Design of Distributed Storage Manager for Large-Scale RDF Graphs

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

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

***Abstract*—Storage and management of large-scale RDF repositories is a challange that may be compared to storage and management of large-scale HTML repositories. The main difference is in the modelling power of RDF data model comparing it to HTML hyper-graph model. RDF is much closer to the database data model and requires the capabilities of database management system rather than those offered by informational retrieval query engine. Triple-based storage systems seem to provide the functionality needed for storing RDF graphs. Triples are very natural means for the representation of graphs at various levels of abstraction. To cope with growing demand for querying large-scale triple-stores including up to several Tera triples we propose the use of massively parallel system that can be dynamically configured for particular queries into a set of parallel data-flow machines. The paper presents the design of large-scale triple-store database system *big3store*.**

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

I. INTRODUCTION

There exists a growing interest to gather, store and query data from various aspects of human knowledge including geographical data; data about various aspects of human activities (like music, literature, and sport); scientific data (from biology, chemistry, astronomy and other scientific fields); as well as data presenting the activities of governments and other important institutions.

There is consensus that data should be presented in some form of *graph data model*, where simple and natural abstractions are used to represent data as *subjects* and their *properties* described by *objects*, that is, by means of nodes and edges of a graph. Seeing this from the point of view of knowledge developed in the fields of data modeling and knowledge representation, all existing data models and languages for the representation of knowledge can be transformed, many times very naturally, to some form of a *graph*.

There exist a number of practical projects that allow for gathering and storing graph data. One of the most famous examples is Linked Open Data (LOD) project that gathered more than 32 giga triples from the areas such as media, geography, government, life sciences and others. The language employed for the representation of data is Resource Description Framework (RDF), which is a form of graph data model.

Storing and querying such huge amounts of structured data represent a problem that could be compared to the problem of querying huge amounts of text that appeared after the advent of Internet. The differences are in the degree of structure and semantics that data formats such as RDF and Web Ontology Language (OWL) encompass comparing them to HyperText Markup Language (HTML). HTML data published on Internet represents a huge hypergraph of documents interconnected with links. Links between documents do not carry any specific semantics except representing URIs.

Differently to HTML, RDF is a *data model* where all data are represented by means of triples (subject, predicate, object). In this format, one can represent entities and their properties in a similar way as provided in object-oriented models or AI frames. One can represent objects at different levels of abstraction: RDF can serve to model ordinary data, data modeling schemata as well as meta-data.

Primary modeling principle of RDF is assignment of special meaning to properties with selected names. In this way, we can define the exact meaning of properties that are commonly used to describe documents, persons, relationships and others. *Vocabularies* are employed to standardize the meaning of properties. For example, Dublin Core [5] project defined a set of common properties of things. Next, XML-schema [24] vocabulary defines the properties that can specify types of objects. Furthermore, vocabularies of properties and things are used to define higher-level data models realized on top of RDF. Such examples include, RDF Schema [17] as well as OWL [16] that provide object-oriented data modeling facilities and constructs for the representation of logic.

A. Challenges in storing and querying RDF

The amount of data in the form of triples gathered worldwide is expected to grow further towards peta (i.e., 10^{15}) triples so that existing techniques for storing and accessing data will have to be adapted. Something similar appeared after the development of Internet, where search engines had to cope with huge hypergraph of documents including many giga documents. It seems a natural choice to tend to use similar methods to deal with the problem.

The leading idea of our approach is the use of massively parallel systems where data and query processing are distributed to many data servers. The challenges and problems that will be addressed are the following.

1) Definition of namespace of RDF triple-store,
2) Automatic distribution and replication of RDF data,

 ISBN: 978-1-61208-334-6

3) Intelligent distribution of query processing,
4) Dynamic updates in RDF storage manager,
5) Multi-threaded architecture of query executor, and
6) Distributed cache for query executor.

The first problem deals with definition of methods for naming entities and triples of database that will allow efficient way of managing huge amounts of structured data distributed and replicated to thousands of servers as one uniform name space. Naming schema must include ways to dynamically allocate data server for the execution of query optimally with regards to distance of data server in the network and the execution load of replicas.

The second challenge is about the design of schemas for distribution and replication of data to distributed servers so that optimal computation load is achieved. While manual distribution may seem possible, the storage of peta triples with data from all areas of human interest may require automatic distribution of data, which must be placed on distributed data server in such manner that query execution will be balanced among the servers.

