Relational Processing of RDF Queries: A Survey

Sherif Sakr and Ghazi Al-Naymat
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
{ssakr,ghazi}@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 the attention of the database community and many researchers have proposed different solutions to store and query RDF data efficiently. This survey focuses on using relational query processors to store and query RDF data. We provide an overview of the different approaches and classify them according to their storage and query evaluation strategies.

1. INTRODUCTION

The goal of the Semantic Web is to provide a common framework for data-sharing across applications, enterprises, and communities. By giving data semantic meaning (through metadata), this framework allows machines to consume, understand, and reason about the structure and purpose of the data. The core of the Semantic Web is built on the Resource Description Framework (RDF) data model [17]. RDF describes a particular resource using a set of RDF statements of the form (subject, predicate, object) triples, also known as (subject, property, value). The *subject* is the resource, the *predicate* is the characteristic being described, and the *object* is the value for that characteristic.

Efficient and scalable management of RDF data is a fundamental challenge at the core of the Semantic Web. Several research efforts have been proposed to address these challenges [1, 2, 6, 13, 16, 28]. 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 [27], spatio-temporal data [5] and XML data [11].

This survey focuses on using relational query processors to store and query RDF data. We give an overview of the different approaches and classifies them according to their storage and indexing strategy. The rest of the paper is organized as follows. Section 2 introduces preliminaries of the RDF data model and the W3C standard RDF query language, SPARQL. It also introduce the main alternative relational approaches for storing and querying RDF. Sections 3,4 and 5 provide the details of the different techniques in each of the alternative relational approaches. Finally, Section 6 concludes the paper and provides some suggestions for possible future research directions on the subject.

2. RDF-SPARQL PRELIMINARIES

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, 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 [14]. Figure 1 illustrates a sample RDF graph.

The SPARQL query language is the official W3C standard for querying and extracting information from RDF graphs [22]. It represents the counterpart to *select-project-join* queries in the relational model. It is based on a powerful graph matching facility, 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.

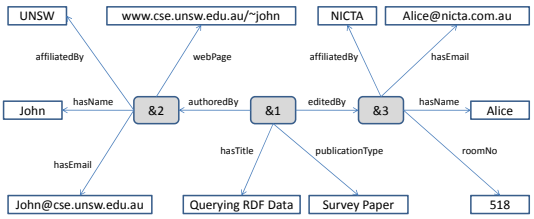

Figure 1: Sample RDF Graph

```
SELECT   ?Z
WHERE  { ?X  hasTitle "Querying RDF Data".
         ?X  publicationType "Survey Paper".
         ?X  authoredBy ?Y.
         ?Y   webPage ?Z. }
```

Figure 2: Sample SPARQL query

A basic SPARQL query has the form:

```
select ?variable1 ?variable2 ...
where { pattern1. pattern2. ... }
```

where each pattern consists of *subject*, *predicate* and *object*, and each of these can be either a variable or a literal. The query specifies the known literals and leaves the unknowns as variables which can occur in multiple patterns to constitute join operations. Hence, the query processor needs to find all possible variable bindings that satisfy the given patterns and return the bindings from the projection clause to the application. 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 a book chapter with the title "Querying RDF Data"*.

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. In addition, RDBMSs have shown their ability to handle vast amounts of data very efficiently using powerful indexing mechanisms. The relational RDF stores can be mainly classified to the following categories:

- **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).

Subject	Predicate	Object
Id1	publicationType	Survey Paper
Id1	hasTitle	Querying RDF Data
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="publicationType" and
T1.Object="Survey Paper"
and T2.predicate="hasTitle"
and T2.Object="Querying RDF Data"
and T3.Predicate="webPage"
and T1.subject=T2.subject
and T4.subject=T1.subject
and T4.Predicate="authoredBy"
and T4.Object = T3.Subject
```

Figure 3: Relational Representation of Triple RDF Stores

Publication

ID	publicationType	hasTitle	authoredBy	editedBy
Id1	Survey Paper	Querying RDF Data	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.publicationType = "Survey Paper"
and Publication.hasTitle = "Querying RDF Data"
and Publication.authoredBy = Person.ID
```

Figure 4: Relational Representation of Property Tables RDF Stores

3. VERTICAL (TRIPLE) STORES

Harris and Gibbins [12] have described the *3store* RDF storage system. The storage system of 3Store is based on a central triple table which holds the hashes for the subject, predicate, object and graph identifier. The graph identifier is equal to zero if the triple resides in the anonymous background graph. A symbols table is used to allow reverse lookups from the hash to the hashed value, for example, to return results. Furthermore it allows SQL operations to be performed on pre-computed values in the data types of the columns without the use of casts. For evaluating SPARQL queries, the triples table is joined once for each triple in the graph pattern where variables are bound to their values when they encounter the slot in which the variable appears. Subsequent occurrences of variables in the graph pattern are used to constrain any appropriate joins with their initial binding. 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 [20] have presented the *RDF-3X* (RDF Triple eXpress) RDF 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

