Implementing SPARQL Support for Relational Databases and Possible Enhancements

Christian Weiske, Sören Auer
Universität Leipzig
cweiske@cweiske.de, auer@informatik.uni-leipzig.de

Abstract: In order to make the Semantic Web real we need the infrastructure to store, query and update information adhering to the RDF paradigm. Such infrastructure can be developed from scratch or benefit from developments and experiences made in other science & technology realms such as within the database domain. For querying RDF data the World Wide Web Consortium released a Working Draft for the SPARQL query language. A large portion of the Web is meanwhile driven by server-side Web applications. PHP is the scripting language most widely used for Web applications. In this paper we present our PHP implementation of the SPARQL standard directly interacting with an underlying database system. The approach is based on the rationale of pushing as much work into the RDBMS as possible in order to profit most from the query optimization techniques developed for relational database systems. The article includes an evaluation of the performance and standard compliance, surveys limitations we discovered when using SPARQL for the implementation of real-life Semantic Web applications and suggests extensions to the SPARQL standard in order to solve these obstacles.

1 Introduction

In order to make the Semantic Web real we need the infrastructure to store, query and update information adhering to the RDF paradigm. Such infrastructure can be developed from scratch or benefit from developments and experiences made in other science & technology realms such as within the database domain. Database systems, for example, have reached a very high degree of maturity with respect to data storage and querying: The SQL standard allows to formulate queries regarding almost all possible nuances of relational algebra. Database systems include sophisticated query optimizers finding optimal data access plans independent from specific query phrasings. Hence, building on top of such full-grown technologies will enable Semantic Web developers to benefit greatly from advances of database systems.

For querying RDF data the World Wide Web Consortium released a Working Draft for the SPARQL query language[PS06]. A large portion of the Web is meanwhile driven by server-side Web applications. PHP is the scripting language most widely used for Web applications[1]. In this paper we present our PHP implementation of the SPARQL standard directly interacting with an underlying database system and suggest enhancements to the

[1] http://www.tiobe.com/tpci.htm

SPARQL standard, which are motivated by real-life applications of our implementation such as OntoWiki [ADR06] and DBpedia [ABL$^+$07].

The approach is based on the rationale of pushing as much work into the RDBMS as possible in order to profit most from the query optimization techniques developed for database systems. This article covers implementation details of the new SPARQL engine, evaluates its performance and coverage of the proposed SPARQL standard, surveys limitations we discovered when using SPARQL for the implementation of semantic web applications, as well as suggests extensions to the SPARQL standard in order to solve these obstacles.

2 A SPARQL Engine for PHP

The interface between the programming language (such as PHP in our case) and the database query language (SPARQL) is an application programming interface (API). A few PHP-based open-source RDF APIs are available, and RAP (RDF API for PHP, [Biz04]) is one of the most mature one amongst them. One of the limitations of RAP was its SPARQL engine. It is built to work on any RDF model that can be loaded into memory. Using SPARQL to query a database required to load the complete database into memory and execute the SPARQL query on it. While this works well with some dozens of RDF triples, it can not be used for databases with millions of triple data - main memory is one limitation, execution time another one (code implemented in a scripting language such as PHP is about 50 times slower[2] than pure C implementations of the same code). Our goal was to create a SPARQL engine that pushes as much work as possible to the database server, using PHP only as translator between SPARQL and SQL [Ins89] and integrating seamlessly into the existing RAP infrastructure.

2.1 Database vs. Main Memory Implementation

RAP's old SparqlEngine implementation works on RDF stores whose data are loaded into main memory. Querying 100 million triples stored in a SQL database causes the whole data to be loaded into the server's RAM - often multiple times, depending on the number of triple patterns the query uses.

Other than to the old implementation, SparqlEngineDb creates SQL queries from the internal SPARQL query representation and lets the database server execute these queries. Query results are converted into the desired result format as the last step.

Unlike the old SparqlEngine implementation, our new SparqlEngineDb pushes all hard and time consuming operations into the database server. Since databases are highly optimized for selecting, filtering, ordering and joining data we can expect a significant speedup using this approach.

