Scalable Numerical SPARQL Queries over Relational Databases

Minpeng Zhu, Silvia Stefanova, Thanh Truong, Tore Risch
Department of Information Technology, Uppsala University
Box 337, SE-75105 Uppsala, Sweden
{Minpeng.Zhu, Silvia.Stefanova, Thanh.Truong, Tore.Risch}@it.uu.se

ABSTRACT

We present an approach for scalable processing of SPARQL queries to RDF views of numerical data stored in relational databases (RDBs). Such queries include numerical expressions, inequalities, comparisons, etc. inside FILTERs. We call such FILTERs *numerical expressions* and the queries - *numerical SPARQL queries*. For scalable execution of numerical SPARQL queries over RDBs, numerical operators should be pushed into SQL rather than executing the filters as post-processing outside the RDB; otherwise the query execution is slowed down, since a lot of data is transported from the RDB server and furthermore indexes on the server are not utilized. The *NUMTranslator* algorithm converts numerical expressions in numerical SPARQL queries into corresponding SQL expressions. We show that NUMTranslator improves substantially the scalability of SPARQL queries based on a benchmark that analyses numerical logs stored in an RDB. We compared the performance of our approach with the performance of other systems processing SPARQL queries to RDF views of RDBs and show that NUMTranslator improves substantially the scalability of numerical queries compared to the other systems' approaches.

Keywords

SPARQL queries; RDF views of relational databases; numerical expressions; query rewrites; query optimization

1. INTRODUCTION

The Semantic Web provides uniform data representation for integrating data from different data sources by using established well-known formats like RDF, RDFS, OWL, and the standard query language SPARQL. Semantic Web seems promising to integrate and search industrial data [2].

Our application scenario is from the industrial domain, where sensors on machines such as trucks, pumps, kilns, etc., produce large volumes of log data. Such log data describes measured values of certain components at different times and can be used for analyzing machine behavior. Furthermore, the geographic locations of machines are often widely distributed and maintained locally in autonomous RDBs called *log databases*. We are developing the FLOQ (Federated LOg database Query) system, which is a system for historical analyses over federations of autonomous log databases using SPARQL queries. To discover abnormal machine behaviors, a user of FLOQ defines SPARQL queries to these log databases. FLOQ processes a SPARQL query by first finding the relevant log databases containing the desired data, then sending local SPARQL queries to them, and finally collecting the local query results to obtain the final result.

(c) 2014, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2014 Joint Conference (March 28, 2014, Athens, Greece) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

In this paper we concentrate on scalable historical analyses by SPARQL queries of log data stored in a single relational database. Suspected abnormal machine behaviors are discovered and analyzed by specifying numerical SPARQL queries to an RDF view of the RDB. The queries analyze log data through numerical FILTERs containing numerical operators [11]. For example, query *Q1* retrieves the machine identifiers *m* for which a sensor has measured values *mv* of measurement class *A* higher than the expected values *ev* by a threshold value *@thA* during the time from *bt* to time *et*. Here *<prod>* denotes the URI for the RDF view of the RDB.

```
Q1:
SELECT ?m ?bt ?et
FROM <prod>
WHERE {?measuresA  log:mA_BySensor  ?sensor.
       ?measuresA  log:mA/bt        ?bt.
       ?measuresA  log:mA/et        ?et.
       ?measuresA  log:mA/m         ?m.
       ?measuresA  log:mA/mv        ?mv.
       ?sensor     log:sensor/ev    ?ev.
       FILTER (?mv > (?ev + @thA))         }
```

In FLOQ, SPARQL queries to RDBs are processed by generating a local execution plan containing calls to one or several SQL queries sent to a back-end RDBMS for evaluation. SPARQL queries that cannot be completely processed by SQL are instead partially processed by an execution plan interpreter in FLOQ. However, in order for the SQL queries to return the minimal required data, it is desirable that as much as possible of the SPARQL query is translated to SQL [8].