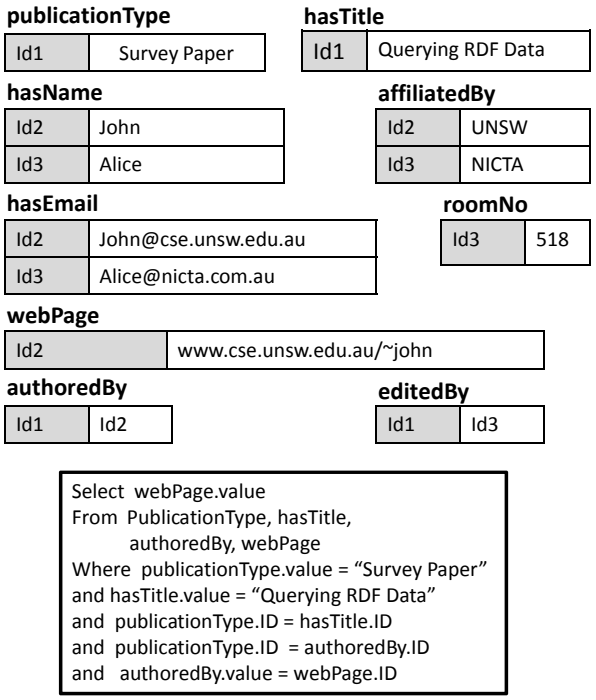

Figure 5: Relational Representation of Binary Tables RDF Stores

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 the RISC-style design philosophy [7] 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. In principle, selectivity estimation has a huge impact on plan generation. While this is a standard problem in database systems, the schema-free nature of RDF data makes the problem more challenging. RDF-3X employs dynamic programming for plan enumeration, with a cost model based on RDF-specific statistical synopses. It relies on two kinds of statistics: 1) specialized histograms which are generic and can handle any kind of triple patterns and joins. The disadvantage of histograms is that it assumes independence between predicates. 2) frequent join paths in the data which give more accurate estimation. During query optimization, the query optimizer uses the join-path selectivity information when available and otherwise assume independence and use the histograms information. In [21] the authors have extended the work further by introducing a run-time technique for accelerating query executions. It uses a light-weight, RDF-specific technique for sideways information passing across different joins and index scans within the query execution plans. They have also enhanced the selectivity estimator of the query optimizer by using very fast index lookups on specifically designed aggregation indexes, rather than relying on the usual kinds of coarse-grained histograms. This provides more accurate estimates at compile-time, at a fairly small cost that is easily amortized by providing better directives for the join-order optimization.

Weiss, et al. [28] have presented the *Hexastore* RDF storage scheme with main focuses on scalability and generality in its data storage, processing and representation. Hexastore is based on the idea of indexing the RDF data in a multiple indexing scheme [13]. It 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. Moreover, 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. These indices materialize all possible orders of precedence of the three RDF elements. A clear disadvantage of this approach is that Hexastore features a worst-case five-fold storage increase in comparison to a conventional triples table.

4. PROPERTY TABLE STORES

Due to the proliferations of self-joins involved with the triple-store, the property table approach was proposed. *Jena* is a an open-source toolkit for Semantic Web programmers [19]. It implements persistence for RDF graphs using an SQL database through a JDBC connection. The schema of the first version of Jena, Jena1, consisted of a statement table, a literals table and a resources table. The statement table *(Subject, Predicate, ObjectURI, ObjectLiteral)* contained all statements and referenced the resources and literals tables for subjects, predicates and objects. To distinguish literal objects from resource URIs, two columns were used. The literals table contained all literal values and the resources table contained all resource URIs in the graph. However, every query operation required multiple joins between the statement table and the literals table or the resources table.

To address this problem, the *Jena2* schema trades-off space for time. 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. The increase in database space consumption is addressed by using string compression schemes. Both Jena1 and Jena2 permit multiple graphs to be stored in a single database instance. In Jena1, all graphs were stored in a single statement. However, Jena2 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.

In principle, applications typically have access patterns in which certain subjects and/or properties are accessed together. For example, a graph of data about persons might have many occurrences of objects with properties name, address, phone, gender that are referenced together. Jena2 uses property table as a general facility for clustering properties that are commonly accessed together. A property table is a separate table that stores the subject-value pairs related by a particular property. A property table stores all instances of the property in the graph where that property does not appear in any other table used for the graph. In Jena1, each query is evaluated with a single SQL select query over the statement table. In Jena2, queries have to be generalized because there can be multiple statement tables for a graph. Using the knowledge of the frequent access patterns to construct the property-tables and influence the underlying database storage structures can provide a performance benefit and reduce the number of join operations during the query evaluation process.

