An Experimental Evaluation of Relational RDF Storage and Querying Techniques

Hooran MahmoudiNasab[1] and Sherif Sakr[2]

[1] Macquarie University, Sydney, Australia
Hooran@ics.mq.edu.au
[2] University of New South Wales, Sydney, Australia
ssakr@cse.unsw.edu.au

Abstract. The Resource Description Framework (RDF) is a flexible model for representing information about resources in the web. With the increasing amount of RDF data which is becoming available, efficient and scalable management of RDF data has become a fundamental challenge to achieve the Semantic Web vision. The RDF model has attracted a lot of attention of the database community and many researchers have proposed different solutions to store and query RDF data efficiently. In this paper, we focus on evaluating the state-of-the-art of the approaches which are relying on the relational infrastructure to provide scalable engines to store and query RDF data. Our experimental evaluation is done on top of recently introduced SP^{2}Bench performance benchmark for RDF query engines. The results of our experiments shows that there is still room for optimization in the proposed generic relational RDF storage schemes and thus new techniques for storing and querying RDF data are still required to bring forward the Semantic Web vision.

1 Introduction

The Resource Description Framework (RDF) is a W3C recommendation that has rapidly gained popularity as a mean of expressing and exchanging semantic metadata, i.e., data that specifies semantic information about data. RDF was originally designed for the representation and processing of metadata about remote information sources and defines a model for describing relationships among resources in terms of uniquely identified attributes and values. The basic building block in RDF is a simple tuple model, (subject, predicate, object), to express different types of knowledge in the form of fact statements. The interpretation of each statement is that subject S has property P with value O, where S and P are resource URIs and O is either a URI or a literal value. Thus, any object from one triple can play the role of a subject in another triple which amounts to chaining two labeled edges in a graph-based structure. Thus, RDF allows a form of reification in which any RDF statement itself can be the subject or object of a triple. One of the clear advantage of the RDF data model is its schema-free structure in comparison to the entity-relationship model where the entities, their attributes and relationships to other entities are strictly defined. In RDF,

M. Yoshikawa et al. (Eds.): DASFAA 2010, LNCS 6193, pp. 215–226, 2010.

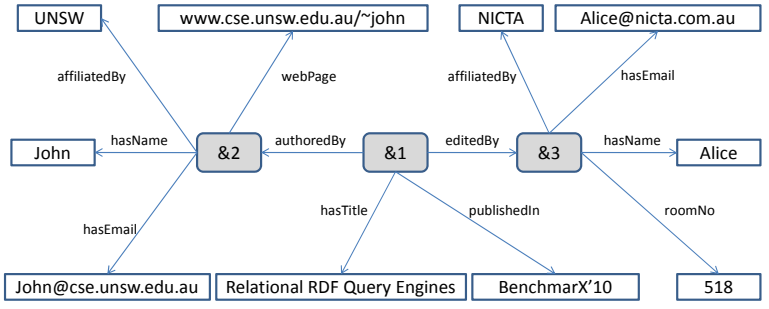


Fig. 1. Sample RDF Graph

```
SELECT  ?Z
WHERE  {?X  hasTitle "Relational RDF Query Engines".
        ?X  publishedIn "BenchmarX'10".
        ?X  authoredBy ?Y.
        ?Y   webPage ?Z.}
```

Fig. 2. Sample SPARQL query

the schema may evolve over the time which fits well with the modern notion of data management, dataspaces, and its *pay-as-you-go* philosophy [11]. Figure 1 illustrates a sample RDF graph.

The SPARQL query language is the official W3C standard for querying and extracting information from RDF graphs [14]. It represents the counterpart to *select-project-join* queries in the relational model. It is based on a powerful graph matching facility which allows binding variables to components in the input RDF graph and supports conjunctions and disjunctions of triple patterns. In addition, operators akin to relational joins, unions, left outer joins, selections, and projections can be combined to build more expressive queries. Figure 2 depicts a sample SPARQL query over the sample RDF graph of Figure 1 to *retrieve the web page information of the author of the paper published in BenchmarX'10 with the title "Relational RDF Query Engines"*.

