Directions and Challenges for Semdata

Orri Erling
OpenLink Software
10 Burlington Mall Road, Suite 265,
Burlington, MA 01803, U.S.A.
oerling@openlinksw.com

ABSTRACT

The data driven world is here. What will RDF's place therein be? In this paper, we explore the inevitable coming together of RDF, analytic oriented databasing, web scale computing and reasoning. To warehouse, analyze, publish and subscribe to anything anywhere, draw conclusions and distill knowledge. This is the challenge.

As a first part of a response, we show how we attain a space consumption of just under 6 bytes per quad of RDF data extracted from the archetypal TPC H benchmark data set and how we get comparable query times with the data stored in relational and RDF forms in Virtuoso. This is achieved with a new column-wise compressed storage subsystem to be at the core of later Virtuoso releases. This represents a more than fourfold improvement over previous RDF space efficiency. While RDF scalability in absolute terms has been resolved with the advent of many cluster capable RDF stores, the advances outlined here radically reduce the cost of RDF when compared with its technological alternatives.

We then look at ways of synchronizing RDF data sets across wide area distances and at ways of keeping RDF extractions of local relational data up to date.

We conclude with a preview of research to be undertaken in LOD2, a EU FP7 project starting in the fall of 2010. Among other topics, this will address RDF exploitation of many relational techniques pioneered at in MonetDB at CWI, one of the project partners.

On the basis of all this, we are confident that RDF will strengthen its role in the coming data economy.

1. INTRODUCTION

This paper is divided in four parts. We first discuss the general challenges encountered by semantic data management, specifically in relation to the database world. We then show progress towards RDF to relational performance parity in the context of a column-wise storage model recently developed for OpenLink Virtuoso. We then discuss means of keeping RDF collections synchronized with other RDF collections or relational sources from which they are extracted. Finally, we outline some research directions that will be explored in LOD2, an EU FP 7 research project slated to start in the fall of 2010. As of this writing the project is awaiting final signature by the EC.

2. ON RDF AND DATABASE

The database and RDF communities have traditionally been largely separate, with some database cross-overs into RDF[16][2]. The present VLDB workshop marks a welcome and much needed rapprochement.

The triple is not the exclusive property of the RDF world. With other Entity Attribute Value / Class Relationship (EAV/CR) models such as Microsoft Odata and Google Gdata, the convergence and inter-operation of RDF and these models is both necessary and inevitable. With Linked Open Data enjoying continued adoption for structured information publishing and interchange, the data analysis game is changing.

In addition to an organization analyzing its own data in its own schema for operational insight, we find increasing opportunities for enriching private data with open source data. Not only is the volume growing but so is the diversity of schema, driving a demand for richer and more explicit semantics. The database and semantic worlds meet.

We have seen platform as a service, PaS, software as service, SaS, and are now beginning to see data as service, DaS. The dream career of every age is a rare skill that creates value from something that is plentiful in the environment. Today this skill is making sense of data.

To seize this opportunity, the RDF world needs to become database-credible. This involves comparable efficiency with alternative designs, with competitive edge brought by richer semantics and querying together with vocabulary reuse. This also involves mastering the data publishing and consuming game. If extract-transform-load (ETL) was and still is an enterprise-scale challenge, its DaS counterpart is a web-scale system of publish-subscribe streams among heterogeneous systems operating under separate management.

In subsequent sections we present some intermediate results on how we address these matters in OpenLink Virtuoso.

Looking further at the meeting of semantics and data, we might speak of a semantic mindset and of a data mindset; The data mindset is typified by Martin Kersten's statement to the effect that IFs inside loops kill performance, or Michael Stonebraker's 1998 statement that “no practical

Workshop on Semantic Data Management (SemData@VLDB) 2010, September 17, 2010, Singapore.

application of recursive query theory ... have been found to date."[17]. The semantic mindset would consider non-recursive structures as trivially uninteresting and would consider cache efficiency as "beyond the scope of the paper", while it is in fact the case that caching is the one reason why anything in computers works even as well as it does.

What is the basis of such apparent polarization when both camps are talking about data? Maybe the fundamental assumption of the database mindset is that we are dealing with large homogeneous sets with a separate schema, whereas the fundamental assumption of the semantic mindset is that we are dealing with a unique individual. In does not matter whether the individual is in a class or instance role, as it can be in both at the same time.

When these worlds converge, the semanticist discovers that the individuals in fact are alike enough for database precepts to apply, whereas the databaser discovers that not every element of the set is exactly alike. As an example of the latter, we have found row-by-row run time data typing of columns to be very useful.

But talking of the differences of semanticist and databaser attitudes is needlessly insular. Both camps find themselves in a much wider context.

