A Distributed Query Execution Method for RDF Storage Managers

Kiyoshi Nitta[1], Iztok Savnik[2,3]

[1] Yahoo JAPAN Research, Tokyo, Japan
[2] University of Primorska &
[3] Jozef Stefan Institute, Slovenia

knitta@yahoo-corp.jp, iztok.savnik@upr.si

Abstract. A distributed query execution method for Resource Description Framework (RDF) storage managers is proposed. Method is intended for use with an RDF storage manager called *big3store* to enable it to perform efficient query execution over large-scale RDF data sets. The storage manager converts SPARQL queries into tree structures using RDF algebra formalism. The nodes of those tree structures are represented by independent processes that execute the query autonomously and in a highly parallel manner by sending asynchronous messages to each other. The proposed data and query distribution method decreases the amount of inter-server messages during query executions by use of the semantic properties of RDF data sets.

1 Introduction

There is a growing interest to gather, store, and query data from various aspects of human knowledge. Such data includes geographical data; data about various aspects of human activities such as music, literature, and sport; scientific data from biology, chemistry, astronomy, and other scientific fields; and data related to the activities of governments and other influential institutions.

There is a consensus among Semantic Web researchers that data should be presented in some form of *graph data model* in which simple and natural abstractions are used to represent data as *subjects* and their *properties* described by *objects* — that is, by means of the nodes and edges of a graph. Considering 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, in many cases quite naturally, into some incarnation of a *graph*.

A number of practical projects that allow for the gathering and storing of graph data already exist. One of the most famous examples is the Linked Open Data (LOD) project, which gathered more than 32 giga triples from areas including the media, geography, government, life sciences and others. In that project, the Resource Description Framework (RDF), which is a form of graph data model, was used to represent the data.

Storing and querying such huge amounts of structured data has created a problematic scenario that could be compared to the problem of querying huge amounts of text

that appeared after the advent of the Internet. The differences are in the degree of structure and semantics that data formats such as RDF and OWL encompass compared to HTML. HTML data published on the Internet represents a huge hypergraph of documents interconnected with links. Links between documents do not carry any specific semantics except those representing URIs.

In contrast to HTML, RDF is a *data model* in which all data is represented by triples (subject, predicate, object). In this format, we can represent entities and their properties similarly to object-oriented models or AI frames. Moreover, we can represent objects at different levels of abstraction: RDF can model not only ordinary data and data modeling schemata but also meta-data.

The primary modeling principle of RDF is the 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. The Dublin Core [6] project is an example of defining a set of common properties of things. The XML-schema [23] vocabulary defines the properties that can specify types of objects, and the vocabularies of properties and things are used to define higher-level data models realized on top of RDF. The RDF Schema [17] and the OWL [16] are two examples of providing object-oriented data modeling facilities and constructs for the representation of logic.

The contributions of this paper are as follows. Firstly, we propose an architecture of RDF query processor that gives rise to novel *distributed query execution method* for RDF storage managers. While the architecture is rooted in relational database technologies, we propose flexible and highly parallel solution allowing allocation and execution of very large number of queries expressed as data-flow programs on cluster of servers. Secondly, we propose the use of *semantic distribution* of triples that distributes data based on relationships of triples to the conceptual schema. Semantic distribution provides very general means for partitioning triple-store into non-overlapping portions, and, allows efficient distribution of query processing.

The paper is organized as follows. Section 2 presents architecture of storage system big3store. This section introduces data distribution method and main building blocks of storage system such as front servers and data servers. Conceptual design of query execution is presented in Section 3. Query execution engine is based on efficient parallelisation of query trees. Related work is described in Section 4. Finally, concluding remarks are given in Section 5.

2 Architecture of big3store

To provide fast access to big RDF databases and to allow a heavy workload, a storage manager has to provide facilities for flexible distribution and replication of RDF data. To this end, the 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.

The storage manager for big RDF databases should be based on SPARQL and on the algebra of RDF graphs [19]. To provide a more general and durable storage manager,

its design should be based on the concept of graph databases [2]. Such design would enable adding interfaces for popular graph data models other than RDF to be added later.