Efficient and scalable management of RDF data is a fundamental challenge at the core of the Semantic Web. Relational database management systems (RDBMSs) have repeatedly shown that they are very efficient, scalable and successful in hosting types of data which have formerly not been anticipated to be stored inside relational databases such as complex objects [18], spatio-temporal data [3] and XML data [8]. In addition, RDBMSs have shown their ability to handle vast amounts of data very efficiently using powerful indexing mechanisms. Several research efforts have been proposed to provide efficient and scalable RDF querying engines by relying on the relational infrastructure [2,9,10,19]. These relational RDF query engines can be mainly classified to the following categories:

Subject	Predicate	Object
Id1	publishedIn	BenchmarX'10
Id1	hasTitle	Relational RDF Query Engines
Id1	authoredBy	Id2
Id2	hasName	John
Id2	affiliatedBy	UNSW
Id2	hasEmail	John@cse.unsw.edu.au
Id2	webPage	www.cse.unsw.edu.au/~john
Id1	editedBy	Id3
Id3	hasName	Alice
Id3	affiliatedBy	NICTA
Id3	hasEmail	Alice@nicta.com.au
Id3	roomNo	518

```
Select T3.Object
From Triples as T1, Triples as T2,
     Triples as T3, Triples as T4
Where T1.Predicate="publishedIn"
and T1.Object="Book Chapter"
and T2.predicate="hasTitle"
and T2.Object="Relational RDF Query Engines"
and T3.Predicate="webPage"
and T1.subject=T2.subject
and T4.subject=T1.subject
and T4.Predicate="authoredBy"
and T4.Object = T3.Subject
```

Fig. 3. Relational Representation of Triple RDF Stores

- **Vertical (triple) table stores:** where each RDF triple is stored directly in a three-column table (subject, predicate, object).
- **Property (n-ary) table stores:** where multiple RDF properties are modeled as n-ary table columns for the same subject.
- **Horizontal (binary) table stores:** where RDF triples are modeled as one horizontal table or into a set of vertically partitioned binary tables (one table for each RDF property).

Figures 3,4 and 5 illustrate examples of the three alternative relational representations of the sample RDF graph (Figure 1) and their associated SQL queries for evaluating the sample SPARQL query (Figure 2).

Experimental evaluation and comparison of different techniques and algorithms which deals with the same problem is a crucial aspect especially in applied domains of computer science. Previous studies of RDF query engines [15,16] have been presented in the literature. However, they were different in their focus. For example, [15] compares between the native RDF engines (with no database backend) while [16] compares the performances of triple stores and binary tables in the context of a column-oriented RDBMS, $MonetDB$. This paper takes a different focus by providing an extensive experimental study for evaluating the state-of-the-art of the *relational*-based RDF query engines in the context of traditional and most frequently used row-oriented RDBMS (e.g: PostgreSQL, Oracle, SQL Server, ...,etc). The remainder of this paper is organized as follows. Section 2 gives a brief overview over the state-of-the-art of the relational RDF query engines. Section 3 gives on overview of the SP^{2}Bench performance benchmark which we used to perform our experiments. Detailed description of the experimental framework and the experimental results are presented in Section 4. Section 5 concludes the chapter and suggests for possible future research directions on the subject.

Publication

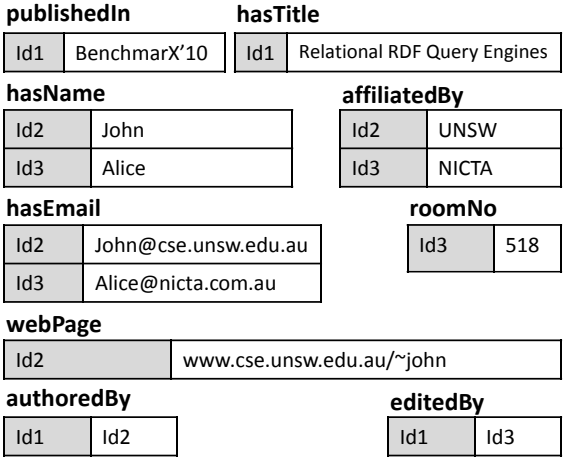
ID	publishedIn	hasTitle	authoredBy	editedBy
Id1	BenchmarX'10	Relational RDF Query Engines	Id2	id3

