Column-Store Support for RDF Data Management: not all swans are white

Lefteris Sidirourgos
CWI, Amsterdam
The Netherlands
lsidir@cwi.nl

Romulo Goncalves
CWI, Amsterdam
The Netherlands
goncalve@cwi.nl

Martin Kersten
CWI, Amsterdam
The Netherlands
mk@cwi.nl

Niels Nes
CWI, Amsterdam
The Netherlands
niels@cwi.nl

Stefan Manegold
CWI, Amsterdam
The Netherlands
manegold@cwi.nl

ABSTRACT

This paper reports on the results of an independent evaluation of the techniques presented in the VLDB 2007 paper "Scalable Semantic Web Data Management Using Vertical Partitioning", authored by D. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach [1]. We revisit the proposed benchmark and examine both the data and query space coverage. The benchmark is extended to cover a larger portion of the query space in a canonical way. Repeatability of the experiments is assessed using the code base obtained from the authors. Inspired by the proposed *vertically-partitioned* storage solution for RDF data and the performance figures using a column-store, we conduct a complementary analysis of state-of-the-art RDF storage solutions. To this end, we employ MonetDB/SQL, a fully-functional open source column-store, and a well-known – for its performance – commercial row-store DBMS. We implement two relational RDF storage solutions – *triple-store* and *vertically-partitioned* – in both systems. This allows us to expand the scope of [1] with the performance characterization along both dimensions – triple-store vs. vertically-partitioned and row-store vs. column-store – individually, before analyzing their combined effects. A detailed report of the experimental test-bed, as well as an in-depth analysis of the parameters involved, clarify the scope of the solution originally presented and position the results in a broader context by covering more systems.

1. INTRODUCTION

This paper has been written in response to the challenge put forward by the organizers of VLDB 2008 and the VLDB endowment to promote independent evaluation of previous papers. For this, we have chosen the paper titled "Scalable Semantic Web Data Management Using Vertical Partitioning", authored by D. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach as a focal point [1]. It is among the first papers in the database community reporting on the effects of modern database engines on scalable RDF data management.

The authors of [1] review the prevalent approaches to implement an RDF storage scheme using existing relational engines. In particular, they compare the *triple-store* approach pioneered in systems such as Sesame [3] to a *property table* approach proposed in [4, 9]. Performance and scalability of both RDF storage schemes are analyzed using a real-life application based on a core table of more than 50 million triples.

Their results and contributions can be summarized as follows:

- To address the limitations of the triple-store and the property table approach, the authors propose a different physical organization technique for RDF data. They *"(...) create a two-column table for each unique property in the RDF dataset where the first column contains subjects that define the property and the second column contains the object values for those subjects"*. This technique can be thought of as a *vertically partitioned* database on property value. We refer to this RDF storage solution as *vertically-partitioned*.

- Inspired by a real-life application scenario, a new benchmark is designed to assess the behavior of RDF storage solutions using a fixed set of queries.

- Using PostgreSQL the authors show that both property table and vertically-partitioned solutions outperform the standard triple-store solution by more than a factor of 2 and have superior scaling properties. Moreover, the authors argue that the vertically-partitioned RDF storage is simpler to implement than the property table.

- The vertically-partitioned approach is implemented on a column-store DBMS, namely C-Store [8], where *"another order of magnitude performance improvement is observed, with query times dropping from minutes to several seconds."* and *"(...) showed that on a version of the C-Store column-oriented database, it is possi-*

VLDB '08, August 24-30, 2008, Auckland, New Zealand

ble to achieve a factor of 32 performance improvement over the current state of the art triple store design."

In line with the charter of the Experiments and Analysis VLDB-track, we embark upon a faithful re-examination of the techniques proposed and re-execution of the performance experiments. The evaluation is focused on the quantitative results obtained from new implementations of triple-store and vertically-partitioned approach on both a row-store and a column-store system. We do not analyze the property table dimension, which requires amongst others an evaluation using database design wizards.

The predominant questions driving this study are "what is the scope of the benchmark?" and "does the performance of the vertically-partitioned approach carry over to more systems?".

The benchmark proposed and used in [1] is based on a real application. This way both data and queries are generally accessible and provide an easy to use yardstick for RDF storage schemes. From a design space perspective we study the extent to which it also provides a canonical basis for the RDF query space. This step leads to an extension of the benchmark by one more query to cover a part of the query space previously left out and to assess the flexibility of the RDF storage solutions.

The code base for the repeatability of the original experiments was obtained from the authors. It should be noted that C-Store is a layer over BerkeleyDB and all queries are hardwired in C++ code. This low-level approach prohibits us from extending the query set or implementing other RDF storage solutions without major resource investments. To this end, we employ MonetDB/SQL [6], a mature fully-functional open source column-store. The limited capabilities of PostgreSQL to experiment with efficient clustered bulk access to base tables, drives our choice to include instead a well-known – for its performance – commercial row-store DBMS, in the sequel referred to as *DBX*.

We implement the experiments on both MonetDB/SQL and DBX. The meticulous report of the experimental testbed and the in-depth analysis of the parameters involved can help to clarify the different aspects of the proposed RDF storage solutions.

Our contributions can be summarized as follows:

- The trends observed for C-Store (and PostgreSQL) are confirmed.
- The original observation in [1] that the vertically-partitioned approach outperforms triple-store when both are implemented in a row-store engine cannot be substantiated in general.
- For the given benchmark, the vertically-partitioned approach outperforms triple-store when both are implemented in a column-store. Moreover, our experiments show that the processing efficiency of column-stores is particularly suited for RDF data management applications.
- We point out potential scalability problems for the vertically-partitioned approach when the number of properties in an RDF dataset is high.

The re-evaluation of techniques and software from previously published results is a sign of maturity of computer science as a scientific discipline. The landscape of techniques for efficient data management is well carved out and fundamental changes are becoming rare. The step taken to re-evaluate results leads to a deepening of our understanding of the techniques and their applicability. It calls for a standing on each others' shoulders, which brings with it an attitude and drive to make the input for the experiments publicly available. Therefore, all additional experiments reported here and the complete software base is available from the authors.