Large number of servers that store distributed database require intelligent distribution of query processing to achieve appropriate response time to queries. This is covered by challenge 3. The uniform distribution schemas like hashing do not take into account semantics and structural properties of data resulting in the distribution where data on particular subject is scattered to too many data servers. The distribution of data based on semantics of data may result more efficient configuration of data servers for fast execution of queries.

In light of recent proposals for architecture of super-computers presented in [8] and by using the knowledge from the area of distributed query processing, we propose the use of global distributed query optimization, which results in optimal distributed query tree and a configuration of data servers forming a fast *dataflow machine*. Similar to super-computer systems the execution is comprised of two phases: in first phase the program or query, in our case, is optimized resulting specific dataflow machine configuration, and, in the second phase, the program executes on the specific hardware configuration, or in our case, on selected configuration of data servers.

Challenge 4 deals with updates of RDF databases. While most RDF data published through Linked Data community are stable, some portions of data are dynamically updated or found. Examples of such data would represent stock data, scientific data, or data presenting the state of institutions. With the growth of RDF databases the problem of updating RDF databases will become more important. Triple-store including very large quantities of data must be designed to provide capabilities for keeping track of changes in existing datasets as well as adding new RDF datasets.

Large-scale parallel computer systems can be recently constructed using commodity hardware that includes multi-processor systems and multi-threaded CPUs. It becomes more demanding to design triple-store architecture that maximizes query execution performance by utilizing concurrency of processes or threads running on large clusters of servers equiped with multiple processors. This problem is the topic of challange 5. We will provide the design of big3store, which exploits process and thread parallelism, by constructing custom parallel architecture of big3store using programming constructs of Erlang.

Finally, large-scale distribution of data and query processing in big3store calls for efficient architecture of *memory hierarchy* that will exploit locality of data. The design of local cache of data servers is presented as challange 6. The leading idea of architecture of local cache will be its tight interrelation with query processor system, which will tend to tie data servers to particular users, and for processing particular portion of data. Data gathered in a cache of data server will contain "local" data most probably needed for processing subsequent queries assigned to a given data server.

B. Outline

The rest of the paper is organized as follows. Section II presents architecture of RDF storage manager big3store. Storage manager is distributed to an array of servers including front servers and data servers, as described in Section II-A. Distribution of RDF database is discussed in Section II-B. Functions of front servers and data servers are described in Sections II-C and II-D. Some implementation aspects of big3store are presented in Section III. In particular, we describe distributed cache in Section III-A, distributed query execution in Section III-B, distributed query optimization in Section III-C, and architecture of dynamic updates in Section III-D. Related work is presented in Section IV and concluding remarks are given in Section V.

II. Architecture of RDF storage manager

To provide fast access to big RDF databases and to allow heavy workload storage manager has to provide facilities for flexible distribution and replication of RDF data. Storage manager has to be re-configurable to allow many servers to work together in a cluster and to allow for different configurations of clusters to be used when executing different queries.

Storage manager for big RDF databases should be based on SPARQL and on algebra of RDF graphs [20]. To provide more general and durable storage manager its design should be based on ideas of graph databases [2]. Such a design would allow adding interfaces for popular graph data models, besides RDF, to be added later.

A. Storage manager as cluster of data servers

Possible distribution and replication is crucial for the design of storage manager to be available globally and to provide heavy workload that is to be expected if LOD data is going to be used by masses.

Heavy distribution and replication is currently possible because of the availability of inexpensive commodity hardware for servers with huge RAM (1-100GB) and relatively large disks. The same idea was used by Google while bootstrapping and remains to be the main design direction for Google data centers [9].

As further detailed in the sequel, cluster of data servers can be easily configured into very fast data-flow machine answering a particular SPARQL query. Similar idea appears

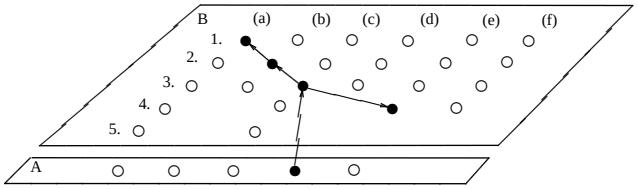

Figure 1. Configuration of servers for particular query.

recently in the area of super-computers [8], where advances in hardware technologies allow preprocessor of compiler to configure hardware facilities for a specific program. Program then runs on specially configured hardware that gains considerable speed.