Person

ID	hasName	affiliatedBy	hasEmail	webPage	roomNo
Id2	John	UNSW	John@cse.unsw.edu.au	www.cse.unsw.edu.au/~john	
Id3	Alice	NICTA	Alice@nicta.com.au		518

```
Select  Person.webPage
From Person, Publication
Where Publication.publishedIn = "BenchmarX'10"
and  Publication.hasTitle = "Relational RDF Query Engines"
and  Publication.authoredBy = Person.ID
```

Fig. 4. Relational Representation of Property Tables RDF Stores




```
Select  webPage.value
From  PublishedIn, hasTitle,
        authoredBy, webPage
Where  publishedIn.value = "BenchmarX'10"
and hasTitle.value = "Relational RDF Query Engines"
and  publicationType.ID = hasTitle.ID
and  publicationType.ID  = authoredBy.ID
and   authoredBy.value = webPage.ID
```

Fig. 5. Relational Representation of Binary Tables RDF Stores

2 Relational RDF Query Engines: State-of-the-Art

2.1 Vertical (Triple) Stores

Harris and Gibbins [9] have described the 3store RDF storage system which is based on a central triple table that holds the hashes for the subject, predicate, object and the RDF graph identifier. A symbols table is used to allow reverse lookups from the hash to the hashed value and to allow SQL operations to be performed on pre-computed values in the data types of the columns without the use of casts. To produce the intermediate results table, the hashes of any SPARQL variables required to be returned in the results set are projected and the hashes from the intermediate results table are joined to the symbols table to provide the textual representation of the results.

Neumann and Weikum [13] have presented the RDF-3X (RDF Triple eXpress) query engine which tries to overcome the criticism that triples stores incurs too many expensive self-joins by creating the exhaustive set of indexes and relying on fast processing of merge joins . The physical design of RDF-3x is workload-independent and eliminates the need for physical-design tuning by building indexes over all 6 permutations of the three dimensions that constitute an RDF triple. Additionally, indexes over count-aggregated variants for all three two-dimensional and all three one-dimensional projections are created. The query processor follows RISC-style design philosophy [4] by using the full set of indexes on the triple tables to rely mostly on merge joins over sorted index lists. The query optimizer relies upon its cost model in finding the lowest-cost execution plan and mostly focuses on join order and the generation of execution plans.

Weiss et al. [19] have presented the Hexastore RDF storage scheme with main focuses on scalability and generality in its data storage, processing and representation. Hexastore does not discriminate against any RDF element and treats subjects, properties and objects equally. Each RDF element type have its special index structures built around it and every possible ordering of the importance or precedence of the three elements in an indexing scheme is materialized. Each index structure in a Hexastore centers around one RDF element and defines a prioritization between the other two elements. Two vectors are associated with each RDF element (e.g. subject), one for each of the other two RDF elements (e.g. property and object). In addition, lists of the third RDF element are appended to the elements in these vectors. In total, six distinct indices are used for indexing the RDF data. A clear disadvantage of this approach is that Hexastore features a worst-case five-fold storage increase in comparison to a conventional triples table.

2.2 Property Table Stores

Jena is a an open-source toolkit for Semantic Web programmers [20]. It uses a denormalized schema in which resource URIs and simple literal values are stored directly in the statement table. In order to distinguish database references from

literals and URIs, column values are encoded with a prefix that indicates the type of the value. A separate literals table is only used to store literal values whose length exceeds a threshold, such as blobs. Similarly, a separate resources table is used to store long URIs. By storing values directly in the statement table it is possible to perform many queries without a join. However, a denormalized schema uses more database space because the same value (literal or URI) is stored repeatedly. Jena permit multiple graphs to be stored in a single database instanceand supports the use of multiple statement tables in a single database so that applications can flexibly map graphs to different tables. In this way, graphs that are often accessed together may be stored together while graphs that are never accessed together may be stored separately.

