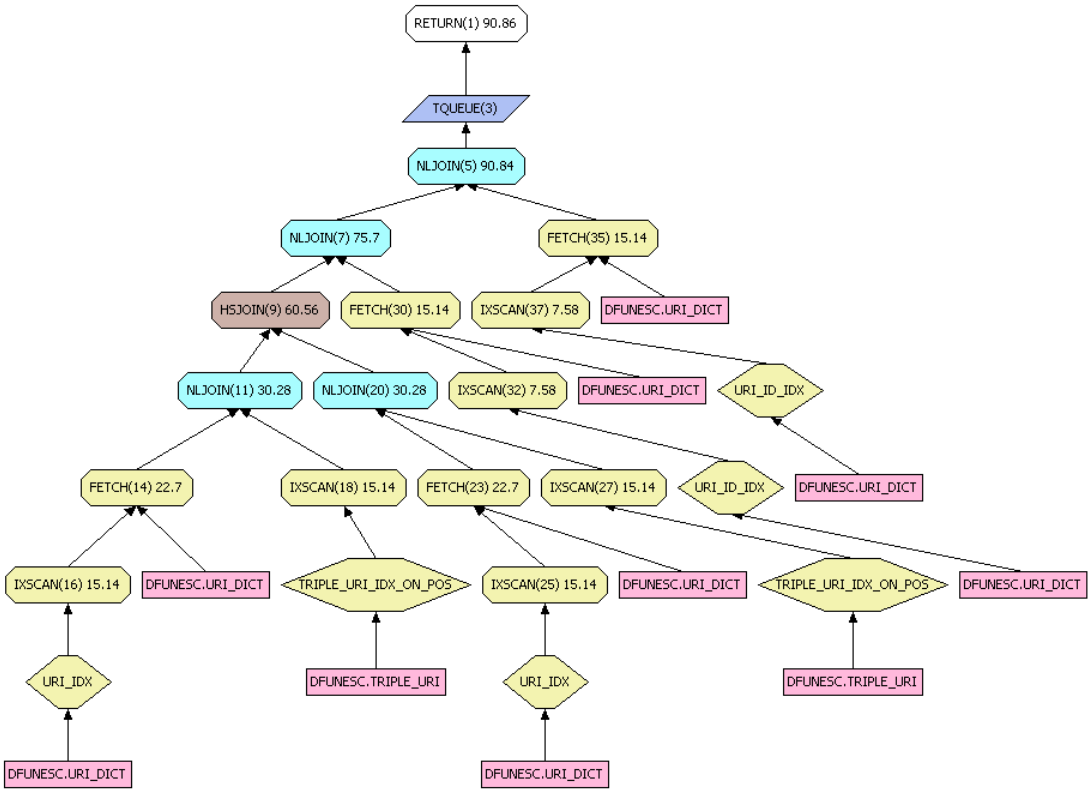


Figure 33. DB2 Access Plan, fetching from dictionary tables

The Figure above shows a typical access plan for a simple SPARQL query. To resolve resource identifiers to IDs using the dictionary, the URI index is used to find row identifiers, which are then used to fetch records from the dictionary. In general, DB2 will use a nested loop join to join the dictionary identifier with the triple table. This is

reasonable considering that the result of the dictionary look-up is guaranteed to be one record since there is a uniqueness constraint on the URI column of the dictionary. From this access plan it can also be seen that DB2 does not access the base triple table and, since the index has all triple columns, only index scans are used to join triples. As mentioned in Chapter 2, DB2 allows additional columns to be included in the leaves of an index. This option can be used by dictionary indexes to allow for index only scans when resolving resource identifiers to string and vice versa. The access plan using this option is shown in Figure 33.