[2] `http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=php&lang2=gcc`

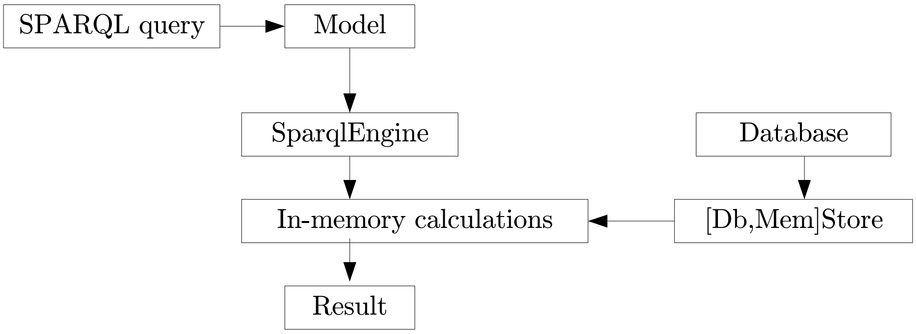

Figure 1: SparqlEngine data flow

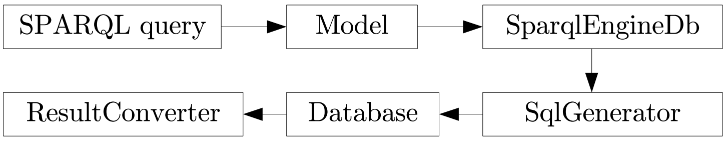

Figure 2: SparqlEngineDb data flow

Another concept introduced in SparqlEngineDb are result renderers. The old engine had one default result type, PHP arrays. A parameter was added to the query method to allow returning XML data, and the engine had an extra method `writeQueryResultAsHtmlTable` that added formatting capabilities to the model class implementation. Beside mixing of view and controller classes, the existing system was in no way extensible or customizable. Core classes needed to be changed to add new output formats.

`SparqlEngineDb` (and now also the old memory based `SparqlEngine`) support `ResultRenderer` - pluggable drivers that convert raw database results into the desired output format - be it SPARQL Query Results XML Format, [BB06], the JavaScript Object Notation [3] JSON for web applications or an own application specific renderer. The system allows great flexibility without ever needing to change internal classes.

2.2 Generating SQL

The main job SparqlEngineDb performes is generating one or several SQL queries from a SPARQL query's object representation after parsing it.

SQL queries are created specifically for RAP's database schema. This unnormalized

[3] `http://www.json.org/`

schema collects all triple data in a single "statements" table. The table structure is visualized below:

Column name	**Data type**
modelID	integer
subject	varchar
subject_is	varchar(1)
predicate	varchar
object	blob
object_is	varchar(1)
l_language	varchar
l_datatype	varchar

The SQL query generation involves the following four steps before the query is complete and can be sent to the database:

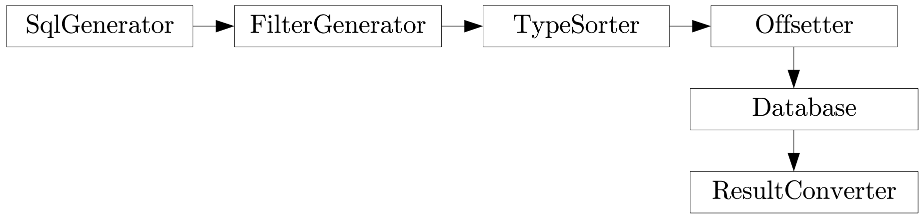

Figure 3: SQL generation data flow

After generating SQL joins for each individual triple pattern, the filter generator creates suitable SQL `WHERE` clauses from SPARQL filter expressions.