In FLOQ numerical SPARQL queries are defined over an automatically generated RDF view over an RDB expressed in *ObjectLog* [6], which is a Datalog dialect that supports objects for representing URIs and typed literals [9], disjunctive queries for UNION expressions, and foreign predicates to represent numerical operators in queries. The SPARQL queries are parsed into ObjectLog queries to the RDF view. Internally representing queries in ObjectLog permits domain calculus query transformations and optimizations before generating the execution plan. Calls to tuple calculus SQL query strings are made as foreign predicates. Foreign predicates are also used for accessing URIs in the execution plan. Doing all processing in

the RDB is complicated, and requires implementing SPARQL operators not supported by SQL as RDB-specific UDFs. We show that ObjectLog query transformations enable scalable execution by the RDBMS.

Numerical SPARQL queries contain variables bound to numbers and calls to numerical functions and operators. For scalable execution, it is important that such numerical expressions are pushed into corresponding SQL expressions and executed on the RDBMS server, which is the subject of this paper. The NUMTranslator algorithm converts numerical SPARQL queries into SQL queries where numerical expressions are pushed into SQL. For example, *Q1* is converted into SQL query *SQL1,* where the numerical expression in the SPARQL FILTER is translated into a corresponding SQL expression.

```
SQL1:
SELECT m.m, bt, et
FROM MeasuresA m, SENSOR s
WHERE m.m=s.m AND
      m.s=s.s AND
      m.mv > s.ev + @thA
```

A particular problem is that SPARQL and ObjectLog are domain calculus languages where variables can be bound to numbers, while SQL is a tuple calculus language where variables have to be bound to tuples in relations. The NUMTranslator algorithm translates domain calculus expressions into corresponding SQL tuple calculus expressions after having applied domain calculus transformation on the ObjectLog representation.

We show that NUMTranslator improves substantially the query performance for numerical SPARQL queries compared to other approaches used by other systems.

In summary the contributions are:

- We propose a table driven approach to translate numerical domain calculus operators into numerical SQL tuple calculus operators.
- We present the NUMTranslator algorithm that extracts numerical ObjectLog expressions and translates them into corresponding numerical SQL expressions.
- We compare the performance of numerical SPARQL queries to RDF views of RDBs with and without applying NUMTranslator, and show that the algorithm substantially improves the query performance.
- We compare the performance of our approach with the performance of other systems processing SPARQL queries over RDF views of RDBs and show substantially better performance.

The rest of this paper is organized as follows: Section 2 presents a scenario where the approach is applicable. Section 3 overviews the system architecture. Section 4 describes the NUMTranslator algorithm. Section 5 discusses performance experiments. Section 6 describes related work. Conclusions and future work are described in section 7.

2. MOTIVATING SCENARIO

We present a common scenario from an industrial setting where it is desirable to analyze historical log data in order to find abnormal machine behavior. Log data from embedded sensors is stored in a relational log database.

Figure 1 shows the schema of the RDB storing log data measured by sensors embedded in machine installations. Table *Machine(m, mm)* stores meta-data about each machine installation, i.e. machine identifier and model name. The table *Sensor(m, s, sm, mc, ev, ad, rd)* stores information about each sensor installation, i.e. the machine installation *m* where a sensor *s* is embedded, sensor model name *sm*, the kind of measurement (measurement class) *mc*, expected sensor value *ev*, absolute error *ad* and relative error *rd*. The attribute *mc*, measurement class is used to identify different kind of measurements, e.g. oil pressure, temperature, etc. The tables *MeasuresA(m, s, bt, et, mv)* and *MeasuresB(m, s, bt, et, mv)* store log data of kind *A* and *B* read from sensors *s* embedded in machine installations *m*. The begin time *bt* and the ending time *et* for a sensor reading are also stored, while the measured value for a certain time stamp is denoted by *mv*. The columns *m*, (*m, s*), and (*m, s, bt*) are primary keys in the tables *Machine*, *Sensor*, and *MeasuresA and MeasuresB*, respectively. The column *m* in tables *MeasuresA*, *MeasuresB,* and *Sensor* references the column *m* in the table *Machine* as foreign key. Furthermore, columns (*m, s*) in tables *MeasuresA* and *MeasuresB* reference columns (*m, s*) in table *Sensor* as a composite foreign key.

```
Machine(m, mm)
Sensor(m, s, sm, mc, ev, ad, rd)
MeasuresA(m, s, bt, et, mv)
MeasuresB(m, s, bt, et, mv)
```

Figure 1. RDB schema for log data

The RDF view of the RDB is illustrated by the RDF graph in Figure 2.