The wider world of web scale witnesses the emergence of alternative database designs that no longer originate in the RDBMS community and of programming models such as map-reduce. The senior guard of database may regard these things as yesterdays news, "been there, done that," not entirely without justification. This is not however the mood of the web generation, for which these things solve actual issues.

The Berkeley Orders Of Magnitude (BOOM)[18][1] project has a very promising outlook on scalability in general, bringing together massive parallelism on cloud platforms and rule languages for tackling web scale problems. Massive data meets logic and distributed computing.

For implementors of parallel RDF databases, it is quickly clear that a system of distributed rules a la BOOM is not fundamentally different from distributed query execution. The same universal principles are encountered from so many different angles. This points the way to tighter integration between reasoning, querying and storage. Today's standalone paradigm of map-reduce appears in this setting as a special case that can be tackled inside a fundamentally more expressive system.

We see some beginning of integration in Vertica with map-reduce and map-reduce -like SQL extensions in Virtuoso and Greenplum. More development in this direction, now also encompassing declarative reasoning seems the way of the future.

3. IT'S ALL ABOUT SIZE

Optimization of I/O cost has for a long time been an important topic in databases, initially in the context of reducing disk I/O and more recently in optimizing traffic between CPU cache and RAM. The surest obstacle to performance parity between RDF and relational models is a radical difference in space consumption. It is true that disk is virtually free but it is even more true that the relative cost of disk access compared to memory based operations only keeps growing. This is the root of the problem and we believe that any solution must start with addressing this. Scalability of RDF storage is no longer a problem in absolute terms, as there are a number of cluster capable stores (Virtuoso[10], OWLIM[11], Bigdata[12], 4store[13] to mention a few). However, when data volumes get into tens of billions of triples, the cost of equipment becomes significant if one is to maintain a reasonable ratio or RAM and database size. Further, at today's prices, memory is the single most expensive part of a server for running databases. If RDF is to break out from its specialty niche and become an alternative for data integration and business intelligence in a broader sense, it needs to compare reasonably with alternatives. RDF is a needlessly hard sell as the data warehouse model if the equipment cost is 4 times that of the relational equivalent. The cost is non-trivial if we are talking tens or hundreds of terabytes.

In this section we show how TPC H data translated to RDF offers storage and query efficiency comparable with its relational form. The RDF purist might argue that this is unfair and that true RDF applications will not have the regular structure of TPC H. We argue in turn that much data is in fact regular, as it originates with relational sources, sensor networks or automatic information extraction. The point of RDF is that schema can evolve record by record, the point is not that each record will have a different schema.

3.1 RDF Indexing Row- and Column-Wise

Column-wise storage has for some time been the preferred storage architecture for relational analytical workloads. It is only intuitive to realize that RDF itself is a property-centric representation. From this there is only a short step to making the physical fact reflect this insight.

To this effect, we have built a column-wise storage engine to complement Virtuoso's existing row-wise representation. This will be a key component of Virtuoso 7, to be released at a future date. The column store can be applied equally well to relational and RDF data.

We begin by indexing RDF quads as two covering indices, first sorted on PSOG and the other sorted on POGS. We refer to the parts of an RDF quad by their customary abbreviations of (P)redicate, (S)ubject, (O)bject and (G)raph. These covering indices will efficiently answer queries where either PS or PO are specified, optionally filtering on G also. For queries where P is unknown, we do wish to avoid a full index scan but we do not wish to store the whole quad yet again. Hence we make sorted projections with distinct values of SP, OP and GS. These indices have much fewer entries than a covering index with all 4 columns would have and are therefore much more compact. For example, a query looking for quads with only G specified first looks in GS for S, then in SP for P, then in PSOG for the actual quad. There are extra join steps but considerable space is saved. Further, these indices do not enter into evaluation of typical relational-like queries, since in these the predicate is by definition specified. Therefore these indices will enter the working set only when queries make use of expressivity specific to the RDF model.

3.2 RDF Indexing in Virtuoso

In Virtuoso, the O column is typed at run time, hence small scalar values, e.g. all numbers and dates, are inlined in the index, thus supporting range lookups with the native collation of the data type. Strings are given unique integer ID's. This is done in part for saving space, in part because a unique ID for strings is needed anyhow when keeping a

Table 1: 1G TPC H Space Consumption

	Row-Wise	Column-Wise
Relational		
Tables	127077	80469
Indices	10389	10335
Tables w/o `l_comment` and `ps_comment`		45224
RDF		
PSOG index	282789	39670 + 168
POGS index	104684	39287 + 152
SP index	52314	20441 + 68
OP index	27767	7121 + 14
GS index	152	148 + 1
All 5 indexes	467706	107070
(bytes per quad)	*(25.96)*	*(5.94)*
All 5, if P specified	387473	79277
(bytes per quad)	*(21.50)*	*(4.40)*
IRI to string dictionary	43615	43615
String literal dictionary	8381	8381

The sizes are measured as counts of 8Kb pages, column-wise RDF numbers are column size + top level index.

full text index of these strings, which is the case in most Virtuoso deployments.