The remainder of the paper is organized as follows. In Section 2 we revisit the benchmark proposed in [1]. In Section 3 we repeat part of the original experiments on different platforms. Section 4 re-runs all experiments and the extended versions of them using both MonetDB/SQL and DBX. We summarize our results and conclude the discussion in Section 5.

total triples	50,255,599
distinct properties	222
distinct subjects	12,304,739
distinct objects	15,817,921
distinct subjects that appear also as objects (and vice versa)	9,654,007
strings in dictionary	18,468,875
data set size	1253 M.Bytes

Table 1: Data set details

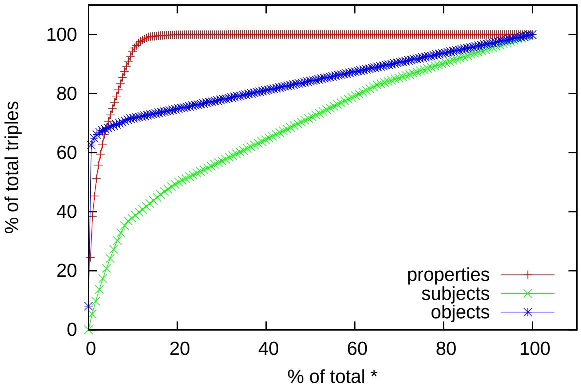


Figure 1: Cumulative Frequency Distribution graphs for properties, subjects and objects

2. BENCHMARK IN DETAIL

Easy to use benchmarks, with a clear embedding in real life applications, form a good yardstick to assess a technical solution. From a scientific perspective, a query benchmark should cover a sizable part of the query design space, preferably encapsulated in a limited set of canonical queries, which can be glued together to issue arbitrarily complex challenges to a system. Likewise, the data distribution should be made explicit to ease clarification of the results obtained.

In this section, we review the RDF benchmark proposed in [1] for performance evaluation of an RDF storage solution. Our analysis of the structural characteristics of both the RDF data set and the queries, reveals useful insights to significantly improve the benchmark for the task at hand.

Triple Patterns	
p1	(s, p, o)
p2	$(?s, p, o)$
p3	$(s, ?p, o)$
p4	$(s, p, ?o)$
p5	$(?s, ?p, o)$
p6	$(s, ?p, ?o)$
p7	$(?s, p, ?o)$
p8	$(?s, ?p, ?o)$

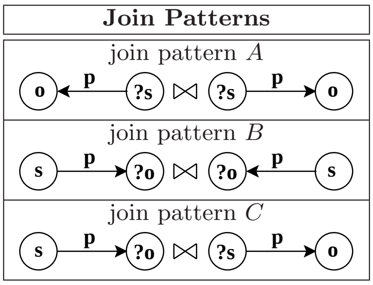


Figure 2: Simple RDF query patterns

Query	Pattern Coverage	
	Triple	Join
q1	p7	–
q2	p2,p8	A
q3	p2,p8	A
q4	p2,p8	A
q5	p2,p7	A, C
q6	p2,p7,p8	A, C
q7	p2,p7	A
q8	p6,p8	B

Table 2: Coverage of the query space

2.1 Data Set

The data set is taken from the publicly available Barton Libraries data set [2]. There are more than 50 million triples in this data set and more than 18 million distinct RDF classes and literals. Table 1 summarizes the numbers obtained by counting over different aspects of the data (i.e., triples, strings, etc.).

Figure 1 illustrates the cumulative frequency distribution of properties, subjects, and objects over the total population of triples. The most frequent property, namely `#type`, appears in 12,327,859 triples, while the top 13% of the total properties account for the 99% of all triples.

Subjects follow a more uniform distribution than properties. The most frequent subject appears in only 3,794 triples; this frequency drops to below 1,000 after the top 10 most popular subjects, reaching 100 already after the first 97 subjects.

The most popular object accounts for 8% of the total triple population. It is the object `#Date`, which appears in 4,035,522 triples, all of them being of the form <`subject #type #Date`>. The next 8 most frequent objects, each ranging from 1,824,082 to 1,008,697 occurrences, also appear as objects for the property `#type`.

The frequency distribution illustrates the highly Zipfian skew of the properties. Queries should be designed such that the predominant characteristics of this distribution is explored. The three distributions are sufficiently distinct to consider each as a potential candidate to be supported by an index.

The distribution figure tempts to focus experiments on the left part only, since there the frequency quickly changes. The most frequent items are also likely candidates as a first target to zoom into as an area of interest. However, from an implementation perspective, the long tail of less frequent items calls for efficient handling of large numbers of small sets; i.e., the least frequent properties lead to a multiplicative increase of the number of columns, each with hardly any data associated. This increases the disk storage requirements, stresses the flexibility of the catalog system, and may overload the optimizer with very complex queries. We come back to this issue in Section 4.4.

2.2 Query Design Space

Most of the proposed RDF query languages are based on *pattern matching* using triples of the form (s, p, o)[1]. A triple is a statement about a subject s that has a property p whose value is an object o. A *simple triple query pattern* consists of a triple where any of the subject, property or object can be bound to a variable, denoted by $?s$, $?p$ and $?o$ respectively. The left table of Figure 2 depicts all 8 possible combinations of simple triple query patterns.

From a relational database perspective, these patterns can be combined into more complex patterns using join conditions. That is, the patterns (s, p, o) and (s', p', o') can be related using an equality condition in 6 different ways. Of these, the predicates $s = s'$, $p = p'$, and $o = o'$ are *semantically strongly typed*. The predicate $s = o'$ (or $o = s'$) can be seen as *semantic role change*. The remainder, $s = p'$, and $o = p'$ (or similarly $p = s'$, and $p = o'$) play a role in *semantic reasoning*, usually found on the RDF *Schema level* (RDF/S). For each combination we have 4 terms left that either are a target variable or constant. It leads to a total of $\frac{2^4 \times 6}{2}$ different patterns to consider for even the simplest queries.

Three different *join patterns* are shown in the right table of Figure 2. These join patterns are of interest because they form the *RDF data graph*. We adopt a standard graph interpretation to illustrate RDF patterns, where nodes are either subjects or objects (i.e., RDF classes) and edge labels denote properties. Edges are directed from subject to object. Join pattern A is a join on the subjects of two triples. Join pattern B is a join on the objects of two triples and join pattern C is a join on the object of one triple and the subject of the other.

Having identified the *query space* for the RDF query language, we can investigate the coverage of this space by the benchmark proposed in [1]. The benchmark covers a small fraction of the query space, namely only 7 queries. This is a direct consequence from the underlying real-life application; not all patterns are equally important. However, for scientifically sound conclusions, the challenge is to determine equivalence classes, i.e., patterns supported by an underlying implementation that exhibit identical behavior.

For a high-level description of each query, we refer the reader to [1]. Figure 3 depicts the graph interpretations for those 7 queries. For clarity, we have omitted the bow-tie symbol of the join operator. The shaded nodes and labels indicate the values returned by the query. For completeness we include the SQL code of all queries in the appendix.

Table 2 lists the queries and their coverage over the simple triple and join patterns (queries *q1* to *q7*). Most of the queries use join pattern A (i.e., a join between subjects), while there is no join between objects (i.e., join pattern B). Only 3 out of the 8 simple triple patterns are used. Patterns { *p1, p3, p4, p5, p6* } are all absent. Of these, pattern *p1*

[1] Including W3C's recommendation of SPARQL [7] and the SQL translations in the appendix of [1].