Since RAP stores RDF triples in the database in an unnormalized way, boolean `TRUE` values for example may have multiple representations in SQL: `T`, `TRUE` and `1`. The same applies to all datatypes, such as for example `xsd:dateTime` values, as the following example illustrates:
`"2004-12-31T18:01:00-05:00"^^<xsd:dateTime>` [4] and
`"2004-12-31T20:01:00-03:00"^^<xsd:dateTime>`

Both literals represent the same date and time, but are physically stored differently. Queries on such unnormalized data are more complex since all possible different value combinations have to be checked.

In RAP, literal values of all data types are stored as BLOBs in the object column of the statements table. Using a simple `ORDER BY object` is not possible, because for example numerical values such as `10` and `2` would be returned in the wrong order. Hence, type casting is necessary, but quite hard and cumbersome to implement correctly considering the unnormalized representation of the data.

[4] We use the namespaceprefix `xsd` for http://www.w3.org/2001/XMLSchema#

Several actions needed to be undertaken in order to deal with these problems. Certain equations in SPARQL's `FILTER` clauses are expanded to several `WHERE` clauses in SQL and concatenated with `OR`. SPARQL's `ORDER BY` clauses need special attention since different data types need to be casted differently. Here, we determine first a list of all data types in the result set and execute one query for each one of the found data types. The final result set is constructed by combining the individual results by means of the SQL `UNION` operator.

Sending multiple queries to the SQL server requires custom handling of `LIMIT` and `OFFSET` constrains. Hence, the Offsetter determines the number of result rows for each single query and calculates which queries to execute at all, and which parts of it to return.

All these inconveniences make `SparqlEngineDb` slower than it would be with a normalized database layout. Future versions of RAP should alternatively provide support for storing triples in normalized form.

3 Evaluation

Our SPARQL implementation was evaluated along three dimensions: we compared its performance with other state of the art engines, evaluate its compliance with the SPARQL standard and review some examples of its usage in real applications.

3.1 Performance Comparison

We used the Lehigh University Benchmark tool suite [GPH05] to generate RDF data for the query speed evaluation. Queries have been tested on databases with sizes ranging from five to 200.000 triples. The competitors were RAP's old memory based *SparqlEngine*, our new *SparqlEngineDb*, *Jena SDB*, *ARC*, *Redland/librdf* and *Virtuoso*.

Two groups of SPARQL queries have been selected for evaluation. Queries of first group focus each on a different SPARQL feature and were designed to be independent of RDF data structure and types, i.e. can be used with any concrete dataset. The second group contains three queries which are specific to the Lehigh Benchmark RDF data and aim at evaluating the runtime of complex queries.

Using the queries, we seek answers to the following questions:

1. How does the implementation behave on small vs. large databases?
2. Which features do the individual implementations support?
3. Are filter clauses optimized in some way?
4. How does the engine behave on complex queries?
5. Which implementation is the fastest?

We found out that some engines do not fully support the SPARQL specification and did not deliver results for some queries. The results of the performance comparison show that Virtuoso is by far the fastest SPARQL implementation available. Our new SparqlEngineDb implementation performs second best on average in all tests with database sizes from 5 to 200.000 triples. Only SparqlEngineDb and Jena were able to provide results for all tested queries. Redland did not execute the `UNION` query while ARC failed on nearly half of the tests (cf. Figure 4).

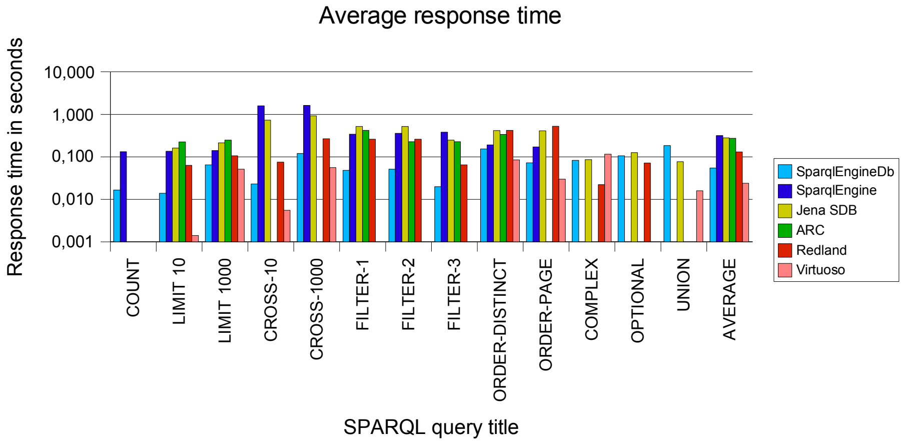

Figure 4: Average response times for different queries.