This index scheme has been implemented with both row-wise and column-wise stored multipart indices. A row-wise index entry uses key compression where 2 bytes are stored for an 8 byte value if there is another value of the same column on the page that is less than 16 units away. Thus most IRI ID's are stored as 2 byte values since these are often repetitive and fairly densely spaced. Further, if the last key part is an integer or IRI ID, a bitmap can be used for representing closely spaced values. The bitmap is either a physical bitmap or an array of integer deltas. Thus one index entry covers many entries which differ only in their last key part. Use of bitmaps accounts for the fact that the POGS index is smaller than the PSOG index.

The column-wise representation starts with a row-wise sparse index with one entry per every few thousand rows. The set of rows per entry of the top level index is called a segment. The columns in the segment are then stored contiguously in any number of compression entries. These compression entries are stored on pages, with each column of a segment consisting of an integer number of compression entries. Each compression entry specifies the compression. Compression types include run length, run length with deltas, integers with deltas, dictionaries, arrays of fixed length integers, arrays of variable length items. Compression entries like integer arrays and "run length + delta" enforce a homogeneous data type on the content whereas other formats like dictionary and variable length array allow for run time types.

In this way, there is no need to declare data types or preferred compression formats since the data at hand always determines what compression is applied. Also overheads associated with run-time typing are incurred only in the actually few cases where heterogeneous types occur next to each other.

Compression works best when contiguous values come from the same property, hence there is an extra benefit to having P as the leading key part in the RDF indices.

Relational column stores like MonetDB store each column of a table as a separate two-column table, called binary association table (BAT). This table maps a synthetic row id to the corresponding value of the column in question. Since the row ids are most often ascending sequences of integers, the ids themselves do not have to be stored. Our RDF scheme essentially replicates this structure, except that in the place of a single integer key with one dependent value we have a 4 part composite key. Three of these four key parts essentially fall away in compression. The P is generally a run length since it is the first ordering column and has a low cardinality. The S is either a bitmap or a delta + run length representation, depending on whether the property is single valued or multi-valued. The O is compressed as it would be in the relational column store. The G is again a run length because G generally does not vary. If we had a high cardinality of G, the column might not compress away but it still would not have to be read in memory unless the query actually specified it, which most queries in our experience do not.

Virtuoso as a relational column store does not favor storage as binary association tables a la MonetDB but rather uses sorted multi-column projections similar to CStore and Vertica. Such a projection is exactly like a row-wise table with the dependent part stored at the index leaf, except that now each row corresponds to a segment of several thousand column-wise compressed rows.

A central reason for using sorted multi-column projections instead of BAT's was the need to have multipart keys that can be partitioned on key column values, thus not on synthetic row id's. This is necessary in a cluster situation with RDF: If we know the predicate and subject, we wish to know based on these alone which partition stores the object, without needing to pass via a synthetic row id.

4. TPC H EXAMPLE

To illustrate space consumption with different storage formats, we take the TPC H[14] qualification database, scale factor 1, consisting of approximately 1Gb.

The row-wise and column-wise relational representations are loaded directly from the TPC H qualification data. The index schemes are TPC H compliant, with indices on `l_partkey`, `o_custkey` and `c_nationkey`.

The RDF representation of the same data is generated by mapping each table to a class and each column to a property. Each row of each table gets a unique IRI consisting of the table name and string representation of the primary key value(s). Additionally, foreign keys like `l_partkey` are given an extra IRI-valued property `has_part` which references the part by IRI. Thus the triple count is somewhat higher than the count of rows multiplied by columns, resulting in 147.5 million quads.

This mapping is then materialized as row-wise and as column-wise stored RDF. The RDF index schemes are identical, differing only in storage model.

The IRI strings and string literals are stored in separate dictionary tables. These are represented row-wise and shared by both row- and column-wise cases. A column-wise representation of these would approximately halve their space usage through use of stream compression for strings but we do not address this further here.

In Table 1 we see the space requirement for the four storage models. The numbers are given as counts of 8Kb database pages. This page counts reflect the amount of

RAM actually occupied when the data in question is fetched in Virtuoso's cache. We do not count unused disk pages since these do not occupy RAM. Unused pages are at times left for subsequent inserts.

We can compare the relational size without the indices with the RDF size for PSOG, as these represent the same information. On the RDF side you may add the string literal dictionary's size to the balance on the RDF side. The relational size for the indices could be compared to the RDF POGS figure. The POGS figure represents having a single part index on every column of every table whereas the relational index size is only for `l_partkey`, `o_custkey`, `ps_suppkey` and `c_nationkey`.