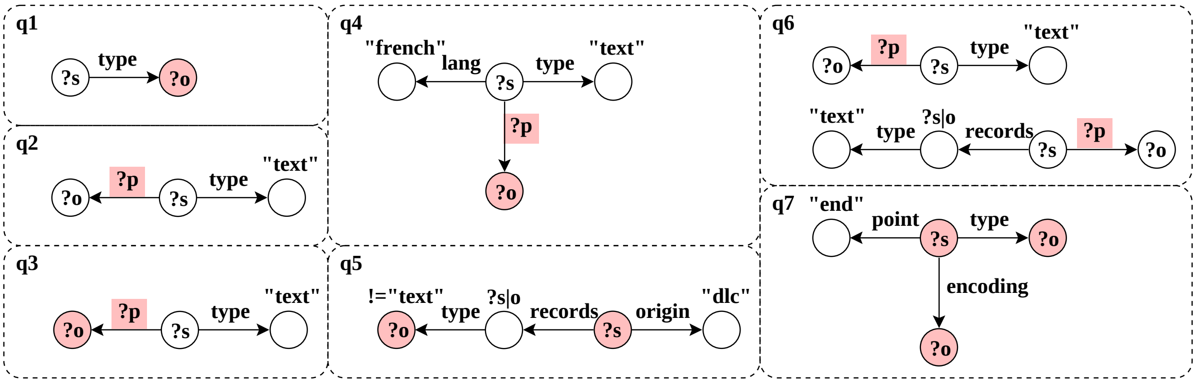


Figure 3: Graph Interpretation of queries q1 to q7

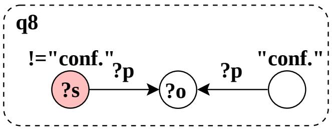


Figure 4: Graph interpretation of query 8

can be seen as a *point query*, which returns true if the triple belongs to the database. It is a basic composite key value lookup query, which should be present in every benchmark to highlight index support in the target system.

A similar conclusion can be drawn from the join coverage. To illustrate a possible extension of the benchmark, we introduce query *q8* using patterns *p6* and *p8* with a join on objects as depicted in Figure 4. A high level description of *q8* is that it returns all subjects that share the same objects with a defined subject (in this case `"conferences"`). These kinds of queries are common in RDF applications since they are part of a big set of inference rules that try to identify connections between different subjects. The SQL equivalent code for a triple-store is:

```
SELECT B.subj
FROM triples AS A,
     triples AS B
WHERE A.subj  = 'conferences'
  AND A.obj   = B.obj
  AND B.subj != 'conferences'
```

The SQL implementation of *q8* for the vertically-partitioned storage is discussed in Section 4.2.

The addition of one more query with join pattern *B* allows us to study the proposed RDF storage schemes by covering the query space in a slightly more canonical way.

2.3 Benchmark Conventions

Every benchmark definition requires a set of rules and definitions to enable comparison of results. This section explains the rules and definitions adopted in our study.

The database loading, clustering and index construction are all kept outside the scope of the benchmark. The rationale is based on the perceived predominantly read-only nature of an RDF store. The benchmark is comprised of the 8 queries above, which should all be run to completion. The timing is focused both on cold and hot runs.

To clarify these rules we define some terminology.

Cold run A cold run is a run of the query right after a DBMS is started and no (benchmark-relevant) data is preloaded into the system's main memory, neither by the DBMS, nor in filesystem caches. Such a clean state can be achieved via a system reboot or by running an application that accesses sufficient (benchmark-irrelevant) data to flush filesystem caches, main memory, and CPU caches.

Hot run The counterpart, a hot run, is defined as repeated runs of the same query without stopping the DBMS, ignoring the initial (semi) cold run.

Real Time The real execution time of a query is defined as the wall clock time passed between the server receiving the query and just before returning the results to the client, i.e., it measures the time spent on the server to read, parse, optimize and execute the query. For a "fair" comparison between stand-alone and client-server architectures, this definition purposely ignores client-server communication.

User Time The user time is related to the real time and involves the same boundaries, but measures (CPU-)time spent by the user-space application (here: the DBMS), excluding the (CPU-)time spent by the operating system.

3. EXPERIMENTATION REDO

In this section we report on our attempt to reproduce and verify the experimental results in [1]. We repeated the experiments both on PostgreSQL and C-Store, but focus our study on C-Store [8] since this is the fastest reported engine.

Our test-bed consists of two machines with different CPU architectures and I/O capabilities. Table 3 details their characteristics. Although the memory size is different, in both machines the data fits in memory during hot runs. The most notable difference is that machine *A* has 2 disks of striped raid (software raid, level 0) capable of reading data with a rate of approximate 100MB/s; while machine *B* has 10 disks of striped raid (software raid, level 5) and reads data with a speed of approximate 390 MB/s. For completeness and ease of comparison, we include the characteristics of the machine used in [1] as machine *C* in Table 3.

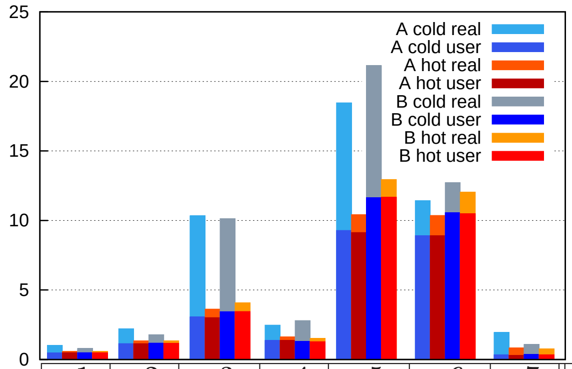
The code base of the original experiments has been obtained from the authors, including a short how-to-install

Machine	*A*	*B*	*C* [1]
Num. of CPU	1	2	1
CPU	AMD Athlon 64 Dual Core	Intel Xeon	Intel Pentium IV Hyperthreaded
CPU speed	2 GHz	3 GHz	3 GHz
cache size	512 KB	1024 KB	1024 KB
RAM size	2 GB	4 GB	2 GB
I/O read	100–110 MB/s	380–390 MB/s	150-180 MB/s
RAID disks	2	10	3
RAID level	0	5	0
Operating System	Fedora 8 (Linux 2.6.22)	Fedora Core 6 (Linux 2.6.23)	RedHat Linux

Table 3: Machine configuration

	Data read from disk	Number of rows returned
q1	100 MB	30
q2	135 MB	9
q3	175 MB	3336
q4	142 MB	297
q5	250 MB	12916
q6	220 MB	14
q7	135 MB	74866