The leading idea for distribution of SPARQL query processing is splitting SPARQL query into parts that are executed on different data servers in such way that the processing time of query is minimal. Data servers executing parts of SPARQL query are connected by streams of data to form cluster configuration defined for a particular SPARQL query. As with super-computers based on configuring intelligent hardware we also have strict separation between two phases: compiling the program into hardware configuration, and executing the program on selected hardware configuration.

Figure 1 presents a cluster composed of two types of servers: *front servers* represented as nodes of plane A, and *data servers* represented as nodes of plane B. Data servers are configured in *columns* labeled from (a) to (f). Complete database is distributed to columns where each column stores a portion of the complete database. The methods for the distribution of RDF data are discussed in the following sections.

Portion of database stored in a column is replicated into rows labeled from 1 to 5. The number of rows for a particular column is determined dynamically based on the query workload for each particular column. More heavy load we have on a given column more row data servers will be chosen for replication. The particular row used for execution of a query is selected dynamically based on current load of servers in a column.

A particular cluster configuration for answering a particular SPARQL query is programmed by front servers where also the optimization of SPARQL query takes place. Front server, receives SPARQL query, parses it to query tree and performs optimization based on algebraic properties of SPARQL set algebra operations. Parts of query tree are sent to internal data servers to define cluster configuration used for particular query execution.

B. Data distribution

The schema for distribution of RDF data to a cluster of data servers has to be designed very carefully. The distribution of RDF data in local data clusters has to be transparent from outside world. Ideally, RDF data would be distributed automatically aiming to distribute transaction load optimally to data servers forming cluster on the basis of the transaction load in a given time period.

RDF data stored in a data center is distributed to *columns* of data servers that form the cluster. Each data server includes a triple-store accessible through TCP/IP. Each column is composed of an array of data servers referred to as *rows* that are the replicas storing the same portion of big3store database.

Distribution of RDF data to columns can be defined in more ways. Firstly, data can be split manually by assigning larger datasets (databases) to columns. An example of such dataset may be dbpedia. This may be practical solution used in the initial phase of big3store implementation. Secondly, RDF data can be split to columns automatically by using SPARQL queries as the means to determine groups of RDF triples that are likely to be accessed by one query. In this context, RDFS classes are employed as the main subject of distribution as suggested in [19]. Groups of classes that are usually accessed together are assigned to *columns* where class instances are stored.

The benefits of splitting a triple store in more separate data stores (tables) has been shown ba Yan et al. in [25]: queries can be executed few times faster. The reason for this can only be the size and height of indexes defined for tables representing triples. This means that few-times less blocks have to be read from database if RDF data is distributed to different tables.

There are two points where automatic reconfiguration of RDF database can be implemented. Firstly, complete database may be automatically distributed into columns as described above. Secondly, the degree of replication of portion of database stored in a column has to be determined. In other words, we have to determine how many rows (replicas) do we need to process queries targeting particular column.

C. Front servers

Front servers are servers where SPARQL queries initiated by remote user are accepted, parsed, optimized and then distributed to data servers.

SPARQL parser checks the syntax of query and returns diagnosis to the user as well as prepare the query tree for the optimization phase. The most convenient approach to optimize SPARQL query is to transform queries into algebra and use algebraic properties for optimization. Algebra of RDF graphs [20] designed for big3store is based on work of Angles and Gutierrez [1] and the work of Schmidt et al. [22].

Algebra of RDF graphs reflects the nature of RDF graph data model. While it is defined on sets, the arguments of algebraic operation and its result are RDF graphs. Furthermore, expressions of RDF graph algebra are graphs themselves. Triple patterns represent the leafs of expressions. Graph patterns are expressions that stand for graphs with variables in place of some nodes and edges.

To be able to ship partial results of distributed query tree among data servers algebra of RDF graphs use operation `copy` introduced by Daniels et al. in [4]. Operation `copy` can be well integrated with operations defined on graphs due to simple set of algebraic rules that can be used for `copy`.