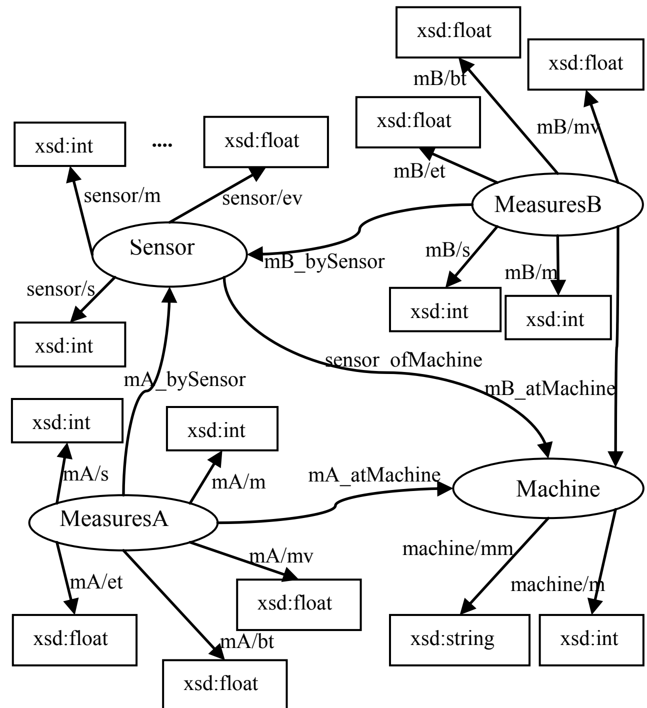

Figure 2. RDF graph of the RDF view for the example RDB

Next we define two more typical numerical SPARQL queries to the log database, *Q2* and *Q3,* that discover abnormal machine behaviors. Query *Q2* identifies a potential failure by retrieving for machine models *M_1, M_2,* and *M_3* those *machineid* where, during the time interval *(bt, et),* the measured value *mv* was above 75% of the allowed deviation *@thA* from the expected value *ev*.

```
Q2:
SELECT ?machineid ?bt ?et
FROM <prod>
WHERE{?measuresA log:mA_bySensor  ?sensor.
      ?measuresA log:mA/bt        ?bt.
      ?measuresA log:mA/et        ?et.
      ?measuresA log:mA/mv        ?mv.
      ?measuresA log:mA_atMachine ?machineid.
      ?machineid log:machine/mm   ?mm.
      FILTER (?mm in ('M_1','M_2','M_3')).
      ?sensor    log:sensor/ev    ?ev.
      FILTER (?mv > (?ev + 0.75*@thA))    }
```

Query *Q3* identifies abnormal behaviors of machines of a measurement class based on absolute deviations: when and for which machine identifiers did the pressure reading of class *B* deviate more than *@thB* from its expected value *ev*?

```
Q3:
SELECT ?m ?bt ?et
FROM <prod>
WHERE {?measuresB  log:mB/bt       ?bt.
       ?measuresB  log:mB/et       ?et.
       ?measuresB  log:mB/mv       ?mv.
       ?measuresB  log:mB_bySensor ?sensor.
       ?sensor     log:sensor/m    ?m.
       ?sensor     log:sensor/ev   ?ev.
       BIND ((?mv-?ev) as ?temp).
       FILTER (abs(?temp) > @thB)       }
```

3. FLOQ OVERVIEW AND QUERY PROCESSING

Figure 3 illustrates processing of numerical SPARQL queries by FLOQ.