Table 5: Data relevant to a query



	q1	q2	q3	q4	q5	q6	q7	G
🟦	1.01	2.21	10.33	2.47	18.46	11.42	1.94	4.2
🟦	0.47	1.14	3.06	1.37	9.28	8.91	0.34	1.8
🟧	0.59	1.33	3.63	1.62	10.42	10.36	0.83	2.3
🟥	0.49	1.14	3.01	1.37	9.13	8.91	0.30	1.7
⬜	0.79	1.79	10.13	2.80	21.13	12.71	1.09	3.8
🟦	0.49	1.18	3.44	1.30	11.64	10.56	0.37	1.9
🟧	0.59	1.35	4.08	1.52	12.95	12.04	0.77	2.4
🟥	0.49	1.17	3.45	1.28	11.67	10.49	0.34	1.9
[1]	0.66	1.64	9.28	2.24	15.88	10.81	1.44	3.4

Table 4: Repetition Results

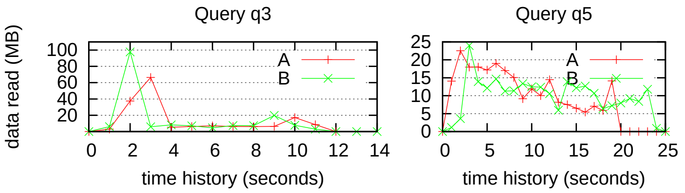


Figure 5: I/O Read history for q3 and q5

description. After fixing some minor engineering problems due to reliance on outdated environment settings (e.g., compilers), the code compiled and ran on our platforms. We compiled it with the GNU/g++ (ver. 3.4) using optimization level `-O3`.

Table 4 details the results of the repetition experiment, performing both cold and hot runs as discussed in Section 2.3. For each machine, we list the real and user time in seconds as reported by C-Store. The results represent the average timing observed over a run of 3 tests, in between cold tests zapping the memory completely. We do not report the standard deviation of the average time of each query, since the differences were less than 30 milliseconds. The last column of Table 4 (titled G) lists the geometric mean for all queries; the last row is taken from [1] and represents the original experiment of C-Store. Albeit small variations due to different machine architectures, the trends observed are in-line with the results presented in [1].

An interesting observation drawn from our first attempts to run the experiments is the impact of I/O management. Machine *B* has a much better I/O throughput capability, it can handle I/O 4 times faster than machine *A*, yet it does not materialize in a significant improvement in the timings. Moreover, despite the significant difference in pure CPU clock speed, the user times on both machines are very similar. In fact, they are slightly higher on machine *B* with higher clock speed, suggesting that the C-Store code compiled with GNU/g++ v3.4 using `-O3` runs more efficiently on the AMD CPU than on the Intel CPU. Triggered by these results, we investigate the actual code behavior using *iostat* and *top* (for memory consumption).

The amount of data transported from disk into memory shows the effects of a restrictive buffer space. The total database is not more than 270 MB on disk[2]. The queries read major portions (Table 5) and we assess that data is read multiple times as well. Figure 5 illustrates for the I/O dominant queries *q3* and *q5*, the amount of data read into memory during the complete run. These figures show that C-Store only exploits a small fraction of the I/O bandwidth. Pre-caching actions and better asynchronous I/O management would have significantly improved the performance.

C-Store is a database kernel in an early development stage, thus usable only to conduct micro benchmarks. It lacks the functionality of a mature database management system making it virtually impossible for a third party to implement, in reasonable time, other RDF storage schemes or even extend with new queries (e.g., the query plans in C-Store are hard-wired in C++ code). For these reasons, we were unable to extend C-Store with either query *q8* (introduced in Section 2) or with other RDF storage schemes like triple-store. The absence of an implementation of triple-store for C-Store in the original paper is a major drawback in our opinion, since there is no comparison between the vertically-partitioned approach and triple-store with both of them implemented in the same column-store engine.

In the next section, we continue our analysis and comparison of vertically-partitioned approach with the state-of-the-art RDF storage schema triple-store on MonetDB/SQL and DBX.

[2] C-Store is loaded with data associated with 28 properties, hence the small size.

4. THE “BLACK SWANS”

A key observation of [1] is that the vertically-partitioned RDF storage approach on a column-store outperforms a triple-store implementation on a row-store engine with a factor 32. In this section we place this claim in a broader perspective using the latest MonetDB/SQL [6] software release[3] and the latest version of DBX. We compare both triple-store and vertically-partitioned storage schemes. All our code is available for independent assessment[4].

4.1 Triple-Store Implementation

Triple-stores have been realized with relational technology for a long time [3, 4, 5, 10]. As for any database design with a priori known query patterns, the choice of indices, clustered and un-clustered, is central to achieving good performance. The authors of [1] implemented triple-store on PostgreSQL by defining three B+tree indices: one clustered on *SPO* (subject, property, object) and two un-clustered on *POS* (property, object, subject) and *OSP* (object, subject, property).

In order to be aligned with the original experiments, we implement the same set of indices on DBX. Since MonetDB/SQL does not include user defined indices, we rely on ordering the data on subject, property, object. However, clustering on *SPO* is not the closest equivalent to the clustering achieved by the vertically-partitioned approach, thus it entails an unfair comparison. The vertically-partitioned approach creates different tables, one for each property, and clusters data subsequently on *SO* (subject, object). The closest equivalent for triple-store is clustering on *PSO* (property, subject, object). In fact, mature B+tree implementations support key-prefix compression, thus in practice not storing the entire property column. Similarly, column-stores with compression (e.g., RLE or delta-compression) can achieve the same effect on the sorted property column.

Based on this observation, we also implement triple-store on DBX with a B+tree clustered on *PSO*. In addition, we define five more un-clustered B+tree indices on all other permutations of (property, subject, object). Having all index permutations allows DBX's optimizer to create more efficient query plans. With MonetDB/SQL, we realize the *PSO*-clustering by sorting the triples table on (property, subject, object).

Our SQL implementation for queries *q1* to *q7* follows that of [1], and for *q8* the one described in Section 2.2. The authors of [1] assume that *“(...) the Longwell administrator has selected a set of 28 interesting properties over which queries will be run (...). For queries Q2, Q3, Q4, and Q6, only these 28 properties are considered for aggregation.”*. The filtering of the 28 properties is achieved by populating a “`properties`” table with these property values and join it against the properties returned by *q2*, *q3*, *q4* and *q6*. We follow the same strategy in our implementation. In addition, we implement a version without applying this filtering. These queries are marked with an * symbol (i.e., *q2**, *q3**, *q4** and *q6**). The SQL code for all queries is given in the Appendix at the end of this paper.

4.2 Vertically-Partitioned Approach

The *vertically-partitioned* RDF data approach of [1] is a horizontal partitioning of the corresponding triples table clustered on property. Triples with the same property are grouped in the same partition, which results in as many partitions as properties. Each partition table is named after the property, which leaves two columns for subject and object. For the Barton Libraries data set this calls for 222 tables, many with just a small number of rows (less than 10).

The vertically-partitioned approach is implemented in both DBX and MonetDB/SQL. For MonetDB/SQL the data is sorted on (subject, object). For each table in DBX we define one clustered B+tree on *SO* (subject,object) and one un-clustered on *OS* (object, subject).