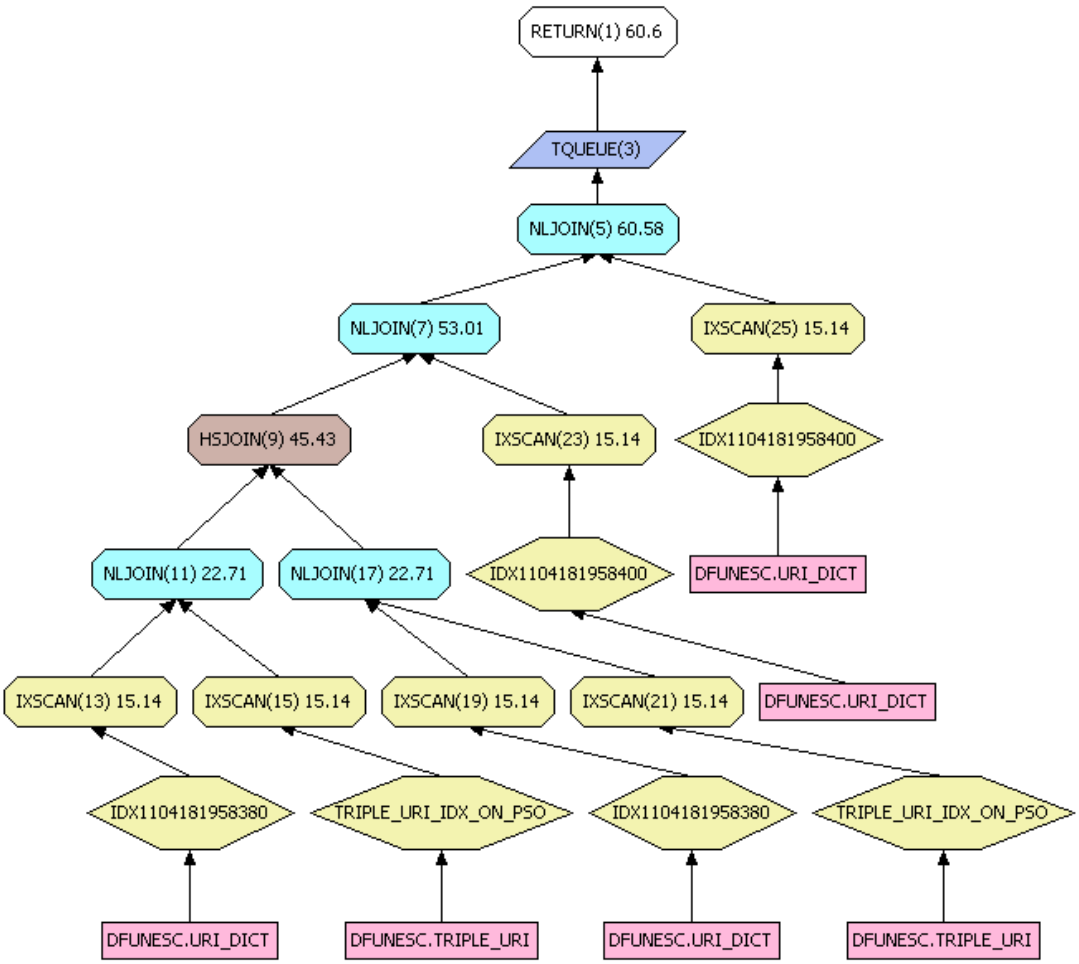


Figure 34. DB2 access plan using index INCLUDE option

The new access plan shows that only index scans are required to satisfy the query. This is basically the same technique used by many native RDF databases, where the triple table is never materialize and only index scans are used. This access plan is typical for the queries used in the benchmark. In general DB2 will use nested loop joins between triple indexes and dictionary indexes, and hash joins to join triples. For all the dataset sizes used in the benchmark DB2 will select hash joins. However for smaller datasets DB2 will use merge joins as shown in Figure 34. This is interesting since merge joins are typically used by native RDF databases to join triples, given that index scans already provide sorted join columns, but clearly DB2 query planner is using other metrics to select hash joins instead.