Global query optimizer will be based on rules and a form of dynamic programming algorithm for optimization of algebraic expressions [19]. Most of rules that apply to relational query optimization can be used for graph patterns. The operation `copy` also has a well defined set of rules that

can be integrated with rules for relational operations. The query optimization algorithm performs beam search guided by query cost estimation. The statistics of big3store distributed database stored with metadata server.

The result of query optimization for a given SPARQL query is a query tree where operations `copy` are placed optimally representing the points where triples are shipped from one data server to another one. The global query is therefore split into parts that are executed on different data servers. Initially, front server sends a query to a data server from a column that includes data needed to process top level of query tree. Note that all query parts are already in optimized form.

D. Data servers with local triple-store

In this section, we present the main features of distributed query evaluation. Firstly, we give a general view of the evaluation of distributed query. Next, we present some properties of local triple-store and the evaluation of queries in local triple-store.

Evaluation of distributed query

The primary job of data server is to evaluate query tree received from front server or some other data server. Query tree includes detailed information about access paths and methods for implementation of joins used for processing the query. We refer to such query tree as *annotated query tree*. Data server evaluates annotated query tree as it is without further optimization.

Triple store of data server accepts queries via TCP/IP and returns results to the return address of calling server. The communication between calling server and a given data server is realized by means of streams of triples representing results of query tree evaluation. When needed, the materialization of stream results is handled by calling server.

Query tree can include parts that have to be executed on some other data servers since data needed for a particular query part is located at some other columns. Such query parts are represented by query sub-trees with root nodes that denote operation `copy`. Again, query sub-trees can include more instances of operation `copy`, so the resulting structure of data servers constructed for a particular SPARQL query can form a tree.

Since operation `copy` is implemented by using stream of triples the query parts that form complete query tree can execute in parallel. While data server processing query sub-tree is computing the next triple to be consumed by a given data server, this data server can process previously read triple or perform some other task like accessing local triples. Moreover big3store can process many query parts in parallel functioning as a parallel dataflow machine.

Local evaluation of queries

Let us now present the evaluation of query on local data server. Let say that data server receives an annotated query tree `qt`. Recall that `qt` includes information about access paths to tables of triples and algorithms to be used for implementation of algebra operations.

Local triple-store includes the implementation of algebra operations and implementation of access paths, i.e., methods for accessing possibly indexed tables of triples. Algebraic operations are: selection with or without the use of index; projection; set operations union, intersection and difference; and variants of nested-loop join with or without index where the index is either index supporting equality joins or range queries.

Non-distributed storage manager for storing triples and indexes for accessing triples has to deal with very similar problems that appear in relational and object-relational storage managers. Since triple-stores are designed mainly around a table with three or four columns we propose to use existent implementation of local storage manager that implements solutions from existent relational and object-relational database technology.

We use local database management system of Erlang called *Mnesia* to store tables of triples. Mnesia includes high-level functions for accessing data stored in possibly distributed tables. Although Mnesia does not support SQL, it provides many practically useful features for distributed environment. Any table can be configured as RAM table or disk table. It supports horizontal partitioning for large tables and transaction control for distributed table operations. Tables can be reconfigured dynamically. Any Erlang object (complicated data structure) can be stored in Mnesia. If a local triple table is small enough, we might construct a fast in-memory storage using Mnesia's direct access functions. If a local table have to store large amount of triples, we might construct distributed and partitioned triple table using safe and robust transactions supporting parallel operations for distributed repositories.

Let us now present also the implementation of operation `copy` in more detail. Operation `copy` implements a stream between two data servers. The stream is realized by first initiating the execution of sub-tree of `copy` (i.e., query part) and requesting that the results are sent back to calling data server by means of a stream. On caller side access to the stream, i.e., the results of operation `copy`, is realized as access method that reads triples from the stream. that

III. On implementation of big3store

The initial prototype of big3store is currently under development in a high-level programming language Erlang. Erlang provides rich set of constructs convenient for distributed programming and offers abstract programming environment to allow rapid-prototyping. Successful implementation of initial prototype will allow gradual improvement of big3store efficiency that can result in a production version of the system.

A. Distributed cache

One of the most important principles used in database management systems is to implement some form of memory hierarchy where data read from the slow media is cached by faster media. In this way, the access to slow media may speed-up significantly. Implementation of distributed query execution on cluster of data servers with huge quantities of RAM calls for the use of memory hierarchy, i.e., exploitation of data fetched or pre-fetched from the disk and then stored in main memory.