Query *q8* is implemented by first visiting the property tables and select objects for which the subject has value `"conferences"`. The union of all qualifying objects is stored in a temporary table t. Next, t is joined back with the property tables after filtering out all `"conferences"` subjects to avoid the join with the same triple twice. The final result is union-ed and the subjects are returned.

In our vertically-partitioned implementation using SQL, the relevant properties are iterated in the `from` clause. Thus, filtering the 28 properties selected by the Longwell administrator is achieved by including only those properties in the `from` clause.

For our full-scale implementation, where no restriction on the properties is introduced, this leads to a sizable SQL clause. Queries *q2**, *q3**, *q4**, *q6** and *q8* grow to a size that seriously challenges the optimizer of DBX, yielding a message analogous to “the generated plans might be sub-optimal due to the size of the SQL statement”. Each query contains more than two hundred unions and joins, since these are exactly the queries that have at least one property bound to a variable.

The exercise of implementing the storage and queries for the vertically-partitioned approach leads to the following observations:

- Since the logical schema of vertically-partitioned storage is data-driven, we had to implement a front-end to produce the correct SQL code. This is mainly due to the fact that SQL does not provide a mechanism to iterate over the tables in the `from` clause. Moreover, in case of an update in properties, the queries have to be re-produced. Here holds the general observation that data-driven logical schemes make queries susceptible to updates.

- The authors of [1], base their criticism against the property table storage solution on three points, one of which is:

  *“**Proliferation of union clauses and joins.** (...), queries are simple if they can be isolated to querying a single property table like the one described above. But if, for example, the query does not restrict on property value, or if the value of the property will be bound when the query is processed, all flattened tables will have to be queried and the results combined with either complex union clauses, or through joins. To address these limitations, we propose a different physical organization technique for RDF data.(...)”*

  However, we identify the same drawback for the vertical-

[3] We use stable version MonetDB 5.6.0 with SQL 2.24.0 from the publicly accessible MonetDB repository on SourceForge.
[4] Contact the authors.

ly-partitioned approach. If a query is not isolated to access a predefined number of properties, the SQL code becomes large and complex. It challenges the capabilities offered by most optimizers.

- Finally, one of the beneficial properties, presented in [1], for the vertically-partitioned RDF storage is:

  *"**Fewer unions and fast joins**. Since all data for a particular property is located in the same table (unlike the property-class schema), union clauses in queries are less common. And although the vertically partitioned approach require more joins relative to the property table approach, properties are joined using simple, fast (linear) merge joins."*

  If the property in a query is bound to a variable, then the rows returned from each property table must be union-ed. In the case where the property is not part of the result, then the union operator must also perform a duplicate elimination. Finally, since the data is not clustered on objects, a query which joins on objects (e.g., query *q8*), will not allow the use of a fast (linear) merge join.

4.3 Experimental Results

In any experimental observation, it is impossible to assert that "all swans are white", no matter how many white swans are observed. However, if at least one black swan is found, one may safely conclude that "not all swans are white". In this section, we embark on our quest to locate at least one "black swan" for each RDF storage solutions and for each database system.

All experiments are conducted on machine *B*, described in Table 3 of Section 3. We conduct experiments for both cold and hot runs, as discussed in Section 2.3. We report both the user time and real time. These times are obtained by the built-in timing mechanism provided by each system.

Table 6 reports on the results obtained with cold runs. Each query is run 3 times and we report the average time. To ensure that none of the relevant data pages reside in memory, between every two query runs we stop the DBMS server, clear the memory, and then restart the server.

Table 7 reports the results for hot runs. For both MonetDB/SQL and DBX we start the server, run the query once to load the relevant data, and then perform 3 measured runs without stopping the server or clearing the memory. We report the average of the latter 3 runs.

The queries marked with an asterisk (*) represent our full-scale experiment where all 222 properties are included in the aggregation. The G* column of Tables 6 and 7 indicates the geometric mean of all 12 queries. To draw the complete picture we also include the results obtained from C-Store in Section 3. Since not all queries are implemented in C-Store, we include also a G column presenting the geometric mean over the initial 7 queries, i.e., excluding *q8* and the asterisk versions of queries *q2*, *q3*, *q4*, and *q6*. Finally, the last column titled $\frac{G*}{G}$ reports the relative increase of the geometric mean G* compared to the geometric mean G, i.e., when moving from the initial set of 7 queries restricted to 28 properties to our extended set of 12 queries also considering all 222 properties.

A detailed reading of the execution times for cold runs in DBX leads to some surprising conclusions. An important observation is that the order of clustering is paramount to the triple-store implementation. Compared to the original proposal of clustering on *SPO*, our choice to cluster on *PSO* achieves a significant improvement.

For example, query *q1* improves by a factor of 5, queries *q2**, *q5* and *q6** by a factor of 3, and the remaining queries depict an improvement of a factor 2. By examining the query plans created by the optimizer of DBX, we notice the beneficial impact of the PSO clustering; the remaining indices have little impact. This result is backed up also by the fact that for queries *q2, q2*, q3, q3*, q6,* and *q6** the user time is close to the real time when *PSO* clustering is used, while for *SPO* the real time is almost twice the user time. In other words, DBX is spending half of the execution time waiting for the data to be retrieved from disk. This indicates that clustering on *PSO* achieves better I/O performance than clustering on *SPO*. However, query *q8* serves as the "black swan" for that last statement. It exhibits the same time with both *PSO* and *SPO*, apparently not benefiting from either clustering.

Vertically-partitioned execution times for cold runs in DBX draw a fuzzy picture. For queries *q1, q4, q5, q6,* and *q7*, which are restricted to (at most) 28 properties, vertically-partitioned storage performs better than triple-store: for *q1, q4,* and *q6* slightly better, for *q5* twice as well and for *q7*, 4 times better. However, the picture changes for the rest of the queries. For queries *q2, q2*, q3, q3*, q4*, q6*,* and *q8*, triple-store is faster than the vertically-partitioned approach, ranging from slightly faster to 9 times faster (i.e., *q4**). The increased number of joins and unions for these queries stresses the row-store optimizer and execution engine to its limits.

The general conclusion for a row-store engine is that the vertically-partitioned approach performs only slightly worse than triple-store, given that the correct clustering and indices are chosen for the latter. For both cold and hot runs, triple-store clustered on *PSO* exhibits a lower geometrical mean G* than vertically-partitioned. We consider this conclusion to be the "black swan" for the conclusion drawn in [1] regarding the performance of the vertically-partitioned approach when implemented in row-stores. However, the same cannot be entirely said for column-stores.

By examining the results of the MonetDB/SQL column-store engine, the vertically-partitioned approach achieves better times than either clusters of triple-store. This behavior can be explained if one compares the user and real time for cold runs of triple-store clustered on *PSO*. MonetDB/SQL spends a large portion of the total time on reading the triples table into memory. On the other hand, this is not the case for the vertically-partitioned approach, since only the property tables relevant to a query must be read into memory. This effect is easily noticeable for queries that are isolated to a few properties, i.e., *q1, q2, q3, q4, q5, q6,* and *q7*. However, the runtime overhead of reading data can be alleviated using a column-store that supports table compression along similar rows.