2.1 Storage manager as cluster of data servers

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

Heavy distribution and replication is currently possible because of the availability of inexpensive commodity hardware for servers with huge RAM (1–100 GB) and relatively large disks. The same concept was used by Google while bootstrapping and remains the main design direction of Google data centers [10].

As we discuss in more detail later, a cluster of data servers can easily be configured into a very fast data-flow machine answering a particular SPARQL query. A similar idea has recently appeared in the area of super-computers [9], where advances in hardware technologies now allow compiler preprocessors to configure hardware facilities for a specific program. The program then runs on specially configured hardware that gains considerable speed.

The leading idea for the distribution of SPARQL query processing is splitting a SPARQL query into parts that are executed on different data servers, thus minimizing the processing time. Data servers executing parts of the SPARQL query are connected by streams of data to form a cluster configuration defined for a particular SPARQL query. Similar to the way in which some super-computers are based on configuring intelligent hardware, we also have a strict separation between two phases: 1) compiling the program into a hardware configuration and 2) executing the program on the selected hardware configuration.

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

The portion of the 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. The heavier the load on a given column, the greater the number of row data servers chosen for replication. The particular row used for executing a query is selected dynamically based on the current load of servers in a column.

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

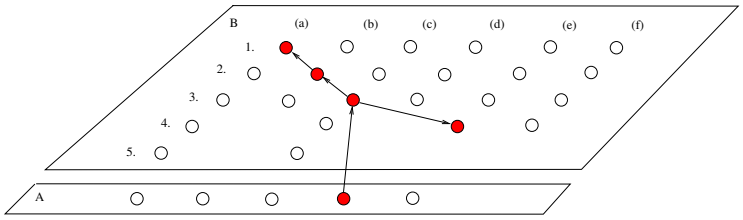

Fig. 1. Configuration of servers for a particular query

2.2 Data distribution

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 replicas storing the identical portion of the big3store database.

Distribution of RDF data to columns can be defined in more ways. First, data can be split manually by assigning larger data sets (databases) to columns. One example of such a dataset is DBpedia [4]. Second, RDF data can be split into 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 can be employed as the main subject of distribution, as suggested in [18]. Groups of classes that are usually accessed together can be assigned to *columns* where similar class instances are stored.

The benefits of splitting a triple store into separate data stores (tables) have been shown in [24]. Basically, queries can be executed a few times faster. The reason for this can only be the size and height of indexes defined for tables representing triples. This means that fewer blocks have to be read from a database if RDF data is distributed to different tables.

There are two scenarios in which the automatic reconfiguration of an RDF database can be implemented. First, a complete database may be automatically distributed into columns, as described above. Second, the degree of replication of the portion of the database stored in a column needs to be determined. In other words, we have to determine how many rows (replicas) are needed to process queries targeting a particular column efficiently.

2.3 Front servers

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

A SPARQL parser checks the syntax of a query and returns a diagnosis to the user as well as prepares the query tree for the optimization phase. The most convenient approach to optimizing a SPARQL query is to transform queries into algebra and then use the algebraic properties for optimization. The algebra of RDF graphs [19] designed for big3store is based on the work of Angles and Gutierrez [1] and of Schmidt et al. [20].

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

In order to ship the partial results of a distributed query tree among data servers, the algebra of RDF graphs uses operation `copy`, first introduced in [5]. Operation `copy` can be well integrated with operations defined on graphs due to the simple set of algebraic rules that can be used for `copy`.

A query optimizer rooted in relational rule-based query optimization has been proposed by Savnik in [18] for handling RDF queries. Similarly to our approach Schmidt and Lausen [20] also use relational rules for the optimization of SPARQL queries. Optimization in big3store is based on a variant of dynamic programming algorithm for optimizing the algebraic expressions called *memoisation*. Since the search space grows exponentially with the number of the rules, we experiment with beam search selecting only the most promising transformations. Query cost estimation, that is vital for guiding beam search, is also rooted in cost estimation of relational database management systems.

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. The global query is therefore split into parts that are executed on different data servers. Initially, the front server sends a query to a data server from a column that includes data needed to process the top level of the query tree. Note that all query parts are already in optimized form.

2.4 Data servers with local triple store

In this section, we present the main features of a distributed query evaluation. We first give an overview of the distributed query evaluation and then present some of the properties of the local triple store and the evaluation of queries within it.

