Survey of RDF Storage Managers

Kiyoshi Nitta
Yahoo JAPAN Research
Tokyo, Japan
knitta@yahoo-corp.jp

Iztok Savnik
University of Primorska &
Institute Jozef Stefan, Slovenia
iztok.savnik@upr.si

***Abstract*—This paper surveys RDF storage manager implementations that belong to a kind of database system treating locally stored RDF triples. They are systematically classified in accordance with the properties of single and multiple process technologies that include the following types: triple table, index structure, query, string translation method, join optimization method, cache, data distribution method, query process distribution method, stream process, and resource sharing architecture. This classification is applied to *3store, 4store, Virtuoso, RDF-3X, Hexastore, Apache Jena, SW-Store, BitMat, AllegroGraph, and Hadoop/HBase*. While the classification structure is presented for each of them, detecting differences between them has become easy. The classification revealed that there will be room for further improvement of the efficient query process by developing multi-process technologies.**

***Keywords*— databases; RDF databases; distributed database systems; query processing system; database system implementation.**

I. INTRODUCTION

Resource description framework (RDF) data are widely used in the Internet and their volume is growing steadily. The linked open data (LOD) project promotes the acceleration of the accumulation of RDF data to provide freely accessible on-line resources [1]. The LOD project leverages RDF's advantages by providing a data publishing principle. Each data element can be distributed to any site of the Internet. Distributed data are connected by RDF links that are also RDF data and can be located on arbitrary sites. This strategy lowers the barrier for publishers to distribute their data freely and contributes to the accumulation of a huge amount of RDF data. There are two major approaches to access those RDF data [2], [3]:

(A-1) *Local Cache*
(A-2) *Federated Search*

Systems based on (A-1) gather a subset of RDF data on local computational resources to accelerate the processing of queries on frequently referenced data. After accepting user queries, systems based on (A-2) distribute sub-queries to several search services distributed over the Internet and integrate replies from them to obtain updated data that is as fresh as possible. While most practical access methods may be constructed by combining these two approaches, technologies used in (A-1) will play an important role for query process efficiency. This paper surveys the challenges and solutions for developing RDF storage managers based on (A-1).

There are three notable survey papers related to RDF storage managers. The RW'11 tutorial [2] gives the most comprehensive survey about scalable RDF processing technologies including centralized RDF repositories, distributed query processing, and scalable reasoning. This tutorial precisely explains distributed query processing system architectures that include semantic web search engines, federated systems, and P2P systems. The SIGMOD'12 tutorial [3] classified approaches for query processing over linked data into a centralized storage approach and distributed storage approach [2]. The centralized storage approach contains triple-stores based on relational database management systems, matrix, XML, and graph. Sakr and Al-Naymat [4] classified triple-stores based on relational database management systems into three categories: a) vertical (triple) table stores, b) property (n-ary) table stores, and c) horizontal (binary) table stores. This classification scheme is also explained in the above tutorials [2], [3]. The core classification structure introduced in these papers is almost the same.

Each survey paper provides a classification structure that classifies research efforts so far by focusing on the distinguishing aspects of researches. These survey papers provide useful insights and perspectives about component technologies of RDF storage managers. However, most researches implement prototype or practical systems that are equipped with combinations of useful technologies. It will be useful for researchers interested in RDF storage manager implementations to provide another type of classifications that gives several attributes to each research system. The contributions of this paper can be summarized as follows:

- Provides systematic classification of RDF storage manager implementations.
- Easily detects differences between given RDF storage manager implementations.
- Pick up attributes concerning effective processes by multi-process environments.

There are Internet pages that classify RDF storage managers. A Wikipedia page [5] provides the most comprehensive list of RDF storage managers (triplestores) with license and API function information. The W3C page [6] provides benchmark results of RDF storage managers for storing large-scale RDF data sets. While this paper provides internal functional information, those Internet pages may provide a useful perspective of RDF storage managers by combining information.

The rest of this paper is organized as follows. The Classification of RDF Storage Managers Based on Local Cache Approach section introduces a classification framework of RDF storage managers. The RDF storage managers section reports each characteristic of existing RDF storage managers using the classification framework. The Challenges section

 ISBN: 978-1-61208-334-6

discusses a few challenges inspired by comparing RDF storage managers. This paper is concluded in the Conclusion section.