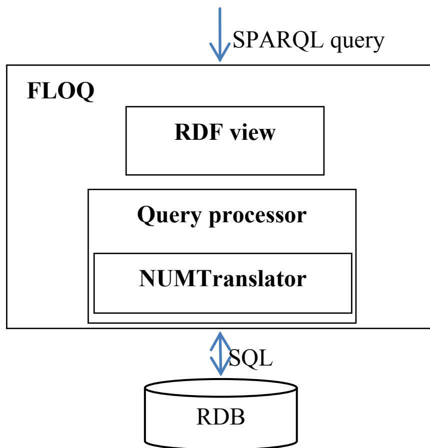

Figure 3. FLOQ query processor

The *RDF view* over the RDB is automatically generated based on the database schema and ontology mapping tables in FLOQ. The used mappings conform to the direct mapping recommended by W3C [10].

We define a unique RDFS class for each relational table, except for link tables [10] representing set-valued properties as many-to-many relationships. In addition, RDF properties are defined for each column in a table. For example, the RDFS class with the URI *<log:mA>* represents the table *MeasuresA*, while *<log:mA/bt>* and *<log:mA/et>* represent the columns *bt* and *et* in *MeasuresA,* respectively.

The RDF view is defined in terms of:

- *Source predicates* $R(a_1, a_2, ..., a_n)$ that represent the content of each referenced relational database table *R* where the tuple $(a_1, ..., a_n)$ represents a row in *R*.
- *URI-constructor* predicates that construct URIs to identify rows in tables.
- *Mapping tables* that map relational schema elements to RDF concepts.

The complete RDF view definitions can be found in [9]. The query processing steps in FLOQ are shown in Figure 4.

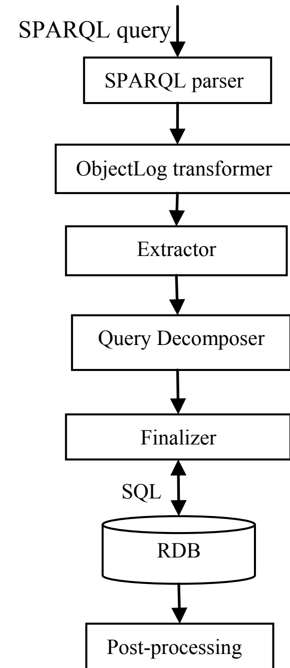

Figure 4. Query processing steps

The *SPARQL parser* first transforms the SPARQL query into an ObjectLog expression where each triple pattern in the query becomes a reference to the RDF view of the RDB. Then the *ObjectLog transformer* generates a simplified disjunctive normal form (DNF) predicate. The NUMTranslator algorithm performs the *extractor* and *finalizer* steps. The extractor collects from conjunctions predicates that can be translated to SQL, called *access filters*. The *query decomposer* then optimizes the query, producing a query execution plan where access filters are called. The finalizer traverses the execution plan to translate the extracted predicates in the access filters into SQL expressions. When the execution plan is interpreted, the generated SQL statements are sent to the RDB for execution. The non-extracted predicates are not translated to SQL and have to be processed outside the RDB by *post-processing* operators. For example,

such operators are URI-constructors and numerical expressions not supported by the SQL engine.

4. THE NUMTRANSLATOR ALGORITHM

The NUMTranslator uses a table-driven approach to define which SPARQL operators to extract and translate into corresponding SQL operators and functions. Table 1 defines the SPARQL to SQL operator translations:

Table 1. SPARQL to SQL operators to translate

SPARQL	SQL	INFIX	FUNCTION
>	>	True	False
<	<	True	False
=	=	True	False
!=	<>	True	False
+	+	True	True
-	-	True	True
ABS	ABS	False	True
UCASE	UPPER	False	True
etc.			

In Table 1 there is one row for each SPARQL operator or function (column *SPARQL)* that can be translated into SQL. The column *SQL* defines the corresponding SQL operator or function. A value in the column *INFIX* is true when the corresponding SQL operator is an infix operator *op* on operands *x* and *y*, i.e. *x op y* (e.g. x+y); otherwise it is an SQL function on format *f(x,y,..)*. The column *FUNCTION* is true when the operator is a non-Boolean function returning a value.