We see that the column-wise relational representation appears artificially large since the comment string columns are stored without compression. A figure excluding these columns is much better and reflects the working set of the queries, since the excluded `l_comment` and `ps_comment` are not referenced by any of the TPC H queries. Otherwise, `lineitem` is well compressed, with low cardinality or ascending columns like `l_orderkey`, `l_linenumber`, `l_orderstatus`, `l_returnflag`, `l_tax`, `l_discount` virtually falling away.

The row-wise RDF representation essentially takes every column value and stores it with 8 to 12 bytes of overhead, hence the very large size of the row-wise PSOG index. Most P, S and G are compressed from 8 to 2 bytes but the overhead is still great, plus there is a 2 byte overhead per each index entry. This state of matters is improved with POGS, since there we have an index entry only for each distinct POG, hence reducing overheads. The usually many values of S for each POG are an array of deltas or bits.

For column-wise RDF, we see P, S and G compress away and row overheads disappear. Therefore the column-wise PSOG plus string literal dictionary is in fact less than the column-wise relational representation. This comes from global removal of string duplicates.

There are two bytes per quad figures. These divide the total page count times page size for all indices by the count of quads. The first figure counts all indices, the second leaves out the distinct PS, OP and GS projections. The latter reflects the working set size when queries stay within a relational-like envelope.

We are not particularly concerned with the large IRI dictionary. This has no relational equivalent and is purely an added RDF cost but for query evaluation this will almost not be accessed, hence it does not spoil the working set. Queries with regexps on IRI's are a corner case without relational equivalent and we are not concerned with this here. A full text index on IRI strings or a tokenized representation as tried in MonetDB[19] is a valid solution. Further, stream compression on the string column will drop the size to about half.

5. EXECUTION ENGINE

A multiple tuple at a time execution model is typical for column stores[2]. The rationale is that locality will be found in accessing values of the same column on nearby rows instead of coming from accessing nearby columns on random rows. However, executing with multiple tuples at a time is just as well applicable to row oriented storage models.

Virtuoso first implemented execution on large batches of query variable bindings for the cluster situation, where this is indispensable for overcoming network latency. We later added execution also in single servers and within servers participating in a cluster. The benefit of this is exploiting locality between multiple accesses to nearby data, so that the random access overhead need be incurred only if the next datum to access is relatively far. There is an added cost coming from sorting the data to be random accessed before the random access but this is typically an in-cache operation and the savings from eliminating random access generally outweigh this cost.

The benefits of vectoring can be clearly seen with rows but are still more marked with columns, since the chance of having multiple accesses on relatively nearby data grows as the density of storage grows. Also the join operation on column-wise indices is specially written with the assumption of vectored usage. For random access on a column-wise index, we have a set of sorted sets of key values. The segment holding the first of these is located by binary search of a sparse row-wise B tree index. This is identical to random access of a row-wise index. After hitting the leaf, there is a segment of contiguous row stored column-wise. These are processed column by column, i.e. all matches of the 1st key part define a set of row ranges. Matches of the 2nd key part are then searched for within these ranges, until all specifications on key parts are exhausted. At this point, we have a set of n ranges of row numbers within the segment. After this there are no more matches of the search criteria within this segment and we move to the next set of search criteria, looking for the match on the same page of the row-wise sparse index. If the next row to access is within 100,000 or so rows, chances are that the segment will be referenced from the same page of the row-wise sparse index. If there is no match within the page or sibling pages of the row-wise index then we reset the search and start it from the top of the row-wise B tree.

This model makes column-wise random access more efficient than row-wise if the random access will hit more than one or two rows per segment, which translates to 1 row in every 1000 to 3000 rows.

We plan to make a detailed comparison of the TPC H workload with relational and RDF data models at a later stage. For now, we will simply show that the column compression benefits do not come at a cost in access performance and that Virtuoso can perform basic joins with RDF and relational versions of the data in roughly comparable time.

The sample query is

```
select count(*), sum(l_extendedprice*(1-l_discount))
from part, lineitem
where p_size < 20
and l_partkey = p_partkey option (order, loop)
```

The SPARQL version is

```
select count (*) sum (?ep * (1 - ?dis))
where {
  ?part tpch:size ?size .
  filter (?size < 20) .
  ?line tpch:has_part ?part .
  ?line tpch:lineextendedprice ?ep .
  ?line tpch:discount ?dis . }
```

The option `(order, loop)` clause means that join order is as in the from clause and that joins are loop joins. Both SPARQL and SQL plans scan the part size property and get `p_partkey`'s or part IRI ID's for RDF. Then an index

Table 2: Execution Times of a Sample TPC H Query

Data Layout	10K vector	100K vector
Relational rows	5887	4771
Relational columns	5971	2944
RDF rows	7458	7260
RDF columns	6372	6557

The times are in milliseconds. The CPU is Intel Xeon 5520.

on `l_partkey` is used for getting `l_orderkey`, `l_linenumber`, which are used with the primary key of `lineitem` for getting `l_extendedprice` and `l_discount`. With RDF, the `has_order` property is used for linking the part IRI to the IRI of the `lineitem`, for which the extended price and discount properties are fetched from PSOG. The plans are identical for row- and column-wise versions of the data. The query retrieves 2.2 million rows from `lineitem`, approximately 38% of the table.