 ISBN: 978-1-61208-334-6

Local cache of data servers can be in Erlang environment implemented by means of in-memory tables that store triples read from disk tables. Access to tables storing triples can be implemented by using additional database layer hiding the access to in-memory tables before reading data from disk. Such database layer would also allow seamless integration of other database management systems besides Mnesia to be plugged to query executor.

The problem of using local caches on data servers is somehow similar to the problem of scheduling on multiprocessor systems that have access to RAM via bus. After a process is executed on a particular processor the cache is loaded with data used in execution of process. Similarly, after a query tree is executed on a particular data server, cache is loaded with data used in the execution of given query.

In the case of multiprocessor scheduling, the next invocation of the same process should be executed on the same processor that includes data used in previous invocations. In the case of distributed query processing, if we select the same data server for processing the next query in the session of particular user we will most likely find some of data needed for this query already in the cache. The algorithm that selects the most appropriate row data server in a given column must therefore takes into account the affinity of user sessions to particular data servers.

The solution proposed in the area of process and thread scheduling is to use two level scheduling. On the first level process is, after creation, associated with particular processor. On the second level of scheduling the processes associated with particular processor are scheduled as in the case of uniprocessor system.

While the task of particular query can be compared to process, we can also compare the access to database system (albeit local to each data server) to the access to common RAM in the case of multiprocessor scheduling. Seeing this from the point of view of distributed query processing, user sessions are associated to data servers while we have to take care of balanced distribution of workload. This may mean that we can expand the set of data servers in a column to be employed for particular user session.

B. Distributed query execution

Whether or not a column local repository has indexes, the whole storage management system should perform distributed query executions considering load regulations. While there are many possible solutions for the load regulation problems, Erlang/OTP programming environment may provide a convenient solution that is suitable for developing initial prototype rapidly.

In order to regulate task loads of clustered column local repositories, fixed number of `gen_server` (general server library of Erlang/OTP) processes are invoked on each physical server. A `gen_server` can update server activity codes dynamically through inter-process messages. The storage management system's bootstrap process initializes some `gen_server`s as front server processes and others as data server processes. The bootstrap process distribute data-to-data server processes according to the column configuration. When a front server process accept a query, it divides the query into sub queries and sends them to idling data server processes according to query optimization algorithms considering efficiency and load regularity.

If a data server process should have indexes, it is a good solution to implement the process as a `gen_server` that calls Mnesia (distributed DBMS for Erlang) library functions internally for processing queries. When a data server process has to replicate to another physical server (copy operation), following steps are executed.

1) Find a remote physical server that has enough available CPU and memory resources (low load), and runs at least one idling data server process.
2) Replicate the Mnesia database instance used by the data server process to the remote physical server.
3) Serialize the `gen_server` implementation codes of the data server process, and install it on an idling data server process on the remote physical server.

The results of queries produced by data server processes are translated as streams. Because `gen_server` model includes message waiting loop as default functionality, it is easy to code synchronous translation of bunch of result elements. Additional codes for implementing FIFO buffer may be enough to make front and data server processes to communicate via triple streams.

C. Distributed query optimization

Query optimization takes place on front server. SPARQL query is parsed and converted into algebra of RDF graphs. Algebra expression is converted into query tree representation, which serves as the basic data structure used in the process of optimization, cost estimation and query evaluation. The design of query processing is rooted in the design of query processor Qios [18], [19], [21].

Algebra of RDF graphs is based on relational algebra extended with operations specific for graphs. The operations are *selection*, *projection*, a form of *join*, set operations *union*, *difference* and *intersection*, and operation *optional*. Finally, algebra of triples includes operation *copy*, which allows for shipping sets of triples among data servers.

Query optimization is based on rules including pushing *projection* and *selection* towards the leafs of query tree, associativity and comutativity of *join*, rules for operations *optional* and *copy*, which integrate well with rules for relational operations. Rules are represented as patterns of query trees that stand for input and output of rule transformation.

General form of query optimization is rooted in an instance of dynamic programming technique called *memoization*. Query tree is optimized by first optimizing the children of query tree root and then by using rules to transform root of query tree. Optimized query sub-trees are inserted into the appropriate *equivalence classes* and their cost is stored for further use. Since the space of all hypotheses (query trees) is too big to explore completely, sub-optimal additions to the basic form of dynamic programming can be employed. For instance *beam search* selects at each point of optimization only the most promising alternatives for query transformation.