II. Classification of RDF Storage Managers Based on Local Cache Approach

RDF storage managers in the local cache approach can be classified in accordance with several aspects. RDF storage managers are represented by $RSM(\mathcal{S}, \mathcal{M})$, where $\mathcal{S}$ and $\mathcal{M}$ show attributes of single and multiple process issues, respectively. Attribute set $\mathcal{S}$ has structure $\mathcal{S}(T_s, I_s, Q_s, S_s, J_s, C_s, D_s, F_s)$. Attribute T_s is the triple table type. It has one of the values of *vertical (v)*, *property (p)*, and *horizontal (h)*. These classes of triple tables are introduced and defined by Sakr et al. [4]. Attribute I_s is the index structure type of triple table. It has one of the values of *6-independent (6)*, *GSPO-OGPS (G)*, and *matrix (m)*. Attribute Q_s is the query type. It has one of the values of *SPARQL (S)* and *original (o)*. Attribute S_s is the translation method type of URI and literal strings. It has one or combonation of the values of *URI (U)*, *literal (l)*, *long (o)* and *none (n)*. Values *URI* and *literal* mean ID translations of URI and literal strings, respectively. Value *long* means that only long URIs and literals are translated to identifiers. Attribute J_s is the join optimization method type. It has one of the values of *RDBMS-based (R)*, *column-store-based (c)*, *conventional-ordering (o)*, *pruning (p)*, and *none (n)*. Attribute C_s is the cache type. It has one of the values of *materialized-path-index (m)*, *reified-statement (r)*, and *none (n)*. Attribute D_s is the dabase engine type. It has one of the values of *RDB (R)* and *custom (c)*. Attribute F_s is the inference feature type. It has one or combination of the values of *TBox (T)*, *ABox (A)*, and *no (n)*. While value *TBox* means inference features on the ontology level, value *ABox* means ones on the assertion level.

Attribute set $\mathcal{M}$ has structure $\mathcal{M}(D_m, Q_m, S_m, A_m)$. Attribute D_m is the data distribution method type. It has one of the values of *hash (h)*, *data-source (d)*, and *none (n)*. Attribute Q_m is the query process distribution method type. It has one of the values of *data-parallel (p)*, *data-replication (r)*, and *none (n)*. Attribute S_m is the stream process type. It has one of the values of *pipeline (p)* and *none (n)*. Attribute A_m is the resource sharing architecture type. It has one of the values of *memory (m)*, *disk (d)*, and *nothing (n)*.

Because some implementations do not disclose internal mechanisms, all attributes can have value *unknown ( )*. TABLE I shows the summary of the classification. The details are described in the next section. The values in the table correspond to the characters in parenthesis of the above description of possible values.

III. RDF storage managers

This section provides a detailed description of each RDF storage manager system. As there are many such systems, we omitted some systems due to space limitations.

A. 3store

The attributes of this implementation are as follows: $T_s = vertical$, $Q_s = SPARQL$, $S_s = string\_id$, $J_s = RDBMS\_based$, $D_s = RDB$, $F_s = TBox$, $D_m = none$, $Q_m = none$, $S_m = none$. 3store [7] was originally used for semantic web applications in particular for storing the hyphen.info RDF data set, which describes computer science research in the UK. The final version of the database consisted of 5,000 classes and about 20 million triples. 3store was implemented on top of the MySQL database management system. It included simple inferential capabilities e.g., class, sub-class, and sub-property queries, that are mainly implemented by means of MySQL queries. Hashing is used to translate URIs into an internal form of representation.

The 3store query engine used RDQL query language originally defined in the framework of the Jena project. RDQL triple expressions are first translated into relational calculus. Constraints are added to relational calculus expressions and translated into SQL. The inference is implemented by a combination of forward and backward chaining that computes the consequences of the asserted data.

B. 4store

The attributes of this implementation are as follows: $T_s = vertical$, $Q_s = SPARQL$, $S_s = string\_id$, $J_s = conventional\_ordering$, $D_s = RDB$, $D_m = hash$, $Q_m = data\_parallel$, $A_m = nothing$. 4store [8] was designed and implemented to support a range of novel applications that have emerged from the semantic web. RDF databases were constructed from web pages including people-centric information resulting from ontology with billions of RDF triples. The requirements were to store and manage $15x10^9$ triples. The design of 4store is based on 3store especially in the way RDF triples are represented.