Detailled information about tested queries and library versions can be found at `http://aksw.org/FIXME`.

Based on the results of the performance evaluation, we see the following improvement areas for SparqlEngineDb:

- Multiple sort variables cause the engine to return results multiple times slower than using just a single variable to sort results.
- Optional patterns are fast on small databases, but get significantly slower if the size increases. The increase of computing time is much slower or even constant with other engines.
- Limit handling in UNION queries needs to be improved. Currently, limits are applied only after the full result set has been calculated.

3.2 SPARQL Standard Compliance

A great amount of work at SparqlEngineDb has been done in order to comply with the SPARQL specification. RAP contains a comprehensive unit test suite that is used to ensure all aspects of the package are working correctly. The SPARQL test suite in RAP includes the test cases created by the RDF Data Access Working Group[5] as well as tests created by the programmers, reproducing former bugs. The total amount of SPARQL test cases in RAP is 151; SparqlEngineDb passes 148 of them.

SparqlEngineDb supports all SPARQL features except named graphs and sorting dates with different timezones. Named graph support could not be implemented due to missing support for named graphs in RAP's current database layout. Sorting time data works correctly on dates in the same time zone only. The reason for this is the lack of time zone support in SQL in string to date conversion functionality.

The performance evaluation tests of different SPARQL engines have shown that SparqlEngineDb and Jena SBD are the only engines to fully support all tested queries.

3.3 Current usage

The development of the new SPARQL engine was driven by the needs of real application projects. During development, support for SparqlEngineDb usage has been added to numerous applications. The following list shows some of them:

- Ontowiki[ADR06], is a semantic wiki engine working completely on RDF data. OntoWiki uses SPARQL to access the underlying databases.
- LDAP2SPARQL[6] provides an LDAP interface to SPARQL endpoints. It allows using LDAP capable clients to retrieve data from RDF databases.
- Vakantieland[7] is a knowledge base for tourist information in the Netherlands and retrieves all data from an RDF database.

While working on these application projects we noted a number of shortcomings which we hope might be tackled in future versions of SPARQL. We summarize the most important ones in the next section.

4 Possible SPARQL Enhancements

SPARQL allows to query RDF data easily. It supports filtering data by simple conditions like regular expressions, data type checks and boolean combinations of them. In comparison to SQL, however, SPARQL's language feature set is quite limited. Features currently not supported by SPARQL are:

- Operating on distinct models

[5] `http://www.w3.org/2001/sw/DataAccess/`

[6] `http://aksw.org/Projects/LDAP/Backend`

[7] `http://www.vakantieland.nl`

- Using operators in `SELECT` and `CONSTRUCT` patterns
- Mathematic functions
- Aggregates and grouping functionality
- Prepared statements
- Data manipulation and transaction control
- Query concatenation
- Variable renaming
- Access control

Using SPARQL today often still means retrieving raw selected data from the SPARQL endpoint. Consecutively, the client needs to re-examine all fetched data as well as filter and convert them into the required format. This produces unnecessary overhead that could be avoided if SPARQL would support data transformation. Pushing more 'work' into the SPARQL server also means that applications are more light-weight and portable across different systems and programming languages. From our point of view, SPARQL should become for RDF graphs what SQL already is for relational databases. SPARQL is still a W3C working draft, thus it is possible that missing features might be added before the standard is finalized. Some ideas for SPARQL enhancements are explained in more detail in the remainder of this section.