Chong et al. [5] have introduced an Oracle-based property table approach which translates the RDF query to a self-join query on Triple-based RDF table store. The resulting query is executed efficiently by making use of B-tree indexes as well as creating materialized join views for specialized subject-property. Subject-Property Matrix materialized join views are used to minimize the query processing overheads that are inherent in the canonical triples-based representation of RDF. The materialized join views are incrementally maintained based on user demand and query workloads. A special module is provided to analyze the table of RDF triples and estimate the size of various materialized views, based on which a user can define a subset of materialized views. For a group of subjects, the system defines a set of single-valued properties that occur together.

Levandoski and Mokbel [12] have presented another property table approach for storing RDF data without any assumption about the query workload statistics. The approach provides a *tailored* schema for each RDF data set based on two main parameters: 1) *Support threshold* which represents a value to measure the strength of correlation between properties in the RDF data. 2) The *null threshold* which represents the percentage of null storage tolerated for each table in the schema. The approach involves two phases: *clustering* and *partitioning*. The clustering phase scans the RDF data to automatically discover groups of related properties. Based on the support threshold, each set of n properties which are grouped together in the same cluster are good candidates to constitute a single n-ary table and the properties which are not grouped in any cluster are good candidates for storage in binary tables. The partitioning phase goes over the formed clusters and balances the tradeoff between storing as many RDF properties in clusters as possible while keeping null storage to a minimum based on the null threshold.

2.3 Horizontal Stores

Abadi et al. [2] have presented *SW-Store* a new DBMS which is storing RDF data using a fully decomposed storage model (DSM) [6]. In this approach, the triples table is rewritten into n two-column tables where n is the number of unique properties in the data. In each of these tables, the first column contains the subjects that define that property and the second column contains the object values for those subjects while the subjects that do not define a particular

Table 1. SP^{2}Bench Benchmark Queries

Q1	Return the year of publication of "Journal 1 (1940)".
Q2	Extract all inproceedings with properties: *creator, booktitle, issued, partOf, seeAlso, title, pages, homepage,* and optionally *abstract*, including their values.
Q3abc	Select all articles with property **(a)** pages **(b)** month **(c)** isbn.
Q4	Select all distinct pairs of article author names for authors that have published in the same journal.
Q5	Return the names of all persons that occur as author of at least one inproceeding and at least one article.
Q6	Return, for each year, the set of all publications authored by persons that have not published in years before.
Q7	Return the titles of all papers that have been cited at least once, but not by any paper that has not been cited itself.
Q8	Compute authors that have published with Paul Erdos or with an author that has published with Paul Erdös.
Q9	Return incoming and outgoing properties of persons.
Q10	Return publications and venues in which "Paul Erdös" is involved either as author or as editor.
Q11	Return top 10 electronic edition URLs starting from the 51th publication, in lexicographical order.
Q12abc	**(a)** Return yes if a person is an author of at least one inproceeding and article. **(b)** Return yes if an author has published with "Paul Erdös" or with an author that has published with "Paul Erdös". **(c)** Return yes if person "John Q. Public" exists.

property are simply omitted from the table for that property. Each table is sorted by subject, so that particular subjects can be located quickly, and that fast merge joins can be used to reconstruct information about multiple properties for subsets of subjects. For a multi-valued attribute, each distinct value is listed in a successive row in the table for that property. The implementation of SW-Store relies on a column-oriented DBMS, C-store [17], to store tables as collections of columns rather than as collections of rows.

