Business Intelligence Extensions for SPARQL

Orri Erling (Program Manager, OpenLink Virtuoso) and Ivan Mikhailov (Lead Developer, OpenLink Virtuoso).

OpenLink Software, 10 Burlington Mall Road Suite 265 Burlington, MA 01803 U.S.A, {oerling,imikhailov}@openlinksw.com
WWW home page: http://www.openlinksw.com/

Table of Contents

Abstract

We believe that the possibility to use SPARQL as a front end to heterogenous data without significant cost in performance or expressive power is key to RDF taking its rightful place as the *lingua franca* of data-integration. To this effect, we demonstrate how RDF and SPARQL can tackle a standard relational workload.

We discuss extending SPARQL for business intelligence (BI) workloads and relate experiennces on running SPARQL against relational and native RDF databases. We use the well known TPC H benchmark as our reference schema and workload. We define a mapping of the TPC H schema to RDF and restate the queries as BI extended SPARQL. To this effect, we define aggregation and nested queries for SPARQL.

We demonstrate that it is possible to perform the TPC-H workload restated in SPARQL against an existing RDBMS withut loss of performance or expressivity and without changes to the RDBMS.

Introduction and Motivation

RDF promises to be a top-level representation for data extracted or dynamically mapped from any conceivable source. Thus, RDF's chief promise is in the field of information integration, analysis and discovery. Yet it is difficult to imagine any business reporting, let alone more complex information integration task that would not involve aggregating and grouping.

As a data access and data integration vendor, OpenLink has a natural interest in seeing SPARQL succeed as a top level language for answering business questions on data mapped from any present day data warehouse or other repository.

This potential role of SPARQL is however fundamentally undermined if SPARQL cannot perform any part of the database industry's baseline business intelligence benchmark, TPC H.

To this effect, we have extended SPARQL with expressions in results, aggregates and grouping and derived tables. These extensions allow a straightforward translation of arbitrary SQL queries to SPARQL. We call this extended SPARQL "SPARQL BI".

We demonstrate the operation of SPARQL BI versions of TPC H queries on relational data managed by Virtuoso and Oracle. We also demonstrate the same workload on the same data stored as RDF in Virtuoso.

Test Data

We use the TPC H schema mapped to RDF in all our examples. The table names are directly converted to classes and the column names are directly converted to predicates in namespace http://www.openlinksw.com/schemas/TPC-H. The prefix tpch is used to refer to this namespace throughout the paper.

SPARQL Extensions

Expressions

In its proposed recommendation form, SPARQL does not allow returning any value that is not retrieved through a triple pattern. Expressions are only allowed in filters but cannot be returned. We lift this restriction by allowing expressions in the result set. Consider:

```
select (?extendedprice * (1 - ?discount))
where {
?line a tpch:lineitem ;
tpch:lineextendedprice ?extendedprice ;
tpch:linediscount ?discount . }
```

We can shorten the notation as

```
select (?line->tpch:extendedprice * (1 - ?line->tpch:discount))
where { ?line a tpch:lineitem }
```

The -> (dereference) operator allows referring to a property without naming it as a variable.

This is exactly equivalent to having a triple pattern binding a variable to the mentioned property of the subject within the group pattern enclosing the reference. For a select, the enclosing group pattern is considered to be the top level pattern of the where clause or in the event of a union, the top level of each term of the union. Each distinct dereference adds exactly one triple pattern to the enclosing group pattern, thus multiple uses -> do not each add a triple pattern. Having multiple copies of an identical pattern might lead to changes in cardinality if multiple input graphs were being considered.

If a lineitem had multiple discounts or extended prices, then we would get the cartesian product of both. If a property referenced via -> is absent, the expression does not get evaluated in the first place.

The optional dereference operator |> will produce an unbound value if the property does not exist. Further, mentioning the same chain of dereferences multiple times in the same group pattern will not cause redundant triple patterns to be added or result in more joining that is necessary.

We further allow expressions in the place of variables in triple patterns. To scope the above query to orders by customers in France, we could write:

```
select (?li->tpch:extendedprice * ?li->tpch:discount)
where {
?li a tpch:lineitem .
?li->tpch:l_orderkey->tpch:o_custkey->tpch:c_nationkey
tpch:n_name "France" . }
The sequence of dereferences expands into triple patterns, as in:
... ?li tpch:l_orderkey ?v1 .
?v1 tpch:o_custkey ?v2 .
?v2 tpch:c_nationkey ?v3 .
?v3 tpch:n_name "Frannce" .
```

Aggregation

We introduce the /sum/, /count/, /avg/, *min* and *max* aggregate functions from SQL. Their semantics with respect to NULL are inherited from SQL. To count result rows without regard to any value being defined, /count (*)/ is introduced.