The queries are run on a single thread in warm cache with a vector sizes of 10000 and 100000. This means that sets of up to 10000 or 100000 variable bindings are passed between query operators.

Increasing the vector size generally improves performance, up to the point where sorting the key values to search for costs more than the gain in accessing more rows per page. Column-wise formats benefit more from vectoring since retrieving a single row has more steps whose cost must be compensated for by getting more values per operation. The column-wise RDF time slightly increases when increasing the vector length, presumably due to CPU cache effects. This subject will be analyzed in more detail at a later date. To illustrate the importance of vectoring, we note that the RDF row-wise time without any vectoring is 12529 msec.

Again, a systematic analysis of the actual TPC H queries with relational and RDF models will be undertaken separately. The present numbers only show that comparable operations take comparable times. The column store implementation is relatively unoptimized, hence the numbers cannot be used for extrapolating relative performance of rows and columns in Virtuoso in general or for comparing Virtuoso with other column stores.

6. FURTHER OPTIMIZATIONS

The column store implementation measured here is typed at run time, hence does not have data type specific versions of join and other operations. These will be necessary for realistic performance comparison with other similar designs.

We note that indexing every predicate from object to subject is not always useful. For example for low cardinality properties like `l_orderstatus` or `c_nationkey` there is little point in making an index since a scan of the values on the PSOG key would be just as efficient.

Since the RDF model is very similar to a column-wise representation with binary association tables, the techniques for caching column to column joins developed in MonetDB are readily applicable.

Further, since the SP and OP projections of RDF quads only serve patterns where only a subject or object is known, it may not be necessary to fill these for every object or subject. Rather, these could be filled on demand, in a manner similar to cracking in [15].

It does not seem a priori necessary to project BAT-like subject, object tables from the quad table, as is for example discussed in [2]. The extra keys compress away and evaluating conditions on these takes negligible time since these are usually run length encoded. In this way we get the advantages of column storage for RDF without the need to adopt an application specific schema.

7. ADVANCES IN RDF PRODUCTION AND PUBLISHING

7.1 Hybrid Relational/RDF Schemes, Mixes of Federation and ETL

In a database context, RDF is primarily a means of integration, especially as concerns integration from sources not controlled by the integrator.

Thus most applications will map multiple local schemata into a common integrated schema. The method of mapping should be 1. flexible, 2. ex[expressive, 3. human-friendly and 4. optimizer-friendly, all four at the same time.

We have found that self-evident C sprintf format strings and simple ordered hierarchies of rules and exceptions let us satisfy all four conditions in 95% of all cases. The remaining 5% of complicated cases require hand-written stored procedures for conversion of individual values; this is less convenient but still tolerable for both budget and SPARQL/SQL optimizers.

The set of mapping rules for a group of tables is referred to as an RDF View. The SPARQL processor can translate a SPARQL request into SQL on the mentioned group of tables if the query contains triple patterns such that the output of an RDF View may match them.

During translation from SPARQL to SQL against a schema described in an RDF view SPARQL triple patterns become unions and groups of patterns become joins. Hence the most important optimization is rewriting of joins of unions into union of joins, with strong elimination of impossible joins. The RDF mapping tool absolutely must support this optimization in every possible way, otherwise it will work only on carefully chosen demos. More specifically, as soon as a predicate on the RDF side of the mapping may be mapped to more than one table/column on the SQL side, such elimination is a requirement.

The RDF mapping tool should pay attention to RDF-specific security, such as graph-level security that is between traditional table-level and row-level security of relational sources. For each triple pattern of a source query it should decide whether costly run-type permission checks should be placed in the generated SQL or whether inexpensive SQL table-level security is sufficient or whether the SPARQL optimizer should entirely block the access to some graphs at the compile time.

There is great benefit in controlling both the RDF to SQL translator and the target SQL dialect. Small extensions of the SQL syntax and the SQL run-time can seriously reduce both compilation and execution time. Examples are new SQL data types for IRIs and RDF literals with datatype or language or a BREAKUP operator that create many “short” output rows from each individual row of a wide input (e.g. all triples about an order made from one row of a join of `order` and `customer`). Custom SQL extensions do not preclude ultimately mapping to third party SQL dialects, but there is advantage in having an extensible intermediate SQL layer.