4.1 The NUMTranslator extractor

The extractor is applied on each ObjectLog conjunction in the simplified predicate received by the ObjectLog transformer. The extractor collects predicates that can be translated to SQL. Such predicates are i) source predicates *SPs* representing RDB tables, and ii) *non-source predicates (NSPs)* that are defined in Table 1 as translatable to SQL.

Figure 5 shows the ObjectLog representation of *Q1* after it has been transformed by the ObjectLog transformer.

```
Q1(m, bt, et):-
1  MeasuresA(m, s, bt, et, mv)        and
2  mv > v36                           and
3  v36 = ev + @thA                    and
4  Sensor(m, s, _, _, ev, _, _)
```

Figure 5. ObjectLog of query *Q1*

In this case all predicates in *Q1* are translatable to SQL since *MeasuresA* and *Sensor* are SPs, and > and + are NSPs defined in Table 1.

The steps of the extractor are the following:

1. Initialize a variable *Xpreds* for the first found SP, denoted *R1,* in the conjunction and bind a variable *Rest* to the other predicates.
2. Iteratively extract from *Rest* the predicates that have some common variable with some extracted predicate in *Xpreds*, which are either SPs or NSPs defined in Table 1.
3. Construct an *access filter* of all extracted predicates in *Xpreds* since those can be fully translated to SQL.
4. While there are some remaining SP, *R2*, in *Rest*, re-initialize *Xpreds* by *R2* and *Rest* by the remaining predicates, and repeat steps 2-3.
5. Finally, construct a conjunction of the access filters and *Rest*.

For example, for *Q1* the predicates in *Xpreds* are extracted in the following order:

1. *MeasuresA(m, s, bt, et, mv)* (line 1), since it is an SP.
2. *>(mv, v36)* (line 2) since > is defined in Table 1 and the variable *mv* is common with the extracted *MeasuresA*.
3. *Sensor(m, s, _, _, ev, _, _)* (line 4) since it is an SP having common variables (*m* and *s*) with *MeasuresA()*.
4. *V36 = ev + @thA* (line 3) since + is defined in Table 1 and the variable *ev* is common with the extracted *Sensor* predicate.

Then the following conjunctive access filter *F1* is formed by the predicates in *Xpreds*:

```
 F1(m,s,bt,et,mv,ev):-
1  MeasuresA(m, s, bt, et, mv)      and
2  Sensor(m, s, _, _, ev, _, _)     and
3  v36= ev + @thA                   and
4   mv > v36
```

No non-translatable predicates remain in *Rest*.

4.2 Query decomposition

To optimize the query produced by the extractor, the query decomposer uses cost-based optimization [6] to produce an optimized execution plan. Based on heuristics and statistic of the queried RDB, execution cost and selectivities of access filter are estimated. Default cost parameters are used by the optimizer to estimate the execution cost and selectivities of predicates if no statistic is available. The decomposer will then reorder the access filters and the post processed predicates to generate an optimized execution plan. We do not further elaborate the query decomposer here.

4.3 The NUMTranslator finalizer

The finalizer translates access filters in the decomposed execution plan into calls to an SQL interface operator, *sql* that sends generated SQL strings to the back-end RDB for execution.

ObjectLog numerical expressions are translated into SQL numerical expressions by recursively replacing all ObjectLog domain variables that represent numerical expressions with their bound expressions. For example, the variable *v36* in line 4 in *F1* doesn't represent a relational column and is replaced by its bound expression in line 3, and then the obtained expressions is *mv > ev + @thA*. Thus for *Q1* the execution plan *P1* becomes the following:

```
                 (m, bt, et)
                      ↑
γ sql(ds, "SELECT m.m, bt, et FROM MeasuresA
m, SENSOR s WHERE m.mv > s.ev + @thA AND
m.m=s.m AND m.s=s.s", (m, bt, et))
```

Figure 6. Execution plan *P1* with NUMTranslator

The execution plan contains an algebra expression where the *apply* operator γ *fn(..)* calls the *foreign predicate sql(ds, q, result)* implemented in Java. The foreign predicate *sql* sends an

SQL query *q* to the RDBMS data source *ds* for execution and iteratively returns bindings of tuples, *result*.