4store is designed to operate on clusters of low-cost servers, and is implemented in ANSI C. It was estimated that the complete index for accessing quads would require around 100 GB of RAM, which was the reason for distributing data to a cluster of 64-bit multi-core x86 Linux servers with each cluster storing an RDF data partition. The cluster architecture is "shared nothing" architecture. Cluster nodes are divided into processing and storage nodes. Data segments stored on different nodes are determined by a simple formula. The formula uses resource identifiers (RID) that are indexes of URIs, literals and blank nodes. When triples are distributed to segments, RID of the triple subject is divided by number of segments. The remaining part of this calculation determines segment number of triple. One of the benefits of such a design is parallel access to RDF triples distributed to nodes holding segments of RDF data. Furthermore, segments can be replicated to distribute the total workload to the nodes holding replicated RDF data. The communication between nodes is directed by processing the nodes via TCP/IP. There is no communication between data nodes.

URIs are represented using resource identifiers that are similarly to those of 3store that were obtained by means of hashing. Triples are represented as quads. Each quad in a particular segment is stored in three indexes. Two of them are implemented using radix tries because of O(k) time complexity. The third index is used to access graphs by using a hash table.

The 4store query engine is based on relational algebra. A Rasqal SPARQL parser is used for parsing SPARQL queries. Queries are processed by SPARQL blocks. First, the

TABLE I. PROPERTIES OF RDF STORAGE MANAGERS

	$\mathcal{S}$								$\mathcal{M}$			
	T_s	I_s	Q_s	S_s	J_s	C_s	D_s	F_s	D_m	Q_m	S_m	A_m
3store	v		S	U	R		R	T	n	n	n	
4store	v		S	U	o		R		h	p		n
Virtuoso	v	G	S	Ulo	R		R	TA	n	n	n	
RDF-3X	v	6	S	Ul	o		R		n	n	n	
Hexastore	v	6	o	Ul		n			n	n	n	
Apache Jena	p		S	Ulo	R	r	R		n	n	n	
SW-Store	h			Uo	c	m	c		n	n	n	
BitMat	v	m	S	Ul	p		c				p	
AllegroGraph			S				c		h	p		m
Hadoop/HBase	h						c			p		m

UNION expressions are collapsed. Next, FILTER expressions are evaluated. Joins are then performed on the remaining blocks. Finally, any remaining FILTERs, ORDER BYs, and DISTINCTs are applied. The primary source of optimization is the conventional join ordering. However, they also use common subject optimization and cardinality reduction. In spite of considerable work on query optimization, 4store lacks complete query optimization as it is provided by relational query optimizers.

C. Virtuoso

The attributes of this implementation are as follows: $T_s = vertical$, $I_s = GSPO\_OGPS$, $Q_s = SPARQL$, $S_s = string\_id$, $J_s = RDBMS\_based$, $D_s = RDB$, $F_s = TBox, ABox$, $D_m = none$, $Q_m = none$, $S_m = none$. Virtuoso [9]–[11] is a multi-model database management system based on relational database technology. Besides the functionality of the relational database management system, it also provides RDF data management, XML data management, content management, and a Web application server.

The approach of Virtuoso is to treat triple-store as a table composed of four columns. The main idea of the approach to RDF data management is to exploit existing relational techniques and add functionality to the RDBMS to deal with features specific to RDF data. The most important aspects that were considered by Virtuoso designers are: extending SQL types with the RDF data type, dealing with unpredictable sizes of objects, providing efficient indexing and extending relational statistics to cope with the RDF store based on a single table as well as efficient storage of RDF data.

The initial solution for storing RDF triples is the use of a quad table storing attributes: subject (S), predicate (P), object (O), and graph (G). Columns S, P, and G are represented as IRI ID. Column O is represented by ID only if it is longer than 12 characters. The mapping between IRIs and local IDs is stored in a table, and the mapping between the long values of O and IDs is stored in a separate table.The quad table is represented using two covering indexes based on the GSPO and OGPS attributes. Since S is the last part of OGPS we can represent it using bitmaps. We have one bitmap for one distinctive value of OGP. Compression is used on page level, which still allow random page access.