Chong et al. [8] have introduced an Oracle-based SQL table function *RDFMATCH* to query RDF data. The results of RDFMATCH table function can be further processed by SQL's rich querying capabilities and seamlessly combined with queries on traditional relational data. The core implementation of RDFMATCH query translates 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 triple-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. These can be direct properties of these subjects or nested properties. A property p_1 is a direct property of subject x_1 if there is a triple (x_1, p_1, x_2). A property p_m is a nested property of subject x_1 if there is a set of triples such as, $(x_1, p_1, x_2), ..., (x_m, p_m, x_{m+1})$, where $m > 1$. For example, if there is a set of triples, $(John, address, addr1), (addr1, zip, 03062)$, then the *zip* property is considered as a nested property of *John*.

Levandoski and Mokbel [15] have presented another property table approach for storing RDF data without any assumption about the query workload statistics. The main goals of this approach are: (1) reducing the number of join operations which are required during the RDF query evaluation process by storing related RDF properties together (2) reducing the need to process extra data by tuning null storage to fall below a given threshold. The approach provides a *tailored* schema for each RDF data set which represents a balance between property tables and binary tables and is 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 (i.e., properties that always exist together for a large number of subjects). 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. One of the main concerns of the partitioning phase is twofold. The first is to ensure that there is no overlap between the clusters and that each property exists in a single cluster. The second is to reduce the number of table accesses and unions necessary in query processing.

Matono, et al. [18] have proposed a path-based relational RDF database. The main focus of this approach is to improve the performance for path queries by extracting all reachable path expressions for each resource and store them. Thus, there is no need to perform join operations unlike the flat tripe stores or the property tables approach. In this approach, the RDF graph is divided into subgraphs and then each subgraph is stored by applicable techniques into distinct relational tables. More precisely, all classes and properties are extracted from RDF schema data, and all resources are also extracted from RDF data. Each extracted item is assigned an identifier and a path expression and stored in corresponding relational table.

5. HORIZONTAL STORES

Abadi, et al. [1] have presented *SW-Store* as a new DBMS which stores RDF data using a fully decomposed storage model (DSM) [10]. 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 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. One advantage of this approach is that while property tables need to be carefully constructed so that they are wide enough but not too wide to independently answer queries, the algorithm for creating tables in the vertically partitioned approach is straightforward and need not change over time. Moreover, in the property-class schema approach, queries that do not restrict on class tend to have many union clauses while in the vertically partitioned approach, all data for a particular property is located in the same table and thus union clauses in queries are less common. The implementation of SW-Store relies on a column-oriented DBMS, C-store [26], to store tables as collections of columns rather than as collections of rows. In standard row-oriented databases (e.g., Oracle, DB2, SQLServer, Postgres, etc.) entire tuples are stored consecutively. The problem

with this is that if only a few attributes are accessed per query, entire rows need to be read into memory from disk before the projection can occur. By storing data in columns rather than rows, the projection occurs for free only where those columns that are relevant to a query need to be read.

[3, 9] have argued that storing a sparse data set (like RDF) in multiple tables can cause problems. They suggested storing a sparse data set in a single table while the complexities of sparse data management can be handled inside an RDBMS with the addition of an interpreted storage format. The proposed format starts with a header which contains fields such as relation-id, tuple-id, and a tuple length. When a tuple has a value for an attribute, the attribute identifier, a length field (if the type is of variable length), and the value appear in the tuple. The attribute identifier is the *id* of the attribute in the system catalog while the attributes that appear in the system catalog but not in the tuple are null for that tuple. Since the interpreted format stores nothing for null attributes, sparse data sets in a horizontal schema can in general be stored much more compactly in the format. While the interpreted format has storage benefits for sparse data, retrieving the values from attributes in tuples is more complex. In fact, the format is called interpreted because the storage system must discover the attributes and values of a tuple at tuple-access time, rather than using precompiled position information from a catalog, as the positional format allows. To tackle this problem, a new operator (called *EXTRACT* operator) is introduced to the query plans to precede any reference to attributes stored in the interpreted format and returns the offsets to the referenced interpreted attribute values which is then used to retrieve the values. Value extraction from an interpreted record is a potentially expensive operation that is dependent on the number attributes stored in a row or the length of the tuple. Moreover, if a query evaluation plan fetches each attribute individually and uses an EXTRACT call per attribute, the record will be scanned for each attribute and will thus be very slow. Thus, a batch EXTRACT technique is used to allow for a single scan of the present values in order to save time.