4.1 Operating on Distinct Models

Following the "SQL for graphs" approach, a missing feature catches the eye: SPARQL is only able to operate on a single "database" - RDF triples belonging to a single model. It is often desirable that a database server supports multiple stores that keep data independent of each other. Most current SPARQL server implementations include support to handle distinct models.

While named graphs try to address this problem, it is often favorable to have fully distinct sets of RDF data that do not influence each other. Accidentally or intentionally omitting the graph name causes a query to be executed on the whole model. Separating models of e.g. different users from each other allows better access control. Further, data loss is restricted to a single model when accidentally executing a delete operation.

With the growth of SPARQL end point implementations, the number of non-standardized, different APIs addressing this problem is growing. Before finalizing the SPARQL specification, a solution for this problem should be provided in order to prevent incompatible solutions between server software implementations. The selection of a specific model in SPARQL can for example be easily realized with an additional keyword `USE`:

```
USE <http://example.com/addressbook>
```

It should be possible to obtain a list of all models available on the server. The syntax of such a query should adhere to the normal SPARQL standard. For example, a special system model (similar to the information schema of relational databases) could be provided.

```
USE <http://example.com/sysmodel>
PREFIX sysmodel: <http://example.com/sysmodel-schema/>
SELECT ?model WHERE { ?x rdf:type sysmodel:model }
```

List available models.

4.2 SPARQL as a Calculator

The SPARQL working draft supports ”extensible value testing” by allowing vendors to add custom functions to their SPARQL implementation. An example is given that shows an `aGeo:distance` function calculating the distance between two locations.

Such functionality can be either provided by adding proprietary functions to the server, or by extending the SPARQL specification to support further base functionality. Standard functionality like trigonometric functions or exponents are regularly needed and can thus be added to the standard feature set of SPARQL. Such a step prevents incompatibilities between implementations and allows applications to be portable between SPARQL endpoints.

Supporting elementary mathematical calculations opens SPARQL to a wide range of possible SPARQL queries and use cases. Further, it drastically reduces the need for client side data filtering and proprietary extensions in SPARQL server implementations. The working draft's example function `aGeo:distance`, for example, could be replaced by using power and a square root:

```
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
SELECT ?neighbor WHERE {
  ?a geo:placeName "Grenoble".
  ?a geo:lat ?ax. ?a geo:long ?ay.
  ?b geo:placeName ?neighbor.
  ?b geo:lat ?bx. ?b geo:long ?by.
  FILTER ( sqrt(pwr(?ax-?bx,2)+pwr(?ay-?by,2)) < 10 ).
}
```

4.3 Extended Operator Usage

SPARQL is limited in a way that only full values of subjects, predicates and objects can be returned from queries. Modifying values of selected variables through the query allows pushing tasks into the database server and reduces the need for data manipulation on client side.

Another often needed feature is to find out which languages are used in conjunction with literals in the data store. SPARQL currently does not provide any way to get a list of all used languages or datatypes. This problem can be solved by allowing operators on variable selection clauses:

```
SELECT DISTINCT datatype(?o) WHERE  { ?s ?p ?o }
```

By adding such functionality, renaming result variables gets important:

```
SELECT DISTINCT substr(str(?o), 0, 10) AS ?name
  WHERE  { ?s ?p ?o }
```

4.4 Grouping and Aggregates

Grouping and aggregate functionality is a useful feature currently missing in SPARQL. Virtuoso already implements aggregate functions by means of implicit grouping[8]. Implicit grouping as Virtuoso provides introduces some problems:

- Queries are hard to understand because variable selection and cgroup creation is mixed.
- It is not possible to select variables without using them to create a group.
- Grouping is only available globally, not for single graph patterns.
- Operators cannot be used on group variables.