Virtuoso includes SPARQL into SQL. SPARQL queries are translated into SQL during parsing. In this way, SPARQL has all aggregation functions. SPARQL UNION is translated directly into SQL, and SPARQL OPTIONAL is translated into left outer join. Since RDF triples are stored in one quad table, relational statistics is not useful. Virtuoso uses sampling during query translations to estimate the cost of alternative plans. Basic RDF inference on TBox is done using query rewriting. For ABox reasoning Virtuoso expands semantics of owl:same-as by transitive closure.

D. RDF-3X

The attributes of this implementation are as follows: $T_s = vertical$, $I_s = 6\_independent$, $Q_s = SPARQL$, $S_s = string\_id$, $J_s = conventional\_ordering$, $D_s = RDB$, $D_m = none$, $Q_m = none$, $S_m = none$. The triple-store RDF-3X reported by Neumann and Weikum [12], [13] built six independent indexes of SPO, SOP, OSP, OPS, PSO and POS (S, P, and O represent the subject, predicate, and object of the RDF triple element, respectively.) from one large triple table. The indexes are compressed using a byte-wise method that was carefully chosen to improve query process performance. They also constructed aggregated indexes for SP, PS, SO, OS, PO, and OP. They focused on join ordering to optimize the query process. The optimization uses selectivity statistics calculated for given queries using selectivity histograms and statistics of frequently accessed paths. Although it is equipped with a table to treat long URI strings as simple IDs, it has been pointed out that its translation performance was very bad [14].

They compared the RDF-3X system with PostgreSQL and MonetDB. They tried Jena2, Yars2, and Sesame 2.0, but those systems could not finish storing benchmark data in 24 hours in their experimental environment. The benchmark data contained the Barton data set (5.1×10^7 triples, 1.9×10^7 IDs, and 285 types of properties), YAGO data set (4.0×10^7 triples, 3.3×10^7 IDs, and 93 types of properties), and LibraryThing data set (3.6×10^7 triples, 9.3×10^6 IDs, and 3.3×10^5 types of properties). RDF-3X exceeded other systems by large margins. The source code is available for non-commercial purposes.

 ISBN: 978-1-61208-334-6

E. Hexastore

The attributes of this implementation are as follows: $T_s = vertical$, $I_s = 6\_independent$, $Q_s = original(customwrapped)$, $S_s = strings\_id$, $J_s = unknown(itseemsnone)$, $C_s = none$, $D_m = none$, $Q_m = none$, $S_m = none$. The Hexastore [15] approach to the RDF storage system uses triples as the basis for storing RDF data. The problems of existent triple-stores investigated are the scalability of RDF databases in a distributed environment, complete implementation of the query processor including query optimization, persistent indexes, and other topics provided by database technology.

Six indexes are defined at the top of the table with three columns, one for each combination of three columns. The index used for the implementation has three levels ordered by the particular combination of SPO attributes. Each level is sorted, giving in this way the means to use ordering for optimization during query evaluation. The proposed index provides natural representation of multi-valued properties and allows fast implementation of merge-join, intersection, and union.

F. Apache Jena

The attributes of this implementation are as follows: $T_s = property$, $Q_s = SPARQL$, $S_s = string\_id$, $J_s = RDBMS\_based$, $C_s = reified\_statement$, $D_s = RDB$, $D_m = none$, $Q_m = none$, $S_m = none$. In terms of the body of knowledge grown from the database community, Jena is a database programming language environment based on RDF for Java [16], [17], [18], [19]. It provides a simple abstraction and interface for manipulation of RDF graphs represented in main memory and backed by the database engine. The persistence of RDF graphs is achieved using a SQL database through a JDBC connection. Jena supports a number of database systems such as MySQL, Postgres, Oracle, and BerkeleyDB. At the core interface for manipulation of RDF graphs, Jena includes a range of RDF parsers, query language, and I/O modules for N3, N-Triple, and XML/RDF.

RDF statements are in a database-back-end of Jena represented using three tables. The URIs of resources are converted to indexes represented in one table. Larger literals are represented in another table while small literals are stored directly in a statement table. Finally, triples are stored in statement tables using indexes for resources and larger objects. Jena uses additional optimizations for fast access to common statements of a graph and reified statements. Furthermore, graphs can be stored in different sets of tables to improve the speed of query processing.

In Jena, persistence is achieved through persistent logical graphs composed of specialized graphs optimized for storing particular types of graphs. The database driver realizes access to databases abstracted away from particular database systems, each of which has their particular driver.