If grouping is desired, aggregate expressions can be combined with non- aggregate expressions in a selection list. The non-aggregate expressions will function as grouping columns, i.e. the aggregates are calculated for each distinct combination of the grouping columns. No special GROUP BY clause is needed.

```
select ?l->tpch:l_linestatus count(*) sum(?l->tpch:extendedprice)
where {l a tpch:lineitem }
```

gives the count and total value of lineitems for each distinct lineitem status. User defined aggregates from Virtuoso SQL can be used in SPARQL as well, using the sql: namespace.

Nesting of Queries and Named Results

SQL allows nesting queries, in effect treating the evaluation of a query as a table (derived table) or as a value in an expression (scalar subquery). We allow embedding a SPARQL select in the place of a triple pattern. The syntax is as in

```
select ?line
where {
?line a tpch:lineitem .
{ select max (?l2->tpch:extendedprice) as ?maxprice
where { ?l2 a tpch:lineitem } } .
filter (line->tpch:extendedprice = ?maxprice) }
```

This selects all lineitems with extendedprice equal to the highest extended- price in the set of lineitems.

We note that we have a SQL-style explicit comparison for joining the nested select with the outer select. The bindings that are in scope in the pattern containing the nested select are also in scope inside the nested select. In this the scope rules resemble SQL' s rules for subqueries.

In Virtuoso SQL and Virtuoso/PL we allow SPARQL queries in all places where "plain" SQL select could be used, e.g., SQL query can contain SPARQL subqueries.

Sample Queries

Due to space constraints, we chose to pick only two of the twenty-two queries of the TPC H workload. These were selected because of their relative complexity and use of nested queries.

The TPC H definition states the business questions for Q15 and Q18 as follows:

"Q15, The Top Supplier Query finds the supplier who contributed the most to the overall revenue for parts shipped during a given quarter of a given year. In case of a tie, the query lists all suppliers whose contribution was equal to the maximum, presented in supplier number order."

"Q18, The Large Volume Customer Query finds a list of the top 100 cus- tomers who have ever placed large quantity orders. The query lists the customer name, customer key, the order key, date and total price and the quantity for the order."

The Q15 SQL text used with Virtuoso is:

```
select s_suppkey, s_name, s_address, s_phone, total_revenue
from
supplier,
( select
l_suppkey as supplier_no,
sum(l_extendedprice * (1 - l_discount))
as total_revenue
from lineitem
where
l_shipdate >= {d '1996-01-01'} and
l_shipdate < {fn timestampadd (
SQL_TSI_MONTH, 3, {d '1996-01-01'} ) }
group by l_suppkey) as revenue
where
s_suppkey = supplier_no and
total_revenue = ( select max(total_revenue)
from ( select
l_suppkey as supplier_no,
sum (l_extendedprice * (1 - l_discount))
as total_revenue
from lineitem
where
l_shipdate >= {d '1996-01-01'} and
l_shipdate < {fn timestampadd (
SQL_TSI_MONTH, 3, {d '1996-01-01'} ) }
group by
l_suppkey) as revenue)
order by s_suppkey;
```

The corresponding SPARQL BI text is:

```
sparql
prefix tpch <http://openlinksw.com/schemas/TPC-H/>
select
?supplier ?s_name ?s_address ?s_phone ?total_revenue
where
{
?supplier a tpch:supplier ;
tpch:s_name ?s_name ;
tpch:s_address ?s_address ;
tpch:s_phone ?s_phone .
{
select
?supplier
(sum(l_extendedprice * (1 - l_discount)))
as ?total_revenue
where
{
?lineitem a tpch:lineitem ;
tpch:l_shipdate ?l_shipdate ;
tpch:l_suppkey ?supplier .
filter (
?l_shipdate >= xsd:date ('1996-01-01') and
?l_shipdate < bif:dateadd (
'month', 3, xsd:date ('1996-01-01')) )
}
}
{
select max (?all_totals.total_revenue) as ?maxtotal
where
{
{
sparql select
(sum(l_extendedprice * (1 - l_discount)))
as ?total_revenue
where
{
?lineitem a tpch:lineitem ;
tpch:l_shipdate ?l_shipdate ;
tpch:l_suppkey ?l_suppkey .
filter
(
?l_shipdate >= xsd:date ('1996-01-01') and
?l_shipdate < bif:dateadd (
'month', 3, xsd:date ('1996-01-01')) )
}
} as all_totals
}
}
filter (?total_revenue = ?maxtotal)
}
order by
?supplier;
```

The Virtuoso text of Q18 is:

```
select c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice,
sum(l_quantity)
from lineitem, orders, customer
where
o_orderkey in (
select l_orderkey
from lineitem
group by l_orderkey
having sum(l_quantity) > 250 )
and c_custkey = o_custkey
and o_orderkey = l_orderkey
group by c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice
order by o_totalprice desc, o_orderdate;
```