If NUMTranslator had not been applied, all numerical operators would have to be post-processed, which would slow down the query execution since filtering cannot be made in the database server.

For example, if NUMTranslator is turned off, for *Q1* the following execution plan *P2* is produced that doesn't contain any numerical SQL operators corresponding to numerical SPARQL operators, which are instead post-processed:

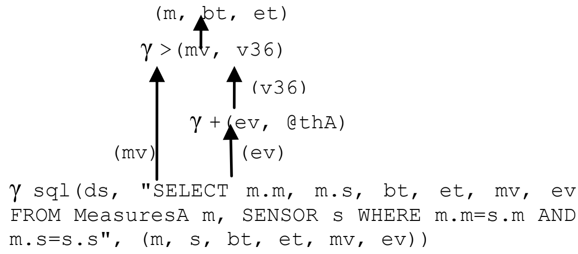

Figure 7. Execution plan *P2* without NUMTranslator

Comparing the two execution plans *P1* and *P2* it can be seen that the *sql* operator in *P2* retrieves much more data than *P1*, so if NUMTranslator is turned off lots of data needs to be filtered out outside the RDB server. Furthermore, the utilization of indexes on the SQL numerical expression by the back-end database server makes significant performance difference. We show in the next section that applying NUMTranslator substantially improves the query performance of numerical SPARQL queries.

5. PERFORMANCE MEASUREMENTS

We compared the performance for executing the numerical queries *Q1, Q2,* and *Q3* in FLOQ with and without applying NUMTranslator. Furthermore, we compared the query performance of FLOQ with the query performance of D2RQ [1] for *Q1, Q2,* and *Q3,* for the same back-end relational database. We tried to run the queries with both ontop [7] and Virtuoso [3] as well, but none of our numerical SPARQL queries could be run, indicating that those systems do not provide full support for processing numerical SPARQL queries.

All experiments are carried out on a MS SQL Server 2008 R2 installed on a server machine with 8 AMD Opteron™ 6128 processors, 2.00 GHz CPU and 16GB RAM. The RDB is populated by loading sensor data into the MS SQL server. B-tree indexes are created on the columns *mm, mv, bt, et, ev, ad,* and *rd* to speed up the queries.

All measurements were taken both for cold and warm runs. The cold runs were made immediately after the RDBMS server was started, which implied that there were no data cached in the buffer pool and the executed query wasn't optimized by the RDBMS. Thus a measured query execution time for a cold run includes the time for i) reading data from disk, ii) SQL query optimization on the RDBMS server, iii) communication, and iv) post-processing of data on the client. The warm runs were made after a query was executed once. Since the back-end RDBMS has a statement cache a same SQL query executed twice will be optimized the first time it is run. Therefore, warm executions do not include RDBMS query optimization time.

The plotted values are mean values of three measurements. The standard deviation is less than 10% in all cases. To investigate the SQL query produced by all the other systems we use the system profiling tool of MS SQL server when running a query.

The following notations are used in the performance diagrams:

- *NUMTranslator*: FLOQ with NUMTranslator turned on, i.e. the SPARQL numerical expressions are translated into corresponding SQL expressions.
- *Naive*: FLOQ with NUMTranslator turned off, i.e. the SPARQL numerical expressions are not translated into corresponding SQL numerical expressions.
- *D2RQ*: D2RQ version [0.8.1] configured with the system's default mappings.

Figure 8, 9 and 10 show the execution times for both cold and warm runs for *Q1, Q3, and Q2* while scaling the databases size from 1 GB to 15 GB.