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

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

The query tree can include parts that have to be executed on some other data servers if 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 additional instances of operation `copy` so that the resulting structure of data servers constructed for a particular SPARQL query can form a tree.

Since operation `copy` is implemented by using a stream of triples, the query parts that form a complete query tree can be executed in parallel. While the data server is processing the query sub-tree that computes the next triple to be consumed by a given data server, it can also process previously read triples and/or perform other tasks such as accessing local triples. Moreover, big3store can process many query parts in parallel as a parallel data-flow machine.

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

The local triple store includes the implementations of algebra operations and of access paths, i.e., methods for accessing indexed tables of triples. Algebraic operations include selection with or without the use of index; projection; set operations for union, intersection, and difference; and variants of nested-loop join with or without index, where the index supports either equality joins or range queries.

A non-distributed storage manager for storing triples and indexes for accessing triples has to deal with similar problems to those faced by relational storage managers. We use a local database management system called *Mnesia*, which is a part of Erlang programming language [3] distribution, to store and manage tables of triples, referred to as *triple-stores*. Triple-store of big3store is a table including four attributes: triple id, subject, property and object. Adding triple ids to triple-store is the decision that we expect will allow more consistent and uniform storage of various data related to triples, such as, named graphs and other groupings of triples, properties of triples (reification), and the like. Each triple-store maintains 6 indexes for accessing SPO attributes and additional index for triple ids.

We tend to use low levels of Mnesia storage manager including access to tables and indexes since optimization is performed by global query optimizer of big3store. Furthermore, lower levels of relational storage manager can be easily replaced by some other storage manager or even with file-based storage system. We relate this level of storage manager to data storage facilities of Hadoop [22]. Maps, for instance, represent main indexing mechanism of Mnesia while they are well comparable to Hadoop Maps. In this way, we achieve the simplicity of lower parts of storage manager in comparison to the complexity of RDBMs—this may represent a trend started with Hadoop. We can compare our work on compiling and executing high-level data-flow programs with programs and scripts written using Hadoop. Finally, for practical reasons, some features of Mnesia will be used for implementation of caching in big3store. Data about user context, including the results of all his queries, can be easily stored in Mnesia RAM tables increasing in this way significantly the speed of query evaluation.

Let us now give some more details about implementation of operation `copy`. As stated briefly before, operation `copy` implements a stream between two data servers. This stream is realized by first initiating the execution of a sub-tree of `copy` (i.e. a

query part) and then requesting that the results be sent back to the calling data server by means of a stream. On the caller side, access to the stream, i.e., the results of operation `copy`, is realized as an access method that reads triples from the stream.

2.5 Distribution of Triples and Triple Patterns

The main idea of *semantic distribution* is to distribute database triples on the basis of database schema. In other words, the distribution of triples to data servers (columns) is based on the relationship of triples to triple-base schema.

We suppose that we have triple-base such that complete schema for ground triples is defined and stored in database. An example of such RDF datasets is YAGO [13] which includes classes of all subjects, schema for all properties as well as taxonomy of classes.

In the sequel we will first present semantic distribution function in general. Class-based and property-based semantic distribution functions will be described in more detail. Some properties and trade-offs of semantic distribution function will be given.

Distribution function The distribution of triples can be achieved by means of a *distribution function* dist(), which maps a triple or a triple pattern into a set of column identifiers. Each column identifier represents a portion of the complete triple-base. The sizes of distributed portions must be similar.

The proposed method for semantic distribution is general since it allows various subsets of $\{S, P, O\}$ to be used as the means for distribution.

For instance, distribution can be defined on S part of the schema: instances of S's type c are stored in a column assigned to type c. Similarly, triples can also be stored on the basis of P part. In this case, each property has a column where its instances are stored. Furthermore, if we would like to separate properties defined for particular classes, distribution might be defined on the S and P parts of the triple schema.

Let us now consider two types of semantic distribution in more detail. Firstly, we present *class-based* semantic distribution which uses S part of triples, and secondly, we consider *property-based* semantic distribution which uses P part of triples.