With good SPARQL and SQL optimizers, we can avoid the materialization of RDF in almost all cases, the inference is the biggest exception not yet eliminated. Fortunately, good SPARQL optimization lets the application developer to partially compensate the missing inference by adding more rules to the RDF view. E.g., if an RDF View is about lecturers and students and some external application queries for persons then the RDF View may make two extra mapping rules that will generate `?X rdf:type Person` when `?X rdf:type Lecturer` or `?X rdf:type Student` can be generated. For applications of average complexity, the SPARQL optimizer handles thousands of rules per RDF View, so a hundred of “inference replacement” rules does not make a great difference.

One does not have to choose between full translation from SPARQL to SQL and an extract transform load (ETL) scheme where queries are run on physical RDF data periodically ETL’d from a relational original. It is possible to reference a combination of physical RDF data and relational data mapped via RDF views in the same query.

It should be noted, however, that joining relational data of independent sources or joining relational and RDF requires extensive use of pairs of mutually inverse functions for converting between IRI’s existing as native RDF and IRI’s synthesized from relational key values. This is automated for most of C-style sprintf format strings but adds some handwriting for complicated cases. Of course, the SPARQL and SQL optimizers should know the list of inverse functions and understand how to rewrite a SPARQL query so that the target optimizer sees conditions that it can deal with. For example, one should generate conditions on columns and not conditions on expressions involving columns.

Local relational data can be mixed with RDF of different origin, including local resources, external RDF resources retrieved on demand and results of requests to remote SPARQL end points. IRI dereferencing is now in intensive use and the need for detailed service descriptions is growing fast.

As a simple example, consider a query that mixes TPC-H data with DBpedia and other LOD data. For each country that is now a member of EU, and for each sale to the country in question the query divides the sum of the sale by GDP values of the country in the year of sale, the sum of these results represent the relative impact of the company in the country relative to the “weight” of the country. The SPARQL optimizer should know that `country_id` and country name and country IRI in DBpedia are all unique and can be transformed to each other by bijections, and produce a good SQL that can be freely optimized. The SQL optimizer will decide where to start from: find all EU countries first or process all sales and check whether the sale is in EU country or do something else.

7.2 Keeping RDF Extractions Up To Date

Any reasonably good reporting tool can produce an RDF dump of a database by applying text templates for Turtle or NTriples syntax to output of some simple SQL queries.

If a mapping exists for relational data then it can be made better by SPARQL

```
construct { ?s ?p ?o } where { ?s ?p ?o }
```

and appropriate formatting procedure for output.

A similar SPARUL INSERT statement can store the extracted RDF in local RDF storage.

All these methods can convert the data once but offer no way of keeping data in sync, short of repeating the whole ETL process.

A fourth method has been added recently. For a given collection of RDF Views, one can create an initial RDF materialization with SPARUL and at the same time create triggers on the source tables for keeping the RDF up to date. For this to work, there must be a copy of the relational data in Virtuoso but this can be kept up-to-date with Virtuoso’s heterogeneous incremental replication capability or other federated SQL techniques. Future work will improve the efficiency of the generated triggers but even now this approach works well in many cases. This will never be the method of choice for fast-changing OLTP sources, though. Among other things, OLTP availability conflicts with the need for human-friendly and interoperable IRIs that may contain, e.g., a company name instead of unique internal `supplier_id`. Re-naming of a popular supplier will trigger cascading changes in numerous triples about related orders etc. — a nightmare typical for a heavily denormalized database.

7.3 Replicating RDF - Log Shipping and Sync With Ad Hoc Diffs

Almost all publicly available RDF data are distributed as archives with periodic updates. As data sets grow, processing of such archives becomes annoying for both publisher and subscribers. Fortunately, other methods become available.

Reasonably efficient analogs of textual “universal diff” and “patch” exist for RDF graphs. See for example [20]. The publisher can reduce traffic by publishing a “release” and “patches” between the “head” and the “release” versions of the data. The used algorithm is not universal, it works only for graphs without blank nodes or for graphs where blank nodes are identifiable by their properties or by the subjects they belong to. However these cases cover most of “industrial” cases — from big dumps like dbpedia (they’re free from blank nodes) to collections of FOAF data (they do not contain blank nodes that are connected only to other blank nodes).

The algorithm can pay attention to the declared ontology of the data being compared when a diff is generated. This can make the diff algorithm faster and the resulting patch can be safely applied to a different version of the graph, with accurate diagnostics of conflict. It becomes possible to merge independent changes of a common original graph. It becomes possible to handle subscribers’ amendments in an automated way. The diff-and-patch approach cuts the traffic costs but the lag of publishing is still significant, because making diffs takes time, in linear proportion to the size of the data in the base case.

To keep data fresh, transactional replication of RDF data is now available. The publisher logs all changes in RDF data in a replication log that is immediately available to subscribers and stored while not delivered to all subscribers. If the log grows faster than some subscriber can read and process it, the subscriber may disconnect and retry the synchronization later. This is the preferred means of replication for maintaining read-only copies of data to which most updates originate centrally.