 ISBN: 978-1-61208-334-6

D. Architecture for managing dynamic updates

Although most amount of RDF data are stable, some data are dynamically updated. It makes difficult for data manager to build index on the set including updated data. If data managers are distributed on cluster of several PC servers, data statistics and query patterns strongly influence means for distributing and caching data over the cluster. The data might change or grow dynamically. The system load may also influence distribution configuration and cache lifetime. These make the problem more complicated. Occasionally, new RDF links might be discovered through unrelated search processes.

RDF repositories should include efficient means for accessing dynamically updated data. If elements in a data-set are updated frequently, computational cost of indexing the data-set may exceed speed-up benefit of accessing operations. We will have the threshold updating frequency by investigations with practical experiments. Because columns exceeding the threshold should not have indexes, they should be stored in a special type of repository for efficient retrieval.

One possible solution is to implement triple-store local to *column* by a set of tiny proactive on-memory processes. Such repositories can be easily coded using Erlang programming language. Adding, deleting, and modifying operations only require accesses to the target triple-store processes. The manager of column triple-store can broadcast query messages to local triple-store processes. Each process may respond asynchronously to the caller, if the query matches its contents. For the manager receives answers asynchronously, it can provide query results as a stream to its caller process. Copy operations can be easily implemented using the process dictionary serialization function of Erlang programming language. Triple-store processes execute copy or modification reflection operations independently. If the process is coded to execute those operations only under low load situations, the repository may have a method for high load tolerance.

IV. RELATED WORK

This section presents some of more important systems for querying RDF data including: 3store, 4store, Virtuoso, RDF-3X, and Hexastore; see survey presented in [14] for a more complete overview of RDF storage managers.

a) 3store: 3store [10] was originally used for semantic web applications in particular for storing hyphen.info RDF dataset describing computer science research in UK. Final version of database consisted of 5000 classes and about 20 million triples. 3store was implemented on top of MySQL database management system. It included simple inferential capabilities, e.g. class, sub-class, and sub-property queries mainly implemented by means of MySQL queries. Hashing is used to translate URIs into internal form of representation.

Query engine of 3store used RDQL query language originally defined in frame of Jena project. RDQL triple expressions are first translated into relational calculus. Constraints are added to relational calculus expressions and they are translated into SQL. Inference is implemented by a combination of forward and backward chaining computing the consequences of asserted data.

b) 4store: 4store [11] was designed and implemented to support a range of novel applications emerged from semantic web. RDF databases were constructed from web pages including people-centric information resulting ontology with billions of RDF triples. The requirements were to store and manage $15x10^9$ triples.

4store is designed to operate on clusters of low-cost servers. It 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 to distribute data to a cluster of 64-bit multicore x86 Linux servers each storing a partition of RDF data. The architecture of cluster uses ”Shared Nothing” architecture. Cluster nodes are divided into processing and storage nodes. Data segments stored on different nodes are determined by a simple formula calculating RID of subject modulo number of segments. The benefits of such design is parallel access to RDF triples distributed to nodes holding segments of RDF data. Furthermore, segments can be replicated to distribute total workload to the nodes holding replicated RDF data. The communication between nodes is directed by processing nodes via TCP/IP. There is no communication between data nodes.

The 4store query engine is based on relational algebra. Primary source of optimization is the conventional ordering on the joins. 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: Virtuoso [6], [7], [15] is a multi-model database management system based on relational database technology. The approach of Virtuoso is to treat triple-store as a table composed of four columns. The main idea of the approach to management of RDF data is to exploit existing relational techniques and to add functionality to RDBMS in order to deal with features specific to RDF data. The most important aspects that were considered by Virtuoso designers are: extending SQL types with RDF data type, dealing with unpredictable sizes of objects, providing efficient indexing and extending relational statistics to cope with RDF store based on single table, as well as efficient storage of RDF data.