That said, the "black swans" are queries *q2*, q3*, q6** and *q8*. For these queries, triple-store sorted on *PSO*, or *SPO* for *q8*, exhibits better times. Mainly because, the overhead of reading all tables of the vertically-partitioned storage scheme and the cost of the increased number of joins and unions is bigger than the overhead of reading the triples table into memory. This observation becomes even more clear in the hot runs. Since reading data into memory is not an

	store	cluster	time	q1	q2	q2*	q3	q3*	q4	q4*	q5	q6	q6*	q7	q8	G	G*	$\frac{G*}{G}$
DBX	triple	SPO	real	12.59	53.65	108.76	50.35	144.81	16.08	13.82	45.06	127.45	170.99	9.62	19.45	31.4	40.8	1.3
			user	9.69	28.82	70.50	30.48	94.70	9.06	6.89	12.88	76.74	114.66	1.91	9.68	14.6	21.0	1.4
		PSO	real	2.35	34.08	37.93	39.73	72.72	10.64	9.84	14.01	54.66	60.66	8.62	19.61	15.5	20.9	1.3
			user	1.77	30.85	36.46	36.49	63.67	3.68	2.85	11.04	50.16	58.79	1.72	9.56	9.5	13.1	1.4
	vert.	SO	real	1.92	44.29	99.46	49.88	121.08	10.11	84.03	6.32	51.23	173.49	2.70	39.75	12.0	28.2	**2.4**
			user	1.57	40.62	73.56	46.27	95.80	6.34	14.63	5.78	47.01	154.67	1.24	8.37	9.3	17.5	**1.9**
MonetDB	triple	SPO	real	3.06	12.16	12.30	14.04	27.32	11.10	11.00	32.86	25.79	26.08	29.03	6.65	14.6	14.5	1.0
			user	1.26	2.96	3.16	4.7	16.52	1.48	1.712	2.83	6.67	6.21	2.07	3.76	2.6	3.3	1.3
		PSO	real	2.66	6.48	6.62	8.59	16.92	14.85	20.67	4.11	9.60	8.96	3.46	8.43	6.0	7.8	1.3
			user	0.72	2.32	2.40	3.83	10.89	2.09	2.30	1.21	3.90	3.95	0.21	4.50	1.4	2.2	1.6
	vert.	SO	real	1.20	3.50	9.16	5.22	19.34	2.28	6.22	2.00	7.20	16.58	0.61	7.99	2.3	4.4	**1.9**
			user	0.68	1.87	5.85	2.96	14.16	0.57	2.68	1.09	4.94	12.46	0.06	3.35	0.9	2.0	**2.2**
C-Store	vert.	SO	real	0.79	1.79	–	10.13	–	2.80	–	21.13	12.71	–	1.09	–	3.8	–	–
			user	0.49	1.18	–	3.44	–	1.30	–	11.64	10.56	–	0.37	–	1.9	–	–

Table 6: Experimental results for cold runs.

	store	cluster	time	q1	q2	q2*	q3	q3*	q4	q4*	q5	q6	q6*	q7	q8	G	G*	$\frac{G*}{G}$
DBX	triple	SPO	real	4.29	42.61	93.11	34.86	97.92	8.02	6.12	11.70	89.11	142.10	1.34	14.47	13.2	21.1	1.6
			user	4.29	33.31	68.88	34.16	95.11	8.02	6.10	11.68	74.96	120.36	1.27	10.58	12.3	19.0	1.5
		PSO	real	1.72	40.18	38.35	45.65	67.32	3.22	2.49	10.61	57.52	63.04	1.42	12.14	9.8	13.6	1.4
			user	1.72	40.17	38.35	45.64	66.85	3.22	2.47	10.60	57.33	63.03	1.34	8.02	9.7	13.1	1.4
	vert.	SO	real	1.55	39.62	74.85	45.17	94.59	6.12	14.18	5.69	45.57	154.81	1.25	11.55	9.1	17.7	**1.9**
			user	1.55	39.61	74.83	45.16	94.09	6.12	14.15	5.67	45.56	153.08	1.18	7.49	9.1	17.0	**1.9**
MonetDB	triple	SPO	real	1.53	3.50	3.63	5.28	17.54	1.68	1.98	2.77	8.37	7.33	1.82	4.76	2.9	3.7	1.3
			user	1.36	2.73	2.91	4.33	15.40	1.41	1.65	2.30	6.20	5.70	1.65	3.75	2.4	3.1	1.3
		PSO	real	0.78	2.80	2.83	4.36	12.59	1.70	1.97	1.44	5.67	4.59	0.18	5.23	1.5	2.4	1.6
			user	0.69	2.31	2.31	3.69	10.54	1.59	1.86	1.16	3.80	3.65	0.17	3.60	1.3	2.0	1.5
	vert.	SO	real	0.79	1.50	5.50	2.64	14.01	0.50	2.57	1.29	4.65	11.51	0.06	5.05	0.9	2.0	**2.2**
			user	0.68	1.44	5.20	2.52	13.25	0.48	2.40	1.03	4.40	11.23	0.06	4.20	0.8	1.9	**2.4**
C-Store	vert.	SO	real	0.59	1.35	–	4.08	–	1.52	–	12.95	12.04	–	0.77	–	2.4	–	–
			user	0.49	1.17	–	3.45	–	1.28	–	11.67	10.49	–	0.34	–	1.9	–	–

Table 7: Experimental results for hot runs.