On a database level Jena includes three types of operations: operation ‘add’ that inserts new triples in a database, operation ‘delete’ that removes RDF statements from a database, and operation ‘find’ that retrieves RDF statements from a database. Query language RDQL converts SPARQL statements into a set of find patterns including variables that can be executed as joins. While Jena1 includes a query processing engine that does not include query optimization in a database system sense, Jena2 passes on queries to the database engine. Query optimization thus takes place in a database engine. Finally, Jena2 includes mechanisms for efficient retrieval of reified statements.

G. SW-Store

The attributes of this implementation are as follows: $T_s = horizontal$, $S_s = string\_id$, $J_s = column\_store\_based$, $C_s = materialized\_path\_index$, $D_s = custom$, $D_m = none$, $Q_m = none$, $S_m = none$. Abadi proposes the use of vertical partitioning for the representation of RDF databases [20]. The advantages of using column-stores for storing RDF are: efficient representation of NULL values; efficient implementation of multi-valued attributes; support for heterogeneous records; efficient merge-joins of sorted columns; and reduced number of unions in queries. To achieve fast access to selected access paths, they are materialized.

Database management system SW-Store [21] is based on vertical partitioning of RDF data. It has been shown that storage system based on columns can significantly improve some types of queries on RDF databases. Column-oriented storage system in SW-Store is improved by using column-oriented compression; optimization for fixed length tuples; optimization of merge-join code; using column oriented query optimization; and by materialized path expressions. Empirical results support the proposed use of column-oriented store in comparison to triple-store representation and property table representation of RDF database.

While above presented novel features of column-oriented data stores are important, it seems that the most important contribution of SW-Store is to show the possibility to use simple tools like sorting and map-based indexes on large-scale distributed clusters of servers. The simplicity of tools, on one hand, and the possibility of using database technologies like query optimization on column-stores, on the other, can result in very efficient query execution.

H. BitMat

The attributes of this implementation are as follows: $T_s = vertical$, $I_s = matrix$, $Q_s = SPARQL$, $S_s = string\_id$, $J_s = pruning$, $D_s = custom$, $S_m = pipeline$. The triple-store reported by Atre et al. [14] used a three dimensional compressed matrix index named BitMat. It avoids maintaining materialized triples as much as possible. Its query processing algorithm of SPARQL joins consists of two phases. The first phase performs efficient pruning of the triples. The second phase performs variable binding matching across the triple patterns to obtain the final results. Both two steps use compressed BitMats without any join table construction.

The BitMat index triple-store performed better than RDF-3X [12] for some queries that require managing large number of triples during join processes. They classified triple-pattern join queries to three types: a) highly selective triple patterns, b) triple patterns with low-selectivity but which generate few results, and c) low-selectivity triple patterns and low-selectivity join results. The BitMat system performed well processing type b) queries.

 ISBN: 978-1-61208-334-6

I. AllegroGraph

The attributes of this implementation are as follows: $Q_s = SPARQL$, $D_s = custom$, $D_m = hash$, $Q_m = data\_parallel$, $A_m = memory$. AllegroGraph [22] is a triple store built on an object store AllegroCache. They are proprietary products of Franz Inc. The precise architecture of AllegroGraph has not been fully disclosed. The strongest point is its capacity limit and processing speed. A clustered version of AllegroGraph stored 1 trillion triples in about 14 days [23]. In the scalability ranking of triple stores edited by W3C [6], AllegroGraph is ranked first as of August 2011. Its benchmark experiments were performed using PC servers that have huge memories (from 48GB to 1TB). AllegroGraph does not necessarily perform materialization to process queries. We could not find enough information about how the clustered version distributes data and optimizes query processes.

J. Hadoop/HBase

The attributes of this implementation are as follows: $T_s = horizontal$, $D_s = custom$, $Q_m = data\_parallel$, $A_m = memory$. Hadoop [24] offers software environment for the implementation of large-scale distributed systems processing data on clusters of servers. It was initially designed to support fast distributed processing of very large HTML graphs. Hadoop allows for the implementation of data centers comprised of up to many 1000 servers.

The basis of Hadoop includes a set of interfaces to distributed file system, general I/O operations, RPC, serialization, and persistent data structures. A distributed file system HDFS runs on large clusters of commodity machines. MapReduce model allows for efficient manipulation of sequential data files (SequenceFile) in distributed cluster of servers. *Dictionary* data structure for indexing records based on keys (MapFile) is used to support efficient implementation of operation *map*.