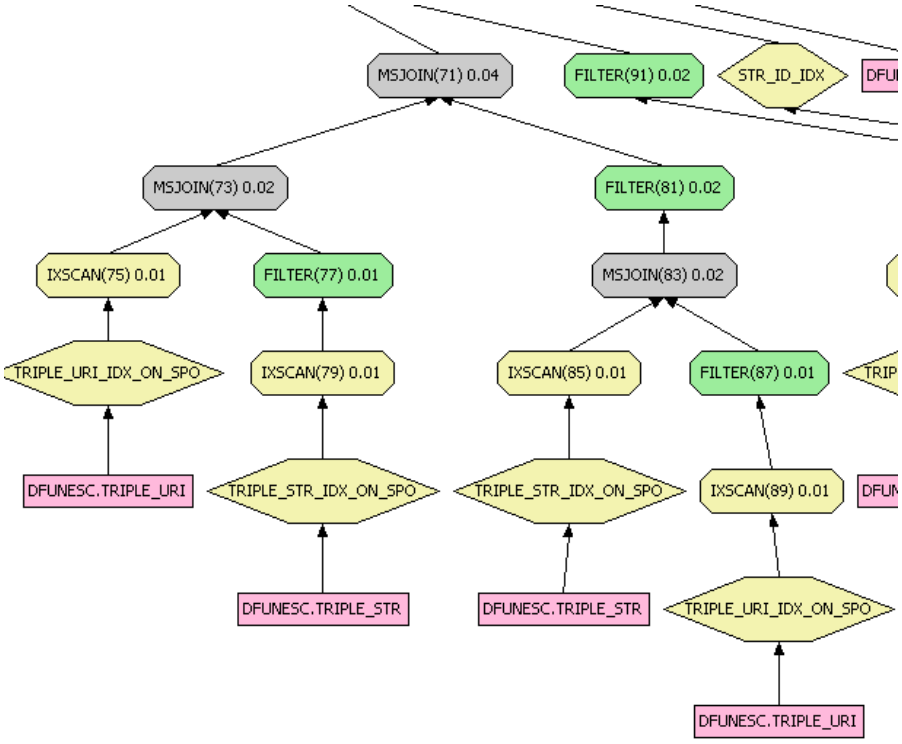


Figure 35. DB2 access plan using merge-joins.

Chapter 5: Conclusion

It is clear from the benchmark results that more investigation regarding database configuration is required to implement a scalable triplestore with good overall performance using the design described in this report. While overall query performance is worse compared to the benchmark systems, for many queries execution time was in the same order of magnitude. This is encouraging given that both these Jena and Sesame are using similar indexing data structures to the DB2 implementation. It is not unreasonable to think that expert configuration of the DB2 system could result in much better performance. A first step would be to focus on problematic queries, notably query 7, which degrades considerably as the number of triple increases. This query is notable in that uses optional and nested optional graph patterns extensively, which indicates a performance problem in the design of the triplestore for the execution left outer joins. Database configuration is also likely to be the cause of scalability issues beyond 100 million triples.

Perhaps the most interesting aspect of the evaluation is the comparison between the different DB2 triplestores configurations. Hashing had a noticeable impact on overall query performance. This performance boost is expected since hashing avoids a dictionary lookup to resolve query strings, but it was unexpected to find that the use of hashing could performance on certain queries. These effects could be explained by the fact that the SPARQL compiler does not use hashed values for filter expressions, which requires dictionary lookups. In general, a better understanding of the effects of the dictionary compression design is necessary.

Finally, it was surprising to find that clustering had a significant impact on performance when compared to the indexed configuration. As mentioned in previous

chapters, creating all six indexes on a triple table gives the query planner the option to always select index scans, and avoid fetching records from the base triple table. Using separate clustered tables with a single index allows the SPARLQ compiler to suggest that a specific index should be used. The cluster table selection algorithm used in the evaluation was naïve, using only information about term types (fixed, join variable or free variable) in a triples triple pattern. No dataset statistics or global optimization algorithms were used, and yet it is clear that restricting the access paths available to the query planner had a positive impact on query performance. Also, note that because the complexity of the queries the DB2 optimization class used in the evaluation was lowered to effectively disable dynamic programming cost-based join order selection and force the query planner to use a greedy algorithm instead. The fact that such a simple cluster selection algorithm had an impact in performance suggests that an interesting area of future investigation would be to find out what other optimization techniques could be use during SQL query generation to improve access plans for RDF query workloads.

Appendices

A. Test System Configuration

```
Windows 7 Ultimate 64-bit Service Pack 1
Intel i3 Dual Core 3.2GHz with 12GB RAM
Two 1TB 5400rpm 64MB cache hard-drives configured in RAID 1 (mirror)
Java 1.6.0_24 for Windows 64-bit
IBM DB2 Enterprise Server Edition 9.7 for Windows 64-bit (x86-64)
Jena/TDB Fuseki version 0.1.0
Sesame 2.3.2 with apache-tomcat 6.0.32
Postgresql 64-bit version 9.0.3
```