Class-based distribution The first possibility of triple distributions is to distribute triples in partitions on the basis of the S part of triple schema, i.e., based on RDF class. A triple belongs to a RDF class if it describes the property of an instance of that class. The RDF class of the instance is determined by means of the $rdf{:}type$ property. We assume that all instances of classes have an $rdf{:}type$ relationship defined in a given triple store.

What are the effects of triple distribution based on classes in terms of query evaluation? In the case of queries related to the properties of two classes, the queries are evaluated on two different servers. In the case of queries tackling the properties of instances of three RDF classes, they are evaluated on three servers, etc.

In the case in which spreading the evaluation of queries to more data servers is desired, the properties pertaining to a particular class must not be stored in one column but rather must be distributed into additional columns.

Continuing to distribute the properties of RDF classes to different columns would in the end result in a distribution function based on the P-part of triples and not on RDF classes. This would make sense if we could determine experimentally that it is more efficient to distribute a query tree to data servers such that one query node is executed on one query node.

Property-based distribution The second type of semantic distribution can be defined on the basis of P part of triples, i.e., based on RDF properties. Since each ground triple includes the property part same as does the schema triple denoting given ground triple, it is easier to define the distribution based on properties. Columns are assigned to each property so that database triples are distributed uniformly to data servers.

In this case the distribution function can be simply defined by extracting P part of triple or triple pattern, and, then, mapping property to columns using predefined table. However, a problem appears in the case P part of triple pattern includes a variable. The only possible way to query properties when using distribution based on properties is by sending "broadcast" query to data servers of all columns.

Trade-offs of distribution function It remains to be determined experimentally what kind of distribution function behaves optimally for a given triple-base and query workload.

The first aspect of query evaluation that needs to be considered is weather it is better to store data and evaluate all query nodes related to a given class c on one data server or, is it better to split data and query nodes related to c to separate data servers (i.e., columns).

Another variable of query evaluation, that is not directly related to semantic distribution, is to determine how many query nodes should be assigned to one data server in average to give optimal performances.

In one extreme, complete query is executed on one data server and the other extreme is that each query node is executed on a separate data server, each of which is connected by streams. The optimal distribution of queries to an array of data servers may require assigning query nodes to data servers such that each data server executes an average of two to three query nodes.

The patterns in mapping nodes of query trees to data servers that achieve fast execution of query trees on a given triple-base need to be excavated through experiments. The patterns can be used as a target structure of query optimization. One way to do that is to use patterns for tuning parameters of query optimization such as cost of moving intermediate results of queries among data servers. Another way to use patterns in query optimization is to restrict search space by focusing search to queries that match excavated patterns.

3 Conceptual Design of Query Execution

The query execution module (QEM) of the big3store system takes query tree structures as inputs and produces streams of result messages as outputs. When a SPARQL query

is requested to be execute on the big3store system, the query is translated into a graph structure called a *query tree* (an example is shown in Figure 3). The query tree can be modified by the query optimizer module of big3store to make the structure better for efficient execution. While query trees are represented in some data structures in big3store, QEM takes query trees represented by processes. These processes are dynamically generated by the preparation procedure of the big3store system. Query trees are executed by those processes cooperating with other big3store processes. Therefore, QEM is not a single process but rather consists of several processes including query tree processes to be executed. These processes can be distributed to multiple physical server machines that are connected by an ordinary network. A process is identified by a combination of server id and process id.

While most parts of the big3store system have not been developed, the prototype of QEM was developed and tested using small example data on a single server (non-distributed) environment. The reason for developing QEM first is that the investigation of query execution efficiency seems to be the most important challenge. The next step of our research will be to experiment the QEM execution in distributed environments.

In this section, we introduce processes for performing such query executions and then give an example of the entire flow of a query execution.

3.1 Query tree

Process `query_tree` accepts requests for managing query nodes. A physical server machine has only one `query_tree` process in big3store. It manages active query node processes, which are described in the next subsection; accepts requests for creating and deleting query nodes on any server; and provides unique process ids in the server on which the `query_tree` is running for creating query nodes on the server.