Sorted sequence files and map indexes provide programming environment for the implementation database operations such as selection, projection and joins. Hadoop includes dataflow programming language Pig, which can realize some forms of classical database operations. However, sorted files and map-based index (dictionary) provide limited basis for the implementation of database structures and operations. For instance, merge joins can be implemented efficiently while index-based joins and range queries can not be implemented without extending the functionality of Hadoop.

HBase [25] is column-oriented database system implemented on top of Hadoop. It was initially based on ideas of Google's Bigtable [26]: database comprises a large set of columns describing HTML files that represent rows of the table. HBase is designed for horizontal distribution of tables into regions that are managed by one server. Map-reduce techniques can be employed to process table rows. Abadi has shown in [20] that RDF can be efficiently stored by means of column-oriented stores.

IV. Challenges

While an accompany paper provides a comprehensive list of challenges on RDF storage managers, this section discusses a few challenges inspired by viewing TABLE I.

Listed RDF repositories recorded more varied values with $\mathcal{S}$ attributes than with $\mathcal{M}$ attributes in TABLE I. Most $\mathcal{M}$ attributes have *none* or *unknown* values. This means that researches so far have succeeded in achieving good performances by developing single process technologies. While practical semantic web applications tend to process large-scale data sets, solutions based on data distribution parallelism have become more popular. There will be room for further improvement of efficient query processes by developing multi process technologies considering the situation.

Caching techniques have not been researched that much. Only *Apache Jena* and *SW-Store* reported confirming the efficiency of caching techniques. Those performances depend upon types of queries and the number of different queries. Technologies for automatic investigation and classification of processing queries might become important to utilize caching technologies.

Many researches have been carried out for developing efficient join algorithms with index structures. This area has a long history in the research of database management systems [27]. While the accumulated RDF data-set is rapidly growing and SPARQL queries are basically constructed from joins of triple patterns, join operations will be applied more strongly in semantic web applications. The multi-process technologies mentioned above might produce breakthroughs of efficient join operations.

Because the standard data access method for the RDF data-set is W3C's recommendation SPARQL, most RDF storage managers can accept SPARQL queries. The semantics of SPARQL is clearly described using RDF algebra [28], [29], [30]. SPARQL-based RDF storage managers rarely cause semantic mismatch due to the existence of proposed RDF algebras. These papers also reveal that some kinds of SPARQL expressions require huge computational cost. Most of these expressions are constructed by using the *OPTIONAL* operator. While this operator was introduced to make the query language convenient enough, efficient processing of such queries will be one of the most crucial challenges of RDF storage managers.

V. Conclusion

This paper surveyed the RDF storage manager implementations based on the local cache approach by introducing the systematic classification structure $RSM(\mathcal{S}, \mathcal{M})$. This classification was applied to *3store, 4store, Virtuoso, RDF-3X, Hexastore, Apache Jena, SW-Store, BitMat, AllegroGraph, and Hadoop/HBase*. The RSM structure was presented for each of them, so detecting the differences between them became easy. By having the $\mathcal{M}$ part in RSM structure, the classification revealed that there will be room for further improvement of the efficient query process by developing multi process technologies.

VI. Ackowledgement

This work was supported by the Slovenian Research Agency and the ICT Programme of the EC under PlanetData (ICT-NoE-257641).

 ISBN: 978-1-61208-334-6

REFERENCES

[1] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - the story so far," Int. J. Semantic Web Inf. Syst., vol. 5, no. 3, pp. 1–22, 2009.

[2] K. Hose, R. Schenkel, M. Theobald, and G. Weikum, "Database foundations for scalable rdf processing," in Proceedings of the 7th international conference on Reasoning web: semantic technologies for the web of data, ser. RW'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 202–249. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033313.2033317

[3] A. Harth, K. Hose, and R. Schenkel, "Database techniques for linked data management," in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 597–600. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213909

[4] S. Sakr and G. Al-Naymat, "Relational processing of rdf queries: a survey," SIGMOD Rec., vol. 38, no. 4, pp. 23–28, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1815948.1815953

[5] "Triplestore," http://en.wikipedia.org/wiki/Triplestore, 2013, [retrieved Dec. 2013].