6. CONCLUDING REMARKS

RDF is a main foundation for processing the semantic of information stored on the Web. It is the data model behind the Semantic Web vision whose goal is to enable integration and sharing of data across different applications and organizations. The naive 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 estimations to optimize the join ordering [20, 28].

Another approach to reduce the self-join problem is to create separate tables (property tables) for subjects that tend to have common properties defined [8, 15]. Since Semantic Web data is often semi-structured, storing this data in a row-store can result in very sparse tables as more subjects or properties are added. Hence, this normalization technique is typically limited to resources that contain a similar set of properties and many small tables are usually created. The problem is that this may result in union and join clauses in queries since information about a particular subject may be located in many different property tables. This may complicate the plan generator and query optimizer and can degrade performance.

Abadi, et al. [1] 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 and thus data compression ratio. Hence, bandwidth requirements are further reduced 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 inserted tuple (one for each attribute).
- **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.

Abadi et al. [1] have reported that the performance of binary tables is superior to clustered property table while Sidirourgos et al. [25] 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 [1] reported that storing RDF data in column-store database is better than that of row-store database while [25] experiments have shown that the gain of performance in column-store database depends on the number of predicates in a data set. Other independent benchmarking projects [4, 23, 24] have shown that no approach is dominant for all queries and none of these approaches can compete with a purely relational model. Therefore, they are convinced 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.

7. REFERENCES

[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB Journal*, 18(2):385–406, 2009.

[2] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proceedings of the 2nd InternationalWorkshop on the Semantic Web*

(SemWeb), 2001.

[3] Jennifer L. Beckmann, Alan Halverson, Rajasekar Krishnamurthy, and Jeffrey F. Naughton. Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, page 58, 2006.

[4] Christian Bizer and Andreas Schultz. Benchmarking the Performance of Storage Systems that expose SPARQL Endpoints. In *Proceedings of the 4th International Workshop on Scalable Semantic Web knowledge Base Systems (SSWS)*., 2008.

[5] Viorica Botea, Daniel Mallett, Mario A. Nascimento, and Jörg Sander. PIST: An Efficient and Practical Indexing Technique for Historical Spatio-Temporal Point Data. *GeoInformatica*, 12(2):143–168, 2008.

[6] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the First International Semantic Web Conference (ISWC)*, pages 54–68, 2002.

[7] Surajit Chaudhuri and Gerhard Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB)*, pages 1–10, 2000.

[8] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 1216–1227, 2005.

[9] Eric Chu, Jennifer L. Beckmann, and Jeffrey F. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 821–832, 2007.

[10] George P. Copeland and Setrag Khoshafian. A Decomposition Storage Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–279, 1985.

[11] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 252–263, 2004.

[12] Stephen Harris and Nicholas Gibbins. 3store: Efficient Bulk RDF Storage. In *Proceedings of the First International Workshop on Practical and Scalable Semantic Systems (PSSS)*, 2003.

[13] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In *Proceedings of the Third Latin American Web Congress (LA-WEB)*, pages 71–80, 2005.

[14] Shawn R. Jeffery, Michael J. Franklin, and Alon Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 847–860, 2008.

[15] Justin J. Levandoski and Mohamed F. Mokbel. RDF Data-Centric Storage. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, 2009.

[16] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. RStar: an RDF storage and query system for enterprise resource management. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 484–491, 2004.

[17] Frank Manola and Eric Miller. RDF Primer, W3C Recommendation, February 2004. http://www.w3.org/TR/REC-rdf-syntax/.

[18] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A Path-based Relational RDF Database. In *Proceedings of the 16th Australasian Database Conference (ADC)*, pages 95–103, 2005.

[19] Brian McBride. Jena: A Semantic Web Toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[20] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endownment (PVLDB)*, 1(1):647–659, 2008.

[21] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 627–640, 2009.

[22] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF, W3C Recommendation, January 2008. http://www.w3.org/TR/rdf-sparql-query/.

[23] Michael Schmidt, Thomas Hornung, Norbert Küchlin, Georg Lausen, and Christoph Pinkel. An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In *Proceedings of the 7th International Semantic Web Conference (ISWC)*, pages 82–97, 2008.

[24] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench: A SPARQL Performance Benchmark. In *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, pages 222–233, 2009.

[25] Lefteris Sidirourgos, Romulo Goncalves, Martin L. Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *Proceedings of the VLDB Endownment (PVLDB)*, 1(2):1553–1563, 2008.

[26] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 553–564, 2005.

[27] Can Türker and Michael Gertz. Semantic integrity support in SQL: 1999 and commercial (object-)relational database management systems. *VLDB Journal*, 10(4):241–269, 2001.

[28] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endownment (PVLDB)*, 1(1):1008–1019, 2008.