3 SP^{2}Bench Performance Benchmark

In [15] Schmidt et al. have presented the **SP**ARQL **P**erformance **Bench**mark (SP^{2}Bench) which is based on the DBLP scenario [1]. The DBLP database presents an extensive bibliographic information about the field of Computer Science and, particularly, databases. The benchmark is accompanied with a data generator which supports the creation of arbitrarily large DBLP-like models in RDF format. This data generator mirrors the vital key characteristics and distributions of the original DBLP dataset. The logical RDF schema for the DBLP dataset consists of *Authors* and *Editors* entities which are representation types of *Persons*. A superclass *Document* which is decomposed into several sub-classes: *Proceedings, Inproceedings, Journal, Article, Book, PhDThesis, MasterThesis, Incollection, WWW resources.* The RDF graph representation of these entities reflects their instantiation and the different types of relationship between them.

In addition, the benchmark provides 17 queries defined using the SPARQL query language on top of the structure of the DBLP dataset in a way to cover the most important SPARQL constructs and operator constellations. The defined queries vary in their complexity and result size. Table 1 lists the SP^{2}Bench

Benchmark Queries. For more details about the benchmark specification, data generation algorithm and SPARQL definition of the benchmark queries, we refer the reader to [15].

4 Experimental Evaluation

4.1 Settings

Our experimental evaluation of the alternative relational RDF storage techniques are conducted using the IBM DB2 DBMS running on a PC with 3.2 GHZ Intel Xeon processors, 4 GB of main memory storage and 250 GB of SCSI secondary storage. We used the SP^{2}Bench data generator to produce four different testing datasets with number of triples equal to: 500K, 1M, 2M and 4M Triples. In our evaluation, we consider the following four alternative relational storage schemes:

1. **Triple Stores (TS):** where a single relational table is used to store the whole set of RDF triples (subject, predicate, object). We follow the RDF-3X and build indexes over all 6 permutations of the three fields of each RDF triple.
2. **Binary Table Stores (BS):** for each unique predicate in the RDF data, we create a binary table (ID, Value) and two indexes over the permutations of the two fields are built.
3. **Traditional Relational Stores (RS):** In this scheme, we use the Entity Relationship Model of the DBLP dataset and follow the traditional way of designing normalized relational schema where we build a separate table for each entity (with its associated descriptive attributes) and use foreign keys to represent the relationships between the different objects. We build specific partitioned B-tree indexes [7] for each table based on the referenced attributes in the benchmark queries.
4. **Property Table Stores (PS):** where we use the schema of *RS* and decompose each entity with number of attributes ≥ 4 into two subject-property tables. The decomposition is done blindly and based on the order of the attributes without considering the benchmark queries (workload independent).

4.2 Performance Metrics

We measure and compare the performance of the alternative relational RDF storage techniques using the following metrics:

- **Loading Time:** represents the period of time for shredding the RDF dataset into the relational tables of the storage scheme.
- **Storage Cost:** depicts the size of the storage disk space which is consumed by the relational storage schemes for storing the RDF dataset.
- **Query Performance:** represents the execution times for the different SQL-translation of the SPARQL queries of SP^{2}Bench over the alternative relational storage schemes.

Table 2. A comparison between the alternative relational RDF storage techniques in terms of their loading times

	Loading Time (in Seconds)			
Dataset	**Triple Stores**	**Binary Tables**	**Traditional Relational**	**Property Tables**
500K	282	306	212	252
1M	577	586	402	521
2M	1242	1393	931	1176
4M	2881	2936	1845	2406

Table 3. A comparison between the alternative relational RDF storage techniques in terms of their storage cost

	Storage Cost (in KB)			
Dataset	**Triple Stores**	**Binary Tables**	**Traditional Relational**	**Property Tables**
500K	24721	32120	8175	10225
1M	48142	64214	17820	21200
2M	96251	128634	36125	43450
4M	192842	257412	73500	86200

All reported numbers of the query performance metric are the average of five executions with the highest and the lowest values removed. The rational behind this is that the first reading of each query is always expensively inconsistent with the other readings. This is because the relational database uses buffer pools as a caching mechanism. The initial period when the database spends its time loading pages into the buffer pools is known as the warm up period. During this period the response time of the database declines with respect to the normal response time.

For all metrics: the lower the metric value, the better the approach.

4.3 Experimental Results

Table 2 summarizes the loading times for shredding the different datasets into the alternative relational representations. The RS scheme is the fastest due to the less required number of insert tuple operations. Similarly, the TS requires less loading time than BS since the number of inserted tuples and updated tables are smaller for each triple.