Virtuoso integrates SPARQL into SQL. SPARQL queries are translated into SQL during parsing. SPARQL has in this way 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: Triple-store RDF-3X presented by Neumann and Weikum [12], [13] builts 6 independent indexes of SPO, SOP, OSP, OPS, PSO and POS (for subject, property and object columns) from one large triple table. The indexes are compressed using a byte-wise method that was carefully chosen to improve query process performance. Join re-ordering is used to optimize query process. The optimization uses selectivity statistics calculated for given queries using selectivity histograms and frequent path statistics. Although it equips a

 ISBN: 978-1-61208-334-6

table to treat long URI strings as simple ids, Atre et al. in [3] points that its search performance was very bad. RDF-3X system was compared with PostgreSQL and MonetDB. The benchmark data contained Barton data. RDF-3X exceeded other systems with large margins. The source code is available for non-commercial purposes.

e) Hexastore: Hexastore [23] approach to RDF storage system uses triples as the basis for storing RDF data. The problems of existent triple-stores pursued are the scalability of RDF databases in distributed environment, and complete implementation of query processor including query optimization, persistent indexes, and other topics provided by database technology.

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

V. Conclusion

The design of large-scale storage manager for RDF is presented in the paper. The presented work is focused to the definition of most important design directions and implementation decisions of big3store. Hardware architecture of big3store is based on massive parallel array of data servers arranged into columns. Rows of columns are replicas, i.e., data servers that store a portions of big3store database. Distribution of complete big3store database is guided by semantic information used to group RDF triples.

The initial prototype in Erlang programming environment is currently under development. Erlang provides efficient programming constructs for implementation of massively parallel systems. Distributed query evaluation system of big3store, for instance, will use processes to represent query nodes that stand for operations of algebra of RDF graphs. SPARQL queries, optimized by means of programming technology provided by relational database systems, are translated to data-flow machine composed of Erlang processes. Therefore, array of distributed data servers becomes resource for optimized allocation of data-flow machines executing individual queries.

Acknowledgment

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

References

[1] R. Angles and C. Gutierrez. The expressive power of sparql. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, pages 114–129, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.

[3] 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*, WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.

[4] D. Daniels, P. G. Selinger, L. M. Haas, B. G. Lindsay, C. Mohan, A. Walker, and P. F. Wilms. Introduction to distributed query compilation in r*. IBM Research Report RJ3497 (41354), IBM, June 1982.

[5] Dublin core metadata inatiative. http://dublincore.org/, 2013.

[6] O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In *CSSW*, pages 59–68, 2007.

[7] O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In *Networked Knowledge - Networked Media*, volume 221 of *Studies in Computational Intelligence*, pages 7–24, 2009.

[8] M. J. Flynn, O. Mencer, V. Milutinovic, G. Rakocevic, P. Stenstrom, R. Trobec, and M. Valero. Moving from petaflops to petadata. *Commun. ACM*, 56(5):39–42, May 2013.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[10] S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage. In *1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, pages 1–15, 2003. Event Dates: 2003-10-20.

[11] 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.

[12] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proc. VLDB Endow.*, 1(1):647–659, Aug. 2008.

[13] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, Feb. 2010.

[14] K. Nitta and I. Savnik. Survey of rdf storage managers. Technical Report (In preparation), Yahoo Japan Research & FAMNIT, University of Primorska, 2013.

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

[16] Owl 2 web ontology language. http://www.w3.org/TR/owl2-overview/, 2012.

[17] Rdf schema. http://www.w3.org/TR/rdf-schema/, 2004.

[18] I. Savnik. Qios: Querying and integration of internet data. http://osebje.famnit.upr.si/ savnik/qios/, 2009.

[19] I. Savnik. On using object-relational technology for querying lod repositories. In *The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications*, DBKDA 2013, pages 39–44, Jan. 2013. Dates: from January 27, 2013 to February 1, 2013.

[20] I. Savnik and K. Nitta. Algebra of rdf graphs. Technical Report (In preparation), FAMNIT, University of Primorska, 2013.

[21] I. Savnik, Z. Tari, and T. Mohoric. Qal: A query algebra of complex objects. *Data & Knowledge Engineering*, 30(1):57 – 94, 1999.

[22] M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[23] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, Aug. 2008.

[24] Xml schema. http://www.w3.org/XML/Schema, 2012.

[25] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficient indices using graph partitioning in rdf triple stores. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1263–1266, Washington, DC, USA, 2009. IEEE Computer Society.

 ISBN: 978-1-61208-334-6