3.2 Query node

Process `query_node` implements a query node that constructs a query tree with other query nodes to represent a query. A `query_node` process can run on any physical server machine on which a `query_tree` process is running. Each `query_node` process has its own hash table on which to store multiple property values. It provides put and get interfaces for accessing property values. While various types of relations between query nodes can be represented by properties, *parent* relations are used for representing the stem structures of query trees. Each query tree has a *root* query node that has no value for a parent property. Each query node excepting the root query node must have a parent property for representing its parent query node by a combination of server id and process id. A `query_node` process must have a type property that indicates an operation type of RDF algebra [19]. Currently, *triple pattern* and *join* types are implemented.

Triple pattern query node A triple pattern `query_node` process represents a matching condition for triples. It must have a *triple_pattern* property for representing a triple pattern that consists of IRIs, literals, or variables in subject, predicate, and object slots.

It can handle *list_vars*, *eval*, and *result* asynchronous messages. If it receives a *list_vars* message, it sends a list of variables contained in the triple pattern to its parent by a *construct_vars* asynchronous message. It also sets a *vars* property for reminding the list of variables. If it receives an *eval* message, it sends a *find_stream* asynchronous message to data server processes for invoking streamer processes that repeatedly send *result* messages to the triple pattern process, each of which contains a concrete triple matching the triple pattern. If it receives a *result* message, it sends the message to its parent query node.

Join query node A join `query_node` process represents a conjunctive condition of two query nodes. It must have *nodInner* and *nodOuter* properties for representing the target query nodes of the inner and outer edges, respectively. It can handle *eval*, *construct_vars*, and *result* asynchronous messages.

If it receives an *eval* message, it sends *list_vars* messages to its inner and outer query nodes for listing all variables that appear in the sub tree. At the same time, it sends an *eval* message to the outer query node. Granted query tree structures are left-deep style and only permit outer edges to have join query nodes (Figure 3). Therefore, this eval-propagation strategy successfully constructs a variable list for any granted query tree. If a join query node process receives *construct_vars* messages from inner and outer query nodes, it merges both variable lists and sends the merged list to its parent query node by another *construct_vars* message.

If it receives a *result* message from its outer query node, it sends synchronous messages to data servers for inquiring whether the triple set in the result message satisfies the join condition or not. A result message consists of an *alpha map* and a *val map*, the structures of which are shown in Tables 1 and 2, respectively. The alpha map associates the set of triples and their origin query nodes, each of which has a matching triple pattern, and the val map represents variable bindings that were determined by the set of triples. When the query node asks the data servers to process the result message, the node fetches a triple pattern from its inner query node, substitutes the val map variable bindings in the inner triple pattern, and sends the substituted triple pattern to the data servers. If the data servers find no matching triple, the node does nothing. Otherwise, the node generates new result messages for each matched triple and sends them to a parent query node asynchronously. The new result messages are made by a new alpha map and a new val map. The alpha map is added by an element that maps the found triple with the inner query node and the val map is added by variable bindings determined by the found triple.

Table 1. Alpha map structure

No.	Field name	Description
1	triple	triple id in the triple table
2	query_node	process id of corresponding query node

Table 2. Val map structure

No.	Field name	Description
1	variable	string_id coded id of the variable string
2	value	string_id coded id of the IRI or literal value

3.3 Other processes

There are several other processes that are necessary to execute QEM successfully.

Data server Each process of the `data_server` holds a chunk of triple data, and multiple data server processes are invoked in running the b3s server of a distributed configuration. A data server process dispatches storing and retrieving requests of held triple data. It also accepts requests for producing streams of data that match specific triple patterns. It uses a Mnesia table for storing triple data.

Streamer A data server streamer process is invoked by a data server process when it receives a request for generating streams for a given triple pattern. The streamer retrieves matching triples from the data server's triple table. After sending all stream data, the streamer closes the stream and terminates itself.

Map between string and id Process `string_id` maintains a mapping table and dispatches access operations for the table.

Translating all IRIs and literals into integer ids makes the triple tables smaller. However, the cost of processing the translation might create a bottleneck affecting the execution efficiency of distributed systems. One solution is to run multiple `string_id` processes in different servers, but this would also increase the number of translate operations for identifying the same strings between different `string_id` processes.

3.4 An Example

In this subsection, we describe a message stream flow using the example query shown in Figure 2 in order to explain the conceptual design of the big3store query execution mechanism.

```
SELECT * WHERE {
  ?c <hasArea>      ?a .
  ?c <hasLatitude>  ?l .
  ?c <hasInfration> ?i
}
```

Fig. 2. SPARQL query of **q01a**.