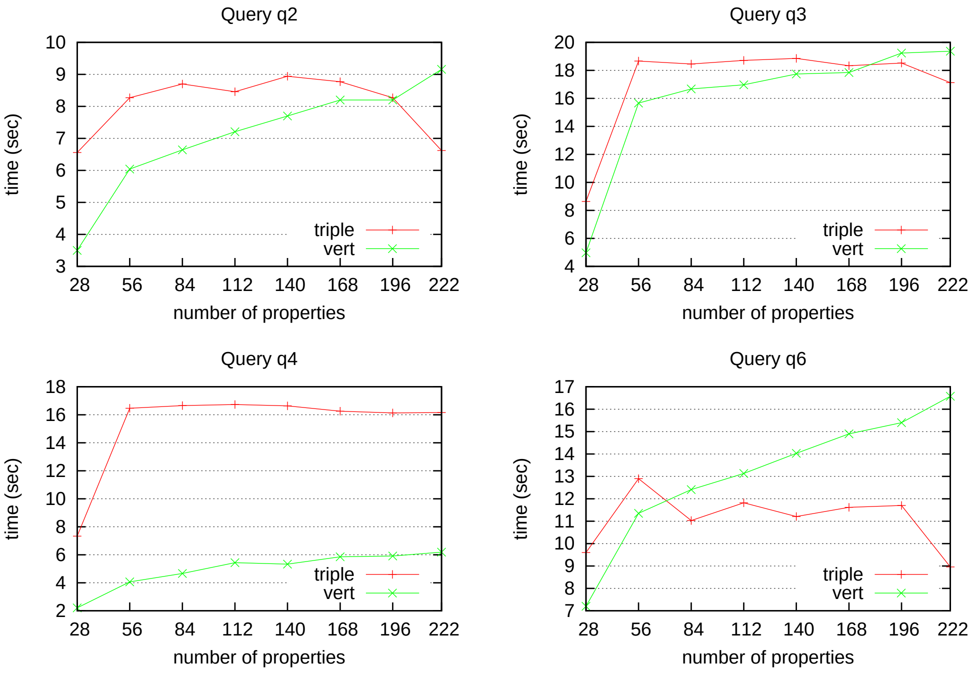


Figure 6: Execution time for queries q2, q3, q4, and q6 when increasing the number of properties

issue anymore, all asterisk (*) versions of the queries are faster on triple-store than on the vertically-partitioned implementation. However, *q4** is yet another “black swan”, since for cold runs it does not follow the same pattern. The increased time is caused by large intermediate results produced by query *q4* and *q4** for the triple-store implementation.

The geometric mean of query times for the vertically-partitioned approach and triple-store implemented on MonetDB/SQL, shows that the former achieves a better performance. Our conclusion is aligned with the results presented in [1], albeit the performance difference is small – within a factor 2 – when the proper clustering is chosen. However, the increased query execution time of the vertically-partitioned approach when all properties are considered is troublesome and calls for further experimentation. The last column titled $\frac{G*}{G}$ of the result tables, shows that when all queries are included in the calculation, the geometric mean for the vertically-partitioned approach, on both DBX and MonetDB/SQL, exhibits a larger increase (factor 1.9 to 2.4) than for triple-store (factor 1.0 to 1.6). This is an indication that the vertically-partitioned approach may exhibit scalability problems if the application does not restrict the number of the properties to be considered for a query, or if the data-set contains more property values.

Finally, the comparison of the execution time for each query over DBX, MonetDB/SQL and C-Store, verify that column-stores are better suited for RDF data management applications. Both MonetDB/SQL and C-Store achieve at least an order of magnitude improvement over the equivalent implementations on the row-store engine DBX. Comparing the column-stores with each other, C-Store is faster than MonetDB/SQL for queries *q1* and *q2*. MonetDB/SQL outperforms C-Store for the rest of the queries, namely *q3, q4, q5, q6,* and *q7*.

4.4 Scale up Experiment

In real-life RDF applications the number of properties may grow in the order of thousands, and their semantics may not permit an a priori restriction on the number of properties to consider. From the experiments conducted so far we notice that the execution time of queries implemented on a vertically-partitioned storage increases as the number of properties is increased from 28 to 222. This behavior indicates a potential scalability problem.

The scalability problem arises from the increased number of tables to be read and the number of unions and joins to be done in the vertically-partitioned approach when the number of properties is increased. The logical schema of vertically-partitioned RDF storage is data dependent, making it sensitive to data distribution. In our row-store experiments this effect becomes clear when we compare execution times of cold runs for queries *q2, q3, q4* and *q6* with their asterisk (*) counterparts, e.g., *q4** is 8 times slower than *q4*.

Figure 6 presents the different times for queries *q2, q3, q4,* and *q6* on MonetDB/SQL, for both triple-store clustered on *PSO* and the vertically-partitioned implementation, when the number of properties is gradually increased. For all queries, there is a clear tendency for the vertically-partitioned query execution times to increase when the number of properties is increased. On the other hand, although initially triple-store is slower than the vertically-partitioned approach, the time line drawn is non-increasing, which results in all cases but *q4* to eventually outperform vertically-partitioned queries. The steep increase observed with triple-store between the 28 and 56 properties is explained by the skewed distribution of the properties in the data-set. These 56 properties account for the 99% of the total rows in the triples table. When all 222 properties are included the execution time of all queries in triple-store drops because there is no final join required anymore to filter out properties.

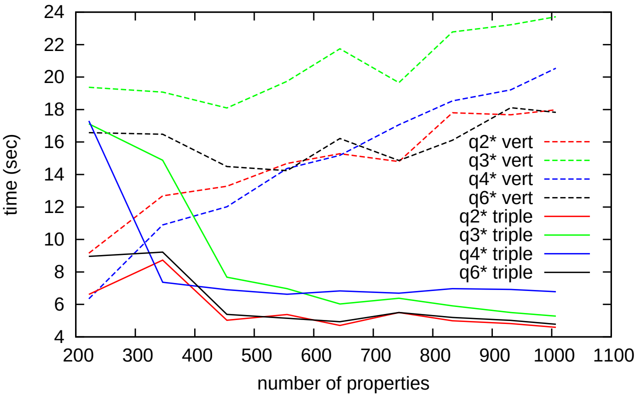


Figure 7: Scalability experiment

The highly Zipfian skew of property distribution and the small number of properties observed on the benchmark data-set[5] keeps this effect to a minimum level. Given an RDF data-set with more properties but with the same overall number of triples, we anticipate that these scalability issues will arise to the surface in a more obvious way.

To further investigate this claim, we conduct a scalability experiment using the same data-set, thus keeping the same number of triples, but increasing gradually the number of properties in the data-set. This is done by splitting in each round an arbitrary number of properties into n sub-properties, where $n = 1 \ldots 9$. The triples defined over the split properties are re-defined on one of the sub-properties following a uniform distribution.

Figure 7 depicts the times obtained for queries *q2*, q3*, q4**, and *q6** for both triple-store and the vertically-partitioned implementation on MonetDB/SQL. We start with the initial data-set (i.e., 222 properties) and re-run the queries each time a splitting step takes place, until the number of properties reaches to 1000.

When the number of properties is 222, vertically-partitioned query execution times outperform the triple-store times. However, when the number of properties is increased, triple-store exhibits better performance results than the vertically-partitioned approach. This is explained from the fact that the size of intermediate results for triple-store is reduced. Although the same reduction of intermediate results takes place for the vertically-partitioned approach, we observe that the query execution times are steadily increasing. The overhead of hundreds of unions and joins becomes more dominant and thus the vertically-partitioned approach scales poorly.

[5] see Section 2

5. SUMMARY AND CONCLUSIONS

In this experimentation paper we look at the scope of validity of the techniques and results presented in [1]. We study the benchmark from its design perspective as being representative for RDF storage. Our study illustrates that a better designed benchmark is called for to cover a significantly larger portion of the query space.

The base line redo experiment confirms the performance trends observed in the original paper. Re-implementation of both triple-store and vertically-partitioned approach in MonetDB/SQL and DBX draws a more balanced picture.