[6] "Largetriplestores," http://www.w3.org/wiki/LargeTripleStores, 2011, [retrieved Dec. 2013].

[7] S. Harris and N. Gibbins, "3store: Efficient bulk rdf storage," in 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03), 2003, pp. 1–15, event Dates: 2003-10-20. [Online]. Available: http://eprints.soton.ac.uk/258231/

[8] S. Harris, N. Lamb, and N. Shadbolt, "4store: The design and implementation of a clustered rdf store," in Proceedings of the The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems, 2009.

[9] O. Erling and I. Mikhailov, "Rdf support in the virtuoso dbms," in CSSW, 2007, pp. 59–68.

[10] ——, "Rdf support in the virtuoso dbms," in Networked Knowledge - Networked Media, ser. Studies in Computational Intelligence, vol. 221, 2009, pp. 7–24.

[11] OpenLink Virtuoso Universal Server: Documentation, OpenLink Software Documentation Team, 2009.

[12] T. Neumann and G. Weikum, "Rdf-3x: a risc-style engine for rdf," Proc. VLDB Endow., vol. 1, no. 1, pp. 647–659, Aug. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1453856.1453927

[13] ——, "The rdf-3x engine for scalable management of rdf data," The VLDB Journal, vol. 19, no. 1, pp. 91–113, Feb. 2010. [Online]. Available: http://dx.doi.org/10.1007/s00778-009-0165-y

[14] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix "bit" loaded: a scalable lightweight join query processor for rdf data," in Proceedings of the 19th international conference on World wide web, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772696

[15] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," Proc. VLDB Endow., vol. 1, no. 1, pp. 1008–1019, Aug. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1453856.1453965

[16] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds, "Efficient rdf storage and retrieval in jena2," Enterprise Systems and Data Management Laboratory, HP Laboratories Palo Alto, Tech. Rep. HPL-2003-266, 2003.

[17] B. McBride, "Jena: A semantic web toolkit," IEEE Internet Computing, vol. 6, no. 6, pp. 55–59, Nov. 2002. [Online]. Available: http://dx.doi.org/10.1109/MIC.2002.1067737

[18] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: implementing the semantic web recommendations," in Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, ser. WWW Alt. '04. New York, NY, USA: ACM, 2004, pp. 74–83. [Online]. Available: http://doi.acm.org/10.1145/1013367.1013381

[19] A. Owens, A. Seaborne, N. Gibbins, and mc schraefel, "Clustered tdb: A clustered triple store for jena," in WWW2009, November 2009. [Online]. Available: http://eprints.soton.ac.uk/266974/

[20] 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, ser. VLDB '07. VLDB Endowment, 2007, pp. 411–422. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325900

[21] ——, "Sw-store: a vertically partitioned dbms for semantic web data management," The VLDB Journal, vol. 18, no. 2, pp. 385–406, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1007/s00778-008-0125-y

[22] AllegroGraph 4.11 Introduction, Franz Inc., 2013.

[23] "Franz' s allegrograph(r) sets new record on intel(r) xeon(r) e7 platform," http://www.franz.com/about/press_room/Franz-Intel_6-7-11.lhtml, 2011, [retrieved Dec. 2013].

[24] T. White, Hadoop: The Definitive Guide. O 'Reilly Media, Inc., 2009.

[25] L. George, HBase: The Definitive Guide. O 'Reilly Media, Inc., 2011.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323

[27] R. Ramakrishnan and J. Gehrke, Database Management Systems, 3rd ed. New York, NY, USA: McGraw-Hill, Inc., 2003.

[28] R. Angles and C. Gutierrez, "The expressive power of sparql," in Proceedings of the 7th International Conference on The Semantic Web, ser. ISWC '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 114–129. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88564-1_8

[29] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," ACM Trans. Database Syst., vol. 34, no. 3, pp. 16:1–16:45, Sep. 2009. [Online]. Available: http://doi.acm.org/10.1145/1567274.1567278

[30] M. Schmidt, M. Meier, and G. Lausen, "Foundations of sparql query optimization," in Proceedings of the 13th International Conference on Database Theory, ser. ICDT '10. New York, NY, USA: ACM, 2010, pp. 4–33. [Online]. Available: http://doi.acm.org/10.1145/1804669.1804675

 ISBN: 978-1-61208-334-6