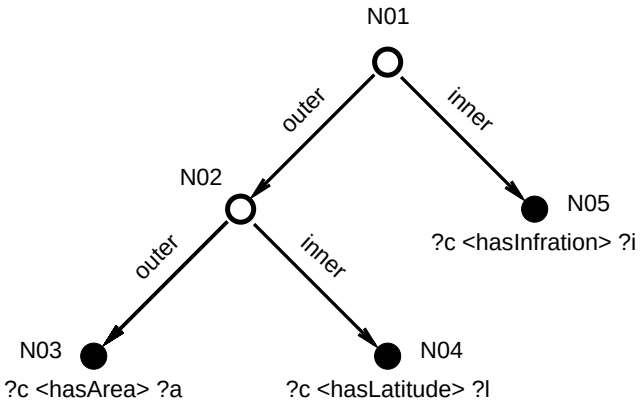

Fig. 3. Query tree structure of query **q01a**.

A query tree can be executed by sending an *{eval, Root, Parent}* asynchronous message to a `query_tree` process. The argument *Root* shows the *root* query node of a query tree to be executed. The `query_tree` process sends an *eval* message to the root node. The argument *Parent* shows a process that receives the result of the query. Figure 3 depicts the query tree structure of the example SPARQL query shown in Figure 2, where the black and white circles show the triple pattern and join query nodes, respectively. Query nodes are numbered N01, N02, ... in this figure for convenience. Query node N01 is a root join query node that is connected to query nodes N02 and N05 by outer and inner edges, respectively. Query node N02 is a join query node that is connected to query nodes N03 and N04 by outer and inner edges, respectively. Query nodes N03, N04, and N05 are triple pattern query nodes. Because query node N01 is the root of the query tree, it receives an *eval* asynchronous message for starting the execution of the query tree.

After receiving the initial *eval* message, the join query nodes propagate *eval* messages following outer edges. Figure 4 shows the messages sent by the processes for executing the query tree of Figure 3. As in Figure 3, black and white circles are used for representing the triple pattern and join query nodes. Each circle also represents an independent process in this figure. Two gray circles are added to represent a data server and a streamer processes. Edges drawn in solid lines show asynchronous messages and edges drawn in dashed lines show synchronous messages or function calls. After query node N01 receives an *eval* message, N01 sends another *eval* message to its outer query node N02. Query node N02 then performs the same action to N03. Because N03 is a triple pattern query node, it sends a *{find_stream, TriplePattern, QueryNode}* asynchronous messages to the data server process. The argument *TriplePattern* is the triple pattern that was set to N03. The argument *QueryNode* is used for specifying the caller of the message. It is N03 in this example. When the data server process receives the *find_stream* message, it invokes a new streamer process on the same physical server machine with

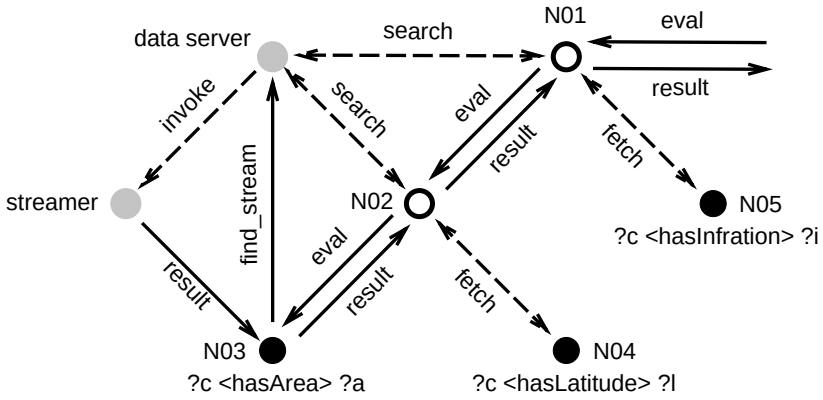

Fig. 4. Message stream flow of the execution of **q01a**.

the data server. The streamer process repeatedly sends *result* asynchronous messages to N03, each of which has a triple that matches the triple pattern of N03.