B. Berlin SPARQL Benchmark Test Queries

Berlin SPARQL Benchmark version 3.0 explore use case was used in the evaluation. Triple terms enclosed by %xx% indicate nodes that are replaced to fixed values by the benchmark test driver. Assume the following prefix header for all queries:

```
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
Q1.
SELECT DISTINCT ?product ?label
WHERE {
   ?product rdfs:label ?label .
   ?product a %ProductType% .
   ?product bsbm:productFeature %ProductFeature1% .
   ?product bsbm:productFeature %ProductFeature2% .
   ?product bsbm:productPropertyNumeric1 ?value1 .
   FILTER (?value1 > %x%)
}
ORDER BY ?label
LIMIT 10
```

```
Q2.
SELECT ?label ?comment ?producer ?productFeature ?propertyTextual1
       ?propertyTextual2 ?propertyTextual3 ?propertyNumeric1 ?propertyNumeric2
       ?propertyTextual4 ?propertyTextual5 ?propertyNumeric4
WHERE {
    FILTER(?product = %ProductXYZ%)
    ?product rdfs:label ?label .
    ?product rdfs:comment ?comment .
    ?product bsbm:producer ?p .
    ?p rdfs:label ?producer .
    ?product dc:publisher ?p .
    ?product bsbm:productFeature ?f .
    ?f rdfs:label ?productFeature .
    ?product bsbm:productPropertyTextual1 ?propertyTextual1 .
    ?product bsbm:productPropertyTextual2 ?propertyTextual2 .
    ?product bsbm:productPropertyTextual3 ?propertyTextual3 .
    ?product bsbm:productPropertyNumeric1 ?propertyNumeric1 .
    ?product bsbm:productPropertyNumeric2 ?propertyNumeric2 .
    OPTIONAL { ?product bsbm:productPropertyTextual4 ?propertyTextual4 }
    OPTIONAL { ?product bsbm:productPropertyTextual5 ?propertyTextual5 }
    OPTIONAL {?product bsbm:productPropertyNumeric4 ?propertyNumeric4 }
}

Q3.
SELECT ?product ?label
WHERE {
    ?product rdfs:label ?label .
    ?product a %ProductType% .
    ?product bsbm:productFeature %ProductFeature1% .
    ?product bsbm:productPropertyNumeric1 ?p1 .
    FILTER ( ?p1 > %x% )
    ?product bsbm:productPropertyNumeric3 ?p3 .
    FILTER (?p3 < %y% )
    OPTIONAL {
       ?product bsbm:productFeature %ProductFeature2% .
       ?product rdfs:label ?testVar }
       FILTER (!bound(?testVar))
}
ORDER BY ?label
LIMIT 10
```

```
Q4.
SELECT DISTINCT ?product ?label ?propertyTextual
WHERE {
  {
      ?product rdfs:label ?label .
      ?product rdf:type %ProductType% .
      ?product bsbm:productFeature %ProductFeature1% .
      ?product bsbm:productFeature %ProductFeature2% .
      ?product bsbm:productPropertyTextual1 ?propertyTextual .
      ?product bsbm:productPropertyNumeric1 ?p1 .
      FILTER ( ?p1 > %x% )
  } UNION {
      ?product rdfs:label ?label .
      ?product rdf:type %ProductType% .
      ?product bsbm:productFeature %ProductFeature1% .
      ?product bsbm:productFeature %ProductFeature3% .
      ?product bsbm:productPropertyTextual1 ?propertyTextual .
      ?product bsbm:productPropertyNumeric2 ?p2 .
      FILTER ( ?p2> %y% )
  }
}
ORDER BY ?label
OFFSET 5
LIMIT 10

Q5.
SELECT DISTINCT ?product ?productLabel
WHERE {
   ?product rdfs:label ?productLabel .
   FILTER (%ProductXYZ% != ?product)
   %ProductXYZ% bsbm:productFeature ?prodFeature .
   ?product bsbm:productFeature ?prodFeature .
   %ProductXYZ% bsbm:productPropertyNumeric1 ?origProperty1 .
   ?product bsbm:productPropertyNumeric1 ?simProperty1 .
   FILTER (?simProperty1 < (?origProperty1 + 120) && ?simProperty1 >
           (?origProperty1 - 120))
   %ProductXYZ% bsbm:productPropertyNumeric2 ?origProperty2 .
   ?product bsbm:productPropertyNumeric2 ?simProperty2 .
   FILTER (?simProperty2 < (?origProperty2 + 170) && ?simProperty2 >
           (?origProperty2 - 170))
}
ORDER BY ?productLabel
LIMIT 5
```

```
Q6.  Removed from version 3 of the benchmark.

Q7.
SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle ?review ?revTitle
       ?reviewer ?revName ?rating1 ?rating2
WHERE {
   FILTER(?product = %ProductXYZ%)
   ?product rdfs:label ?productLabel .
   OPTIONAL {
      ?offer bsbm:product ?product .
      ?offer bsbm:price ?price .
      ?offer bsbm:vendor ?vendor .
      ?vendor rdfs:label ?vendorTitle .
      ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
      ?offer dc:publisher ?vendor .
      ?offer bsbm:validTo ?date .
      FILTER (?date > %currentDate% )
   }
   OPTIONAL {
      ?review bsbm:reviewFor ?product .
      ?review rev:reviewer ?reviewer .
      ?reviewer foaf:name ?revName .
      ?review dc:title ?revTitle .
      OPTIONAL { ?review bsbm:rating1 ?rating1 . }
      OPTIONAL { ?review bsbm:rating2 ?rating2 . }
} }

Q8.
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?rating1
       ?rating2 ?rating3 ?rating4
WHERE {
   ?review bsbm:reviewFor %ProductXYZ% .
   ?review dc:title ?title .
   ?review rev:text ?text .
   FILTER langMatches( lang(?text), "EN" )
   ?review bsbm:reviewDate ?reviewDate .
   ?review rev:reviewer ?reviewer .
   ?reviewer foaf:name ?reviewerName .
   OPTIONAL { ?review bsbm:rating1 ?rating1 . }
   OPTIONAL { ?review bsbm:rating2 ?rating2 . }
   OPTIONAL { ?review bsbm:rating3 ?rating3 . }
   OPTIONAL { ?review bsbm:rating4 ?rating4 . } }
ORDER BY DESC(?reviewDate)
LIMIT 20
```

```
Q9. DESCRIBE SPARQL queries are not supported

Q10.
SELECT DISTINCT ?offer ?price
WHERE {
   ?offer bsbm:product %ProductXYZ% .
   ?offer bsbm:vendor ?vendor .
   ?offer dc:publisher ?vendor .
   ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#US> .
   ?offer bsbm:deliveryDays ?deliveryDays .
   FILTER (?deliveryDays <= 3)
   ?offer bsbm:price ?price .
   ?offer bsbm:validTo ?date .
   FILTER (?date > %currentDate% )
}
ORDER BY xsd:double(str(?price))
LIMIT 10

Q11.
SELECT ?property ?hasValue ?isValueOf
WHERE {
   { %OfferXYZ% ?property ?hasValue }
   UNION
   { ?isValueOf ?property %OfferXYZ% }
}

Q12. CONSTRUCT SPARQL queries are not supported
```