Table 3 summarizes the storage cost for the alternative relational representations. The RS scheme represents the cheapest approach because of the normalized design and the absence of any data redundancy. Due to the limited percentage of the sparsity in the DBLP dataset, the PS does not introduce any additional cost in the storage space except a little overhead due to the redundancy of the object identification attributes in the decomposed property tables. The BS scheme represents the most expensive approach due to the redundancy of the ID attributes for each binary table. It should be also noted that the storage cost of TS and BS are affected by the additional sizes of their associated indexes.

Table 4. A comparison between the alternative relational RDF storage techniques in terms of their query performance (in milliseconds)

	1M				2M				4M			
	TS	**BS**	**RS**	**PS**	**TS**	**BS**	**RS**	**PS**	**TS**	**BS**	**RS**	**PS**
Q1	1031	1292	606	701	1982	2208	1008	1262	3651	3807	1988	2108
Q2	1672	1511	776	1109	2982	3012	1606	1987	5402	5601	2308	3783
Q3a	982	1106	61	116	1683	1873	102	198	3022	3342	191	354
Q3b	754	883	46	76	1343	1408	87	132	2063	2203	176	218
Q3c	1106	1224	97	118	1918	2109	209	275	3602	3874	448	684
Q4	21402	21292	11876	14116	38951	37642	20192	25019	66354	64119	39964	48116
Q5	1452	1292	798	932	2754	2598	1504	1786	5011	4806	3116	35612
Q6	2042	1998	1889	2109	3981	3966	3786	4407	7011	6986	6685	8209
Q7	592	30445	412	773	1102	58556	776	1546	2004	116432	1393	2665
Q8	9013	8651	1683	1918	15932	13006	3409	3902	27611	24412	8012	8609
Q9	2502	15311	654	887	4894	26113	1309	1461	9311	37511	2204	2671
Q10	383	596	284	387	714	1117	554	708	1306	2013	1109	1507
Q11	762	514	306	398	1209	961	614	765	2111	1704	1079	1461

Table 4 summarizes the query performance for the SP^{2}Bench benchmark queries over the alternative relational representations using the different sizes of the dataset. Remarks about the results of this experiment are given as follows:

- There is no clear winner between the triple store (TS) and the binary table (BS) encoding schemes. Triple store (TS) with its simple storage and the huge number of tuples in the encoding relation is still very competitive to the binary tables encoding scheme because of the full set of B-tree physical indexes over the permutations of the three encoding fields ($subject, predicate, object$).
- The query performance of the (BS) encoding scheme is affected badly by the increase of the number of the predicates in the input query. It is also affected by the *subject-object* or *object-object* type of joins where no index information is available for utilization. Such problem could be solved by building materialized views over the columns of the most frequently referenced pairs of attributes.
- Although their generality, there is still a clear gap between the query performance of the (TS) and (BS) encoding schemes in comparison with the tailored relational encoding scheme (RS) of the RDF data. However, designing a tailored relational schema requires a detailed information about the structure of the represented objects in the RDF dataset. Such information is not always available and designing a tailored relational schema limits the schema-free advantage of the RDF data because any new object with a variant schema will require applying a change in the schema of the underlying relational structure. Hence, we believe that there is still required efforts to improve the performance of these generic relational RDF storages and reduce the query performance gap with the tailored relational encoding schemes.

– The property tables encoding schemes (PS) are trying to fill the gap between the generic encoding schemes (TS and BS) and the tailored encoding schemes (RS). The results of our experiments show that the (PS) encoding scheme can achieve a comparable query performance to the (RS) encoding scheme. However, designing the schema of the property tables requires either explicit or implicit information about the characteristics of the objects in the RDF dataset. Such explicit information can not be always available and the process of inferring such implicit information introduces an additional cost of a pre-processing phase. Such challenges call for new techniques for flexible designs for the property tables encoding schemes.