Each triple is represented in an *alpha map* entry in a corresponding result message. When query node N03 receives a *result* message, N03 checks a triple in the message for the selection condition. If the triple satisfies the condition, query node N03 sends the *result* message to its parent query node N02. When query node N02 receives a *result* message, N02 substitutes the triple pattern of N04 with *val map* variable bindings in the message. N02 then sends a synchronous *search* message to the data server process for retrieving triples that match the substituted N04 triple pattern. When query node N02 retrieves a matching triple from the data server, N02 modifies alpha and val maps in the received message and sends the new *result* message to its parent query node N01. Note that plural triples from the data server produce plural *result* messages from query node N02. Query node N01 performs actions similar to N02 for retrieving triples that match the triple pattern of N05. The answers for the query of Figure 2 are sent from query node N01 as a stream of *result* messages, each of which includes a set of triples that is a concrete solution of the query.

4 Related Work

In this section, we present some of the more relevant systems for querying RDF data, including 3store, 4store, Virtuoso, and Hexastore. See the survey presented in [14] for a more complete overview of RDF storage managers.

3store 3store [11] was originally used for Semantic Web applications, particularly for storing the hyphen.info RDF dataset describing 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 a MySQL database management system and included simple inferential capabilities (e.g., class, sub-class, and sub-property queries) mainly

implemented by means of MySQL queries. Hashing was used to translate URIs into the internal form of representation.

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

4store 4store [12] was designed and implemented to support a range of novel applications emerging from the Semantic Web. RDF databases were constructed from Web pages including people-centric information, resulting in ontology with billions of RDF triples. The requirements were to store and manage 15×10^9 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 is why data was distributed to a cluster of 64-bit multicore x86 Linux servers, each storing a portion of RDF data. The architecture of the cluster is a "Shared Nothing" type. Cluster nodes are divided into processing and storage nodes. Data segments stored on different nodes are determined by a simple formula that calculates the RID of the subject modulo number of segments. The benefit of such 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. 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. The Primary source of optimization is 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.

Virtuoso Virtuoso [7, 8, 15] is a multi-model database management system based on relational database technology. The approach of Virtuoso is to treat a triple store as a table composed of four columns. The main concept of the approach to the 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 considered by Virtuoso designers include extending SQL types with RDF data types, dealing with unpredictable object sizes, providing efficient indexing, extending relational statistics to cope with an RDF store based on a single table, and ensuring 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 the semantics of $owl : same - as$ by transitive closure.

Hexastore The Hexastore [21] 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 a distributed environment and the complete implementation of query processors including query optimization, persistent indexes, and other issues inherent in database technology.

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

5 Conclusion and Future Work

The rough design of the big3store system and precise implementation of query execution module (QEM) were presented. A semantic distribution method of RDF data and a distributed query execution method for RDF storage managers were presented. The storage manager big3store converts SPARQL queries into query tree structures using RDF algebra formalism. Nodes of those tree structures are represented by independent query node processes that execute the query autonomously and in a highly parallel manner while sending asynchronous messages to each other. The semantic data distribution method decreases the number of inter-server messages during query executions by using the semantic properties of RDF data sets.

This research is currently at the preliminary stage, and so far only the query execution module has been implemented and tested. While we are currently focusing on the effective execution of SPARQL queries in distributed computational environments, our future work will include the implementation of the minimum big3store system, benchmarks using large-scale triple data, and the confirmation of the efficiency of the proposed methods.

6 Acknowledgments

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. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

4. C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. {DBpedia} - a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154 – 165, 2009. The Web of Data.
5. 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.
6. Dublin core metadata inatiative. http://dublincore.org/, 2013.
7. O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In *CSSW*, pages 59–68, 2007.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194(0):28 – 61, 2013. Artificial Intelligence, Wikipedia and Semi-Structured Resources.
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. 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.
19. I. Savnik and K. Nitta. Algebra of rdf graphs. Technical Report (In preparation), FAMNIT, University of Primorska, 2013.
20. 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.
21. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, Aug. 2008.
22. T. White. *Hadoop: The Definitive Guide*. OâĂŹReilly Media, Inc., 2009.
23. Xml schema. http://www.w3.org/XML/Schema, 2012.
24. 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.