RDB-to-RDF extraction triggers and RDF-to-RDF replication are two thirds of a bridge that might connect relational databases with different schemas. The missing part is an RDF-to-RDB routine that matches RDF changes with patterns specified by an RDF View, group inserted and deleted triples by unique keys of appropriate tables, checks recognized changes for consistency and completeness, makes changes in relational tables and reports triples with rejected changes and/or data not expressible in the destination database schema. The required basic algorithms are known for years, but the implementation will be useless while not backed by solid infrastructure for writing and debugging RDF Views.

8. THE LOD2 VISION

The LOD2 project is a European cooperation between industry and academia for taking the Linked Open Data vision and culture to its next evolutionary level. The project has significant EU funding under the 7th framework program and is slated to run from fall 2010 to fall 2014. We do not give here a full description of the project but rather focus on its database related aspects. The project is coordinated by DBpedia[21] pioneer University of Leipzig with participation from Free University of Berlin, DERI, Wolters Kluwer, Exalead, Open Knowledge Foundation and others. OpenLink Software (Virtuoso) and CWI (MonetDB) represent the database side.

8.1 Benchmarks

In the experience of the authors, benchmarks and the open competition arising around them are one of the central drivers of technological advance. In this sense, individual progress depends on progress by the community. On the other hand, when developing and supporting a commercial product, one hardly ever finds adequate time to experiment with competing products, even though such experimentation has been found most informative.

LOD2 will undertake a systematic review of present RDF stores and leading analytics oriented RDBMS's. A new RDF benchmark will be developed, addressing the shortcomings of the present BSBM[22] and LUBM[23]. Also RDF adaptations of relational benchmarks like TPC H[14] and SSB[24] will be explored in the interest of having a baseline for contrasting RDF against relational, to the extent the applications overlap.

8.2 Parallelism and Reuse of Intermediates

As pointed out earlier, much of the CWI work on column store optimization (such as recycling[3] or reconstruction[15]) is adaptable to RDF, in part thanks to RDF's property centric nature. Mixing the traditionally separate phases of query planning and execution, as in ROX[4], is highly relevant to RDF, since relational database statistics, even if these involve dynamic sampling, fail to exploit correlations between data.

Even with the advances in compression discussed in the present paper, large RDF deployments will have to run on distributed memory clusters. Adapting the single server versions of adaptive query execution as in ROX to parallel databases, as well as maintaining distributed caches of intermediate results will pose new scientific challenges to be explored in LOD2. Most importantly, RDF is not only a schema-last variant of the relational model but has a rich reasoning dimension. For reasoning at large scale to be economical, we can no longer take it as a given that all implicit knowledge will be explicitly materialized by the reasoning process. For efficiency, we would also rather not backward chain the implied knowledge every time it is requested. Thus a demand driven hybrid approach suggests itself. This can build on the experience of cracking and recycling, except that now the operations being cached and maintained across updates are much more complex than indexing and joining.

8.3 Data Integration

Linked data as such is is plentiful but linkage remains lacking: Data sets link to other data in the same set but links between data sets are sparse even when the real-world entities described by the data sets are the same. LOD2 will therefore develop automatic and semi-automatic means of tying the islands making up the Linked Open Data cloud closer together. This builds on experience with things like DL-Learner[25] and related work.

8.4 Discovery and Visualization

The field of computer science appears to have been haunted almost from its inception by two persistent mirages: 1. Producing applications from data definition and 2. enabling a non-technical user with limited knowledge of the data at hand to formulate complex ad hoc queries and generally to make sense of a data set.

In the present data driven world, specially with the advent of linked data and the unprecedented proliferation of integrable sources, the second of these ever elusive aims is becoming even more central. LOD2 will build on the experience of Ontowiki, Dbpedia and OpenStreetMaps visualization experiences for catalyzing rapid development of derivative data products, i.e. mesh-ups. Naturally, the advances in warehousing, querying, coreference resolution and schema alignment will directly support this.

9. CONCLUSIONS AND CHALLENGES

We have shown how we can bring down RDF storage and processing cost to near-relational levels. Further optimization is a matter of engineering, of which there is much experience, the conceptual back of the problem is broken.

Together with this, we have shown how we can keep RDF in sync across RDF sources and relational databases. This is instrumental in making RDF a valid currency in the data as service economy.

We have outlined further research directions for bringing intelligence close to the data and for eliminating needless repetition in a constantly more complex and expressive data refinement pipeline.

Several challenges however remain: How to make the data human understandable and discoverable? How to drive down the cost of integration quality even while quantitative barriers to integration are being overcome? When all points to tighter coupling of database, distributed computing and reasoning, how to avoid a new generation of closed behemoths? How to formulate an interface between data and reasoning such that the indispensable locality and integration is achieved while still retaining the benefits of interchangeable parts?