Virtuoso's approach provides basic grouping functionality but should be extended to prevent these problems. Readability difficulties and the negative effects of implicit grouping can be prevented by introducing a new clause similar to SQL's `GROUP BY`.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name, count(?mbox) as ?count
WHERE { ?x foaf:name ?name.
        ?x foaf:mbox ?mbox. }
GROUP BY ?name
```

It has to be defined at which position in a SPARQL query a `GROUP BY` clause should be allowed. For that it has to be decided whether it is desirable to be able to apply grouping to individual graph patterns. The `ORDER BY` clause is also only allowed on a global level, although it could make sense for ”subqueries” like optional clauses.

[8] http://docs.openlinksw.com/virtuoso/rdfsparqlaggregate.html

4.5 Prepared Statements

In SQL, prepared statements serve several purposes:

- Queries do need to be parsed only once and can be executed multiple times without further parsing overhead.
- Query execution plans need to be calculated only once.
- For web applications, they prevent SQL injection attacks, since there is no need to escape values.
- The amount of data transmitted between client and SQL server is reduced.

Prepared statements can speed up repetitive queries of the same form a great deal and enhance security. By supporting prepared statements, SPARQL would benefit in the same way. We propose the following prepared statement syntax rules:

- Placeholders are defined using a normal SPARQL variable name but using two consecutive question marks as prefix.
- SPARQL implementations need two new API methods, namely `prepare` and `execute`.

Using named placeholders (as Oracle does for SQL[9]) has several advantages over position-based anonymous placeholders (e.g. MySQL [10]):

- The user does not need to know the exact order of parameters the query requires.
- Repeated usage of the same variable requires only one data instance to be provided and transmitted to the server. This saves memory and bandwidth.

The preparing and execution of a SPARQL statement with placeholders in PHP would for example would look as follows:

```
$query = "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?mbox
WHERE { ?x foaf:name ??name. ?x foaf:mbox ?mbox. }";

$statement = $model->prepare($query);

$data1 = $statement->execute(array('name'=>'John Doe'));
$data2 = $statement->execute(array('name'=>'Jane Doe'));
```

[9] http://download-east.oracle.com/docs/cd/B19306_01/appdev.102/b14250/oci05bnd.htm

[10] http://dev.mysql.com/doc/refman/5.0/en/sqlps.html

5 Conclusion and Outlook

We presented our SPARQL implementation SparqlEngineDb, which for the first time enables the largest portion of Web application (implemented in PHP) to base on semantic technologies. When compared to other SPARQL implementations SparqlEngineDb is mostly faster, only preceded by Virtuoso, which technically follows a very similar approach but is implemented solely in C.

Use-case of SparqlEngineDb showed that for implementing real-world applications the use of SPARQL is still cumbersome and lack some crucial features. Our vision is that SPARQL eventually will support a similar comprehensive querying of RDF graphs as SQL allows of relational databases. As a first step in that direction we suggested several enhancements to the SPARQL standard, which could be implemented in a quite canonical downward compatible way.

With a growing usage of SPARQL as the standard to query RDF data, the Semantic Web will gain more interoperability. By continuously evolving the SPARQL standard, it will become easier to create sophisticated applications that operate on data stored in different server implementations.

References

[ABL$^+$07] Sören Auer, Chris Bizer, Jens Lehmann, Georgi Kobilarov, Richard Cyganiak, and Zachary Ives. DBpedia: A Nucleus for a Web of Open Data. In *Proceedings of ISWC 2007, November 11-15, Busan, Korea.*, 2007.

[ADR06] Sören Auer, Sebastian Dietzold, and Thomas Riechert. OntoWiki - A Tool for Social, Semantic Collaboration. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*, pages 736–749. Springer, 2006.

[BB06] Dave Beckett and Jeen Broekstra. SPARQL Query Results XML Format. W3c candidate recommendation, World Wide Web Consortium (W3C), April 2006.

[Biz04] Chris Bizer. RAP (RDF API for PHP). Website, November 2004. http://www.wiwiss.fu-berlin.de/suhl/bizer/rdfapi/.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem*, 3(2-3):158–182, 2005.

[Ins89] American National Standards Institute. *Information Systems -Database Language - SQL*. ANSI X3.135-1992. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1989.

[PS06] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF (Working Draft). W3c working draft, World Wide Web Consortium (W3C), 2006.