Concerning the comparison between the triple-store RDF storage solution and the vertically-partitioned approach, both implemented using a state-of-the-art commercial row-store engine, we conclude that once the proper clustered indices are used, the triple-store performs better than the vertically-partitioned approach. On our column-store implementation we show that the vertically-partitioned approach exhibits better query execution times, for the benchmark data-set at hand. However, we point out potential scalability problems for the vertically-partitioned approach when the number of properties in an RDF data-set is high. With a larger number of properties, the triple-store solution manages to outperform the vertically-partitioned approach also on our column-store engine. Combined with the fact that the logical schema of the vertically-partitioned approach is data-dependent, we are skeptical whether the proposed vertically-partitioned approach is a viable solution for scalable RDF data management applications. We believe, that these experimental results greatly expand the amount of reference data for the performance verdict in [1].

Overall, we may conclude that the processing efficiency of column-stores is particularly suited for RDF data management applications, while traditional systems combined with proper database management skills remain a serious contender.

Acknowledgments

We thank the authors of [1] for their help in setting up the C-store software. The members of the CWI Database group all have contributed one way or another to embark upon this new track of scientific papers and the effort it requires to execute it in a short time frame.

6. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 411–422. VLDB Endowment, September 2007.

[2] Barton Library Catalog Data. http://simile.mit.edu/rdf-test-data/barton/.

[3] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 54–68, 2002.

[4] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227, 2005.

[5] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems*, pages 1–15, 2003.

[6] MonetDB/SQL. http://monetdb.cwi.nl/SQL.

[7] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

[8] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column Oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564, 2005.

[9] K. Wilkinson. Jena Property Table Implementation. In *Proceedings of the Second International Workshop on Scalable Semantic Web Knowledge Base Systems*, pages 54–68, 2006.

[10] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proceedings of the First International Workshop on Semantic Web and Databases*, pages 131–150, 2003.

APPENDIX

Below, we list the SQL code of all benchmark queries as used with the triple-store implementation. The SQL code for the vertically-partitioned implementation is produced by a Perl script. The input of the Perl script is the SQL code of triple-store and a list of properties to be iterated over in the `FROM` clause. We use the newly introduced SQL/2003 constructor `WITH` to define the temporary tables needed for the vertically-partitioned implementation. The actual queries use integer predicates, since all strings are encoded on a dictionary structure.

```
-- Query 1
  SELECT A.obj, count(*)
    FROM triples AS A
   WHERE A.prop = ’<type>’
GROUP BY A.obj;

-- Query 2
  SELECT B.prop, count(*)
    FROM triples AS A, triples AS B,
         properties P
   WHERE A.subj = B.subj
     AND A.prop = ‘<type>’
     AND A.obj = ‘<Text>’
     AND P.prop = B.prop
GROUP BY B.prop;

-- Query 2*
  SELECT B.prop, count(*)
    FROM triples AS A, triples AS B
   WHERE A.subj = B.subj
     AND A.prop = ‘<type>’
     AND A.obj = ‘<Text>’
GROUP BY B.prop;

-- Query 3
  SELECT B.prop, B.obj, count(*)
    FROM triples AS A, triples AS B,
         properties P
   WHERE A.subj = B.subj
     AND A.prop = ‘<type>’
     AND A.obj = ‘<Text>’
     AND P.prop = B.prop
GROUP BY B.prop, B.obj
  HAVING count(*) > 1;

-- Query 3*
  SELECT B.prop, B.obj, count(*)
    FROM triples AS A, triples AS B
   WHERE A.subj = B.subj
     AND A.prop = ‘<type>’
     AND A.obj = ‘<Text>’
GROUP BY B.prop, B.obj
  HAVING count(*) > 1;

-- Query 4
  SELECT B.prop, B.obj, count(*)
    FROM triples AS A, triples AS B, triples AS C,
         properties P
   WHERE A.subj = B.subj
     AND A.prop = ‘<type>’
     AND A.obj = ‘<Text>’
     AND P.prop = B.prop
     AND C.subj = B.subj
     AND C.prop = ‘<language>’
     AND C.obj =  ‘<language/iso639-2b/fre>’
GROUP BY B.prop, B.obj
  HAVING count(*) > 1;

-- Query 4*
  SELECT B.prop, B.obj, count(*)
    FROM triples AS A, triples AS B, triples AS C
   WHERE A.subj = B.subj
     AND A.prop = ‘<type>’
     AND A.obj = ‘<Text>’
     AND C.subj = B.subj
     AND C.prop = ‘<language>’
     AND C.obj =  ‘<language/iso639-2b/fre>’
GROUP BY B.prop, B.obj
  HAVING count(*) > 1;

-- Query 5
 SELECT B.subj, C.obj
   FROM triples AS A, triples AS B, triples AS C
  WHERE A.subj = B.subj
    AND A.prop = ‘<origin>’
    AND A.obj = ‘<info:marcorg/DLC>’
    AND B.prop = ‘<records>’
    AND B.obj = C.subj
    AND C.prop = ’<type>’
    AND C.obj != ’<Text>’;

-- Query 6
 SELECT A.prop, count(*)
   FROM triples AS A,
        properties P,
       (
 (SELECT B.subj
    FROM triples AS B
           WHERE B.prop = ‘<type>’
             AND B.obj = ‘<Text>’)
 UNION
 (SELECT C.subj
    FROM triples AS C, triples AS D
           WHERE C.prop = ‘<records>’
     AND C.obj = D.subj
             AND D.prop = ‘<type>’
             AND D.obj = ‘<Text>’)
) AS uniontable
   WHERE A.subj = uniontable.subj
     AND P.prop = A.prop
GROUP BY A.prop;

-- Query 6*
 SELECT A.prop, count(*)
   FROM triples AS A,
       (
 (SELECT B.subj
    FROM triples AS B
           WHERE B.prop = ‘<type>’
             AND B.obj = ‘<Text>’)
 UNION
 (SELECT C.subj
    FROM triples AS C, triples AS D
           WHERE C.prop = ‘<records>’
     AND C.obj = D.subj
             AND D.prop = ‘<type>’
             AND D.obj = ‘<Text>’)
) AS uniontable
   WHERE A.subj = uniontable.subj
GROUP BY A.prop;

-- Query 7
 SELECT A.subj, B.obj, C.obj
   FROM triples AS A, triples AS B, triples AS C
  WHERE A.prop = ‘<Point>’
    AND A.obj = ‘"end"’
    AND A.subj = B.subj
    AND B.prop = ‘<Encoding>’
    AND A.subj = C.subj
    AND C.prop = ’<type>’;

-- Query 8
 SELECT B.subj
   FROM triples AS A, triples AS B
  WHERE A.subj = ‘conferences’
    AND B.subj != ‘conferences’
    AND A.obj  = B.obj;
```