In the previously mentioned Declarative Imperative keynote, Joseph Hellerstein repeatedly raised the issue that speedup in hardware would no longer feed progress in computing performance. The hardware growth having turned towards parallelism, software engineers would now need to carry their share of the load. Breaking the performance barriers around use of semantic technology will play its part in the software community shouldering its share of the burden.

10. REFERENCES

[1] Hellerstein, Joseph M. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. PODS-2010, keynote. UCB/EECS-2010-90. http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-90.html

[2] Abadi, Daniel J. Query Execution in Column-Oriented Database Systems. MIT PhD Dissertation, February, 2008. http://cs-www.cs.yale.edu/homes/dna/papers/abadiphd.pdf

[3] Milena, G., Ivanova, Martin L., Kersten, Niels J., Nes Romulo A.P. Goncalves. An Architecture for Recycling Intermediates in a Column-store. SIGMOD-2009. http://homepages.cwi.nl/m̃k/onderwijs/adt /lectures/lecture5.pdf

[4] Kader, R.A., Boncz, P., Manegold, S., van Keulen, M. ROX: Run-time Optimization of XQueries. SIGMOD-2009. http://www-db.informatik.uni-tuebingen.de/files/research/pathfinder/publications/rox.pdf

[5] Vertica Systems. Vertica Database for Hadoop and MapReduce. http://www.vertica.com/MapReduce

[6] Cooper B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A. et al. PNUTS: Yahoo!s Hosted Data Serving Platform. Proc. of the VLDB Endowment, 2008-Aug., Vol.1, 1277–1288. ISSN:2150-8097.

[7] Project Voldemort — A distributed database. http://project-voldemort.com/

[8] Ceri, S., Gottlob, G., Tanca, L. Logic Programming and Databases. in: Surveys in Comuter Science. Berlin: Springer 1990

[9] Neo4j.org. Neo4j open source nosql graph database.

[10] Erling, O., Mikhailov, I. Faceted Views over Large-Scale Linked Data. Linked Data on the Web (LDOW2009). http://events.linkeddata.org/ldow2009/papers/ldow2009_paper3.pdf

[11] Ontotext AD. OWLIM — OWL Semantic Repository. http://www.ontotext.com/owlim/index.html

[12] SYSTAP, LLC. Bigdata. http://www.systap.com/bigdata.htm

[13] Harris, S., Lamb, N., Shadbolt, N. 4store: The Design and Implementation of a Clustered RDF Store. SSWS2009. 4store.org/publications/harris-ssws09.pdf

[14] Transaction Processing Performance Council. TPC Benchmark(TM) H (Decision Support) Standard Specification Revision 2.11.0. http://www.tpc.org/tpch/spec/tpch2.11.0.pdf

[15] Idreos, S., Kersten, M.L., Manegold, S. Self-organizing tuple reconstruction in column-stores. SIGMOD 2009. http://homepages.cwi.nl/ĩdreos/IKM_SIGMOD09.pdf

[16] Weiss, C., Karras, P., Bernstein A. Hexastore: sextuple indexing for semantic web data management. Proc. of the VLDB Endowment archive. Vol.1, 1 (August 2008). pp. 1008-1019. ISSN:2150-8097. http://www.adsafemedia.com/pdf/Sextuple_Indexing_for_Semantic_Web_Data_Mang.pdf

[17] Hellerstein J. M., Stonebraker M. (eds). Readings in Database Systems, Third Edition. Morgan Kaufmann, Mar. 1998.

[18] BOOM — Berkeley Orders of Magnitude – Declarative Languages And Systems. http://boom.cs.berkeley.edu/

[19] Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S. Column-Store Support for RDF Data Management: not all swans are white. VLDB 2008. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.6138&rep=rep1&type=pdf

[20] Tummarello, G., Morbidoni, C., Bachmann-Gmr, R., Erling, O. RDFSync: Efficient Remote Synchronization of RDF Models. ISWC / ASWC 2007. pp. 537-551.

[21] Auer, S., Lehmann, J. What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content. In Franconi et al. (eds), Proceedings of European Semantic Web Conference (ESWC 2007), LNCS 4519, pp. 503517, Springer, 2007.

[22] Bizer, C., Schultz, A. Berlin SPARQL Benchmark (BSBM). www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/

[23] Guo, Y., Pan, Z., Heflin, J. LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics 3(2), 2005, pp. 158-182.

[24] O'Neil, P., O'Neil, B., Chen, X. Star Schema Benchmark, Revision 3, June 5, 2009 www.cs.umb.edu/ poneil/StarSchemaB.PDF

[25] Lehmann, J. DL-Learner: Learning Concepts in Description Logics. Journal of Machine Learning Research (JMLR), 2009