The SPARQL BI version is:

```
select ?c_name ?customer ?order ?o_orderdate ?o_totalprice
sum(?l_quantity)
from <http://example.com/tpcd>
where {
?customer a tpch:customer ; foaf:name ?c_name .
?order a tpch:order ; tpch:ordertotalprice ?o_totalprice ;
tpch:orderdate ?o_orderdate ; tpch:has_customer ?customer .
[ a tpch:lineitem ; tpch:linequantity ?l_quantity ;
tpch:has_order ?order ] .
{
select ?sum_order sum (?quantity) as ?sum_q
where {
[ a tpch:lineitem ; tpch:linequantity ?quantity ;
tpch:has_order ?sum_order ]
}
} .
filter (?sum_order = ?order and ?sum_q > 250)
}
order by desc (?o_totalprice) ?o_orderdate
```

Experiments

For this test, we had the test data on an Oracle 10G and a Virtuoso 5.0 server on the same machine. The tables from the Oracle and Virtuoso servers were attached to another Virtuoso server, which served as the SPARQL front end.

For Q15, Virtuoso SQL gave the answer in 180 ms and with SPARQL the answer took 3800. For Q18, Virtuoso SQL gave 340 and SPARQL 371 ms.

The large difference with Q15 is due to the SQL compiler choosing a different plan because the reformulated text has a different structure. Some further tuning will eliminate the difference.

With the Oracle backend, we obtained the correct answers but our setup did not pass the queries through to Oracle as a single SQL statement, hence the performance was less than would have been seen if the queries were natively submitted to Oracle.

Some further adjustments will result in the queries passing through the pipeline as single statements, at which point we will have a negligible translation overhead.

The test database was at 1 per cent scale, hence the results are not about TPC H performance per se but are solely aimed at verifying that the correct answers are produced and queries are executed as close to the data as possible.

We also dumped the data as physical triples stored in Virtuoso. Our aim is to arrange things so that the physical triples version will at no point be more than three times slower than the equivalent relational setting on Virtuoso, running with a database scaled to 100G. Reaching this requires some enhancements to our SQL implementation, specifically for dealing with queries with dozens of joined tables. We note that we get a self-join for each column referenced. This work was not completed at the submission deadline.

Since the features discussed were first implemented within days of the submission deadline, no tuning or adaptation of the Virtuoso SQL was possible within the time limit, hence results are not anywhere near final and most interesting experiments had to be left out.

We intend to further study the comparative performance of SPARQL going to natively stored triples and compare this with SQL performance with single machine and clustered Virtuoso databases. One line of future work is benchmarking SPARQL and SQL based vertical storage schemes. We note that the RDF model is a vertical storage scheme almost by nature. Declaring that triples with given predicates be stored apart, in a table that keeps only subject and object results in a column-oriented store.

We further intend to broaden the scope of the present example around TPC H by including more sources in the mapping. This will demonstrate that the same queries can be run without loss of performance on a number of similar but distinct relational database instances. Thus SPARQL does become a data integration tool that exceeds the capabilities of SQL views merging data from multiple sources, for example.

Conclusions

The work discussed here demonstrates the feasibility of querying existing rela- tional data through extended SPARQL without loss of performance or expres- sivity and without any modification to the relational data store in question. A skeptic might ask what the value of an alternate syntax for SQL is, when SQL is universally known and applied. We would point out that us bringing SPARQL on par with SQL for decision support queries is not aimed at replacing SQL but at making SPARQL capable of fulfilling its role as a language for integration.

Indeed, we retain all of SPARQL's and RDF's flexibility for uniquely identifying entities, for abstracting away different naming conventions, layouts and types of primary and foreign keys and so forth.

In the context of mapping relational data to RDF, we could map several instances of comparable but different schemas to the common terminology and couch all our queries within this terminology. Further, we can join from this world of mapped data to native RDF data, such as the data in the Linking Open Data project. For example, we could join regional sales data to the US census data set within a single query.

Once we have demonstrated that performance or expressivity barriers do not cripple SPARQL when performing traditional SQL tasks, we have removed a significant barrier from enterprise adoption of RDF and open data.

References

1. W3C RDF Data Access Working Group: SPARQL Query Language for RDF: http://www.w3.org/TR/rdf-sparql-query/
2. Transaction Processing Performance Council: TPC-H a Decision Support Benchmark. http://www.tpc.org/tpch/
3. Orri Erling, Ivan Mikahilov: Adapting an ORDBMS for RDF Storage and Mapping. Proceedings of the First Conference on Social Semantic Web. Leipzig (CSSW 2007), SABRE. Volume P-113 of GI-Edition - Lecture Notes in Informatics. Bonner Kollen Verlag, ISBN 978-3-88579-207-9