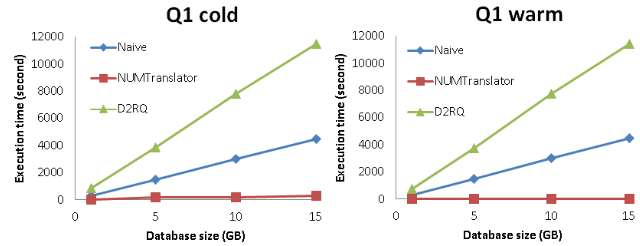

Figure 8. Execution times for *Q1*

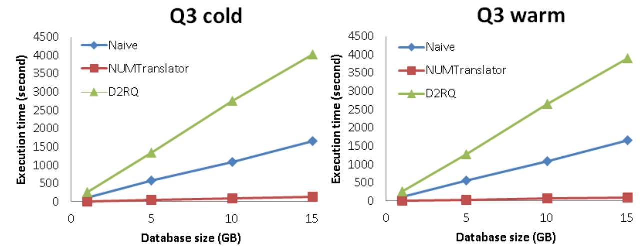

Figure 9. Execution times for *Q3*

Figure 8 and 9 show that *NUMTranslator* substantially improves the query execution scalability compared to *Naïve* for numerical SPARQL queries like *Q1* and *Q3* with highly selective numerical FILTERs: 0.04% for Q1 and 3% for Q3. In these cases pushing the numerical FILTERs to SQL is more profitable than filtering large data amounts on the client. The performance of D2RQ is worse than Naïve since D2RQ sends to the RDBMS an SQL query that doesn't contain numerical expressions, and is a much more complex query with more joins. Furthermore, *Q3* had to be manually changed for D2RQ to remove the BIND operator, since otherwise D2RQ wouldn't return correct result.

Measurement results for *Q2* are shown in Figure 10. For *Q2* the results for *NUMTranslator* and *Naïve* are presented in a separate diagram, since they are very close. It can be seen on Figure 10 that *NUMTranslator* doesn't improve the query performance for non-selective queries like *Q2* where the FILTER selects 43% of the data. In this case pushing the numerical SPARQL filters to be executed to the RDBMS server doesn't make a significant difference compared to post-filtering data on the client.

D2RQ performs worse for *Q2* since it doesn't translate any of the FILTERs and it furthermore generates a very complex SQL query with many joins.

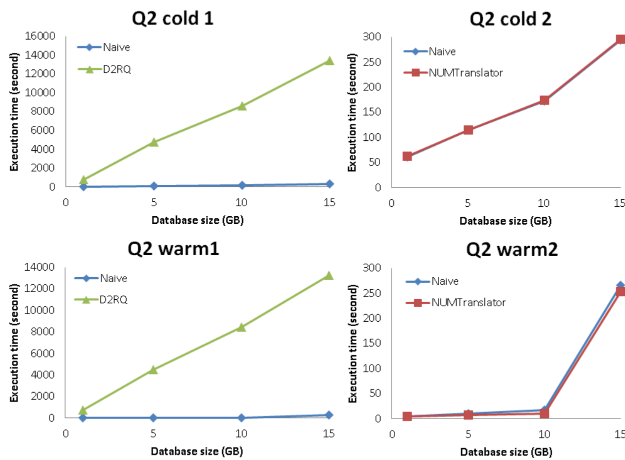

Figure 10. Execution times for *Q2*

In general, the experiments show that NUMTranslator substantially improves the query performance of numerical SPARQL queries where the numerical FILTERs have high selectivity.

6. RELATED WORK

Virtuoso RDF Views [3] and D2RQ [1] are other systems that process SPARQL queries to RDF views of RDBs. These systems implement compilers that translate SPARQL directly to SQL. By contrast, FLOQ first generates ObjectLog queries to a declarative RDF view of the RDB, and then transforms the SPARQL queries to SQL by logical transformations.

We didn't find any publication of how D2RQ compiles numerical SPARQL queries into SQL and the documentation for Virtuoso's SQL generation is very limited [3]. However, by using the profiling tool of the RDBMS and the debug logging of Virtuoso we were able to analyze what queries were actually sent to the RDBMS, showing that neither of those systems translates numerical SPARQL expressions into corresponding SQL expressions.