References

1. Abadi, D. J., Marcus, A., Madden, S. and Hollenbach, K. J. SW-Store: a vertically [artitioned DBMS for Semantic Web data management. In VLDB, 2009.
2. Abadi, D. J., Marcus, A., Madden, S. and Hollenbach, K. J. Scalable Semantic Web data management using vertical partitioning. In VLDB, 2007.
3. Antoniou, G. and Harmelen, F. A Semantic Web Primer, 2nd edition. 2008.
4. ARQ – A SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ
5. Bizer, C and Schultz, A. The Berlin SPARQL Benchmark. In IJSWIS, 2009.
6. Broekstra, J. and Kampman A. Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema. In ISWC 2002.
7. Cyganiak, R. A relational algebra for Sparql. HP-Labs Technical Report, HPL2005-170.
8. Das, S., Chong, E. I., Eadon G. and Srinivasan J. Supporting Ontology-based semantic matching in RDBMS. In VLDB, 2004.
9. Harris, S. and Gibbins, N. 3store: Efficient bulk RDF storage. 2003.
10. Jena: a Semantic Web Framework for Java. http://jena.sourceforge.net
11. Neumann, T. and Weikum, G. RDF-3X: a RISC-style Engine for RDF. In PVLDB, 2008.
12. Neumann, T. and Weikum, G. Scalable join processing on very large RDF graphs. In SIGMOD, 2009.
13. OpenRDF. http://www.openrdf.org
14. OWL 2 Web Ontology Language. http://www.w3.org/TR/owl-overview/. October 2009.
15. Perez, J., Arenas, M. and Gutierrez, C. Semantics and Complexity of SPARQL. In ISWC 2006.

16. RDF Schema 1.0. http://www.w3.org/TR/rdf-schema/. February 2004
17. Resource Description Framework. http://www.w3.org/RDF/
18. SDB – A SPARQL database for Jena. http://www.openjena.org/SDB/
19. Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N. and Manegold S. Column-Store Support for RDF Data Management: not all swans are white. In PVLDB, 2008
20. SPARLQ Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/. W3C Recommendation, January 15 2008.
21. TDB – A SPARQL Database for Jena. http://www.openjena.org/TDB/
22. Weiss, C., Karras, P. and Bernstein, A. Hexastore: Sextuple Indexing for Semantic Web Data Management. In PVLDB, 2008.
23. Wilkinson, K., Sayers, C., Kuno, H. and Reynolds, D. Efficient RDF Storage and Retrieval in Jena2. 2003.