5 Concluding Remarks

A naive relational way to store a set of RDF statements is using a relational database with a single table including columns for subject, property, and object. While simple, this schema quickly hits scalability limitations. Therefore, several approaches have been proposed to deal with this limitation by using extensive set of indexes or by using selectivity estimation information to optimize the join ordering [13,19]. Another approach to reduce the self-join problem is to create separate tables (property tables) for subjects that tend to have common properties defined [5,12]. In [2] Abadi et al. have explored the trade-off between triple-based stores and binary tables-based stores of RDF data. The main advantages of binary tables are:

– **Improved bandwidth utilization:** In a column store, only those attributes that are accessed by a query need to be read off disk. In a row-store, surrounding attributes also need to be read since an attribute is generally smaller than the smallest granularity in which data can be accessed.
– **Improved data compression:** Storing data from the same attribute domain together increases locality, improves data compression ratio and reduce the bandwidth requirements when transferring compressed data.

On the other side, binary tables do have the following main disadvantages:

– **Increased cost of inserts:** Column-stores perform poorly for insert queries since multiple distinct locations on disk have to be updated for each tuple.
– **Increased tuple reconstruction costs:** In order for column-stores to offer a standards-compliant relational database interface (e.g. ODBC, JDBC, etc.), they must at some point in a query plan stitch values from multiple columns together into a row-store style tuple to be output from the database.

In [2] Abadi et al. reported that the performance of binary tables is superior to clustered property table while [16] reported that even in column-store database, the performance of binary tables is not always better than clustered property table and depends on the characteristics of the data set. Moreover, the experiments of [2] reported that storing RDF data in column-store database is better than that of row-store database while [16] experiments have shown that the gain

of performance in column-store database depends on the number of predicates in a data set. Our experiments have shown that no approach is dominant for all queries and none of these approaches can compete with a tailored relational model. Therefore, we believe that there is still required efforts for improving the performance of the proposed generic relational RDF storage schemes and thus new techniques for storing and querying RDF data need to be developed to support the achievement of the Semantic Web design goals.

References

1. DBLP Computer Science Biliography, `http://www.informatik.uni-trier.de/~ley/db/`
2. Abadi, D., Marcus, A., Madden, S., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for Semantic Web data management. VLDB J. 18(2) (2009)
3. Botea, V., Mallett, D., Nascimento, M., Sander, J.: PIST: An Efficient and Practical Indexing Technique for Historical Spatio-Temporal Point Data. GeoInformatica 12(2) (2008)
4. Chaudhuri, S., Weikum, G.: Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In: VLDB (2000)
5. Inseok Chong, E., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL-based RDF Querying Scheme. In: VLDB (2005)
6. Copeland, G., Khoshafian, S.: A Decomposition Storage Model. In: SIGMOD (1985)
7. Graefe, G.: Sorting and Indexing with Partitioned B-Trees. In: CIDR (2003)
8. Grust, T., Sakr, S., Teubner, J.: XQuery on SQL Hosts. In: VLDB (2004)
9. Harris, S., Gibbins, N.: 3store: Efficient Bulk RDF Storage. In: PSSS (2003)
10. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: LA-WEB (2005)
11. Jeffery, S., Franklin, M., Halevy, A.: Pay-as-you-go user feedback for dataspace systems. In: SIGMOD (2008)
12. Levandoski, J., Mokbel, M.: RDF Data-Centric Storage. In: ICWS (2009)
13. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB 1(1) (2008)
14. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF, W3C Recommendation (January 2008), `http://www.w3.org/TR/rdf-sparql-query/`
15. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^2Bench: A SPARQL Performance Benchmark. In: ICDE (2009)
16. Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. PVLDB 1(2) (2008)
17. Stonebraker, M., Abadi, D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-Store: A Column-oriented DBMS. In: VLDB (2005)
18. Türker, C., Gertz, M.: Semantic integrity support in SQL: 1999 and commercial (object-)relational database management systems. VLDB J. 10(4) (2001)
19. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB 1(1) (2008)
20. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF Storage and Retrieval in Jena2. In: SWDB (2003)