The ontop system [7] also enables SPARQL queries to RDF views of RDBs by translating SPARQL to Datalog programs, which are rewritten and translated to SQL. A difference to ontop is the table driven NUMTranslator algorithm, which makes it very easy to extend for new operators. Furthermore, FLOQ generates execution plans containing calls to SQL intermixed with expressions interpreted in the client. This enables FLOQ to interpret in the client SPARQL operators not available in SQL. In addition NUMTranslator translates the domain calculus SPARQL queries into tuple calculus SQL queries by substituting variables with their bound expressions.

7. CONCLUSIONS AND FUTURE WORK

We presented the FLOQ system where the NUMTranslator algorithm uses a table driven approach to translate numerical domain calculus SPARQL expressions into corresponding numerical SQL expressions. This enables scalable processing of numerical SPARQL queries to RDF views over RDBs.

The approach was evaluated on a benchmark scenario in an industrial setting where logged data stored in an RDB was analyzed using numerical SPARQL queries. We compared the performance of the SPARQL queries with and without applying NUMTranslator. The experiments show that NUMTranslator substantially improves the query performance of numerical SPARQL queries in particular when the numerical expressions inside FILTERs are highly selective.

We also compared our approach with other systems that translate SPARQL queries to SQL. Only D2RQ could execute our queries, but substantially slower since D2RQ does not employ an approach similar to NUMTranslator.

As our next step, we will investigate numerical SPARQL queries searching large numbers of distributed log databases combined through an ontology. Another issue is creating benchmarks based on randomly generating SPARQL queries [5]. Furthermore, query processing and mediation strategies over other back-ends than RDBs [4] in our setting should be investigated.

8. ACKNOWLEDGMENTS

This work is supported by EU FP7 project Smart Vortex and the Swedish Foundation for Strategic Research under contract RIT08-0041.

9. REFERENCES

[1] Bizer, C., Cyganiak, R., Garbers, G., Maresch, O., and Becker, C. 2009. *The D2RQ Platform v0.7 - Treating Non-RDF Relational Databases as Virtual RDF Graph*, http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/

[2] Björkelund, A., Edström, L., etc. 2011. On the integration of skilled robot motions for productivity in manufacturing, In *Proc. of IEEE International Symposium on Assembly and Manufacturing*, Tampere, Finland.

[3] Erling, O. and Mikhailov, I. 2009. RDF Support in the Virtuoso DBMS, *Studies in Computational Intelligence*, Vol. 221

[4] Langegger, A., Wöß, W., and Blöchl, M. 2008. A Semantic Web Middleware for Virtual Data Integration on the Web, *5th European Semantic Web Conference ESWC 2008*.

[5] Langegger, A. and Wöß, W. 2009. RDFStats – The Extensible RDF Statistics Generator and Library, *8th International Workshop on Web Semantics, DEXA 2009,* Linz, Austria, August 31-September 40.

[6] Litwin, W. and Risch, T. 1992. Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6.

[7] Rodriguez-Muro, M., Rezk, M., Hardi, J., Slusnys, M., Bagosi, T., and Calvanese, D. 2013. Evaluating SPARQL-to-SQL Translation in Ontop, *ORE 2013*

[8] Sequeda, J. F., and Miranker, D. P. 2013. *Ultrawrap: SPARQL Execution on Relational Data*, Tech. Report, Univ. of Texas at Austin. http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2078.pdf

[9] Stefanova, S., and Risch, T. 2011. Optimizing Unbound-property Queries to RDF Views of Relational Databases. *7t International workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011)*, Bonn, Germany.

[10] Arenas, M., Bertails, A., Prud'hommeaux, E., and Sequeda, J. 2012. A Direct Mapping of Relational Data to RDF, http://www.w3.org/TR/rdb-direct-mapping/

[11] Harris, S., and Seaborne, A. 2013. SPARQL 1.1 Query Language, http://www.w3.org/TR/sparql11-query/