# Algebra of RDF Graphs for Querying Large-Scale Distributed Triple-Store

Iztok Savnik[1], Kiyoshi Nitta[2]

[1] University of Primorska & Jožef Stefan Institute, Slovenia
[2] Yahoo JAPAN Research, Tokyo, Japan
iztok.savnik@upr.si, knitta@yahoo-corp.jp

**Abstract.** Large-scale RDF graph databases stored in shared-nothing clusters require query processing engine that can effectively exploit highly parallel computation environment. We propose algebra of RDF graphs and its physical counterpart, physical algebra of RDF graphs, designed to implement queries as distributed dataflow programs that run on cluster of servers. Operations of algebra reflect the characteristic features of RDF graph data model while they are tied to the technology provided by relational query execution systems. Algebra of RDF graphs allows for the expression of pipelined and partitioned parallelism. Preliminary experimental results show that proposed algebra and architecture of query execution system scale well with large clusters of data servers.

## 1 Introduction

Recent development of graph-based semantic web shows the enormous interest of society to construct a detailed knowledge base (graph) including properties of categories from all popular areas of human activities. Knowledge bases such as Knowledge Graph, Wikidata, YAGO and Knowledge Vault currently include from 1000 up to 350.000 categories, up to 570 Mega instances of categories, up to 35.000 relationship types, and, up to 18 Giga relation instances [8]. However, from many aspects existent knowledge graphs are in their infant stage—more systematic use of intelligent tools for extracting the knowledge from various data sources has just begun.

The need for triple-store systems capable to store and manage from Tera ($10^{12}$) towards Peta triples is obvious. The scalability of storage system and query processing system to this amount of data is currently possible by using large-scale distribution of data into shared-nothing clusters. Query execution system in such environment must be able to employ various types of parallelism to allow simultaneous execution of huge amount of queries and provide reasonable response time.

Distributed triple-store big3store is based on dataflow architecture of query processing. Each query is a tree of algebra operations that is dynamically mapped to the tree composed of processes interconnected by streams of graphs, i.e., sets of triples. The scheduler that maps query trees to set of processes balances the computation load among the servers of cluster.

Triple-store of big3store is distributed into columns that store replicas of partitions into rows—cluster data servers. Data distribution is achieved by means of semantic distribution function [14] that splits the triples on the basis of the relation of each particular triple to the taxonomy of RDF classes and properties.

Algebra of RDF graphs is an abstract model used for the implementation of query execution system. Algebra is defined using set semantics—inputs and outputs of operations are sets of graphs. We present the denotational semantics of algebra and its implementation in the form of physical algebra that is further mapped to sets of processes implementing algebra operations. The architecture of query execution system based on algebra allows for the use of pipelined and partitioned parallelism [7].

Programming environment of parallel programming language Erlang [2] is used for the implementation of big3store. Erlang, together with database management system Mnesia that is tightly integrated with Erlang, may represent alternative data processing system for big data to Hadoop [17]. Indeed, it provides simple and robust parallel programming environment allowing processes to be effectively used in cluster of servers, it incorporates mechanisms that allow for the implementation of reasonable level of fault-tolerance, and, it integrates low-level database system appropriate for telecommunication applications that includes key-value indexes comparable to those of Hadoop storage system.

The contributions of this paper are the following. The architecture of distributed query execution system for processing large-scale RDF graphs based on algebra of RDF graphs is proposed. Query processor uses left-deep query trees to implement pipelined parallelism of algebra operations. Furthermore, it employs semantic triple distribution function [14] to achieve highly flexible partitioned parallelism. Access methods for tripe-patterns that address large partitions of triple-base are distributed to larger number of data servers, while the queries that address small partitions are executed on a single server. Query processor uses affinity scheduling, i.e., two level scheduling that persists to allocate the same data servers for execution of algebra operations for particular user. Finally, it uses key-value indexes in a similar manner to Hadoop to access data triples and to implement index-based nested-loop join operations.

The rest of the paper is organized as follows. The following Section 2 presents algebras of RDF graphs closely related to big3store algebra. Section 3 gives formal definition of algebra of RDF graphs and describes its physical counterpart, physical algebra of RDF graphs. The architecture of big3store query execution system together with detailed description of algebra implementation, is presented in Section 4. Preliminary experimental results are described in Section 5. Finally, Section 6 gives some conclusions and presents further work.

## 2 Related work

Algebra of RDF graphs implemented in distributed triple-store big3store is based on relational algebra and technology of relational database management systems [10, 11]. Database algebras are by nature functional languages where inputs and outputs of algebra operations can be treated as input and output flows of database objects. Operations of database algebra can be combined to form graph structure where operations (nodes) are interconnected by flows of objects [7].

The design of big3store algebra of graphs follows the leading ideas of relational algebra [5] while we identified and incorporated in it the salient features of triple-store data model. Firstly, instead of access methods scanning relational tables we use triple-pattern based access method to triple-store that can use all possible indexes on SPO attributes. Secondly, the results of algebra operations are not relations—sets of tuples—but sets of graphs. Consequently, operations $select$, $project$ and $join$ are adapted for graphs. Selection is based on expressions defined by means of graph nodes. Similarly, operation $project$ eliminates graph edges. Finally, operation $join$ is defined on graphs by introducing graph matching as join predicate.

Similar approach to definition of algebra for querying triple-stores are proposed by Angles and Gutierrez in [1]. Their formalization of SPARQL operations is based on mappings that follow semantics of *triples*. While their definition is tuned for studying expressive power of the language, our work is focused more on the implementation of algebra of graphs in shared-nothing cluster. They have shown in the paper that SPARQL have equivalent expressive power to non-recursive Datalog with negation that has, in turn, equivalent expressive to classical relational algebra.

Schmidt et.al. have proposed SPARQL algebra [15] to be used as foundations for SPARQL query optimization. They have defined set-based semantics for SPARQL by introducing SPARQL set algebra including similar operations to our algebra of graphs. They have identified fragments of SPARQL together with their complexity classes. For instance, they have shown that OPTIONAL-free fragments of SPARQL are either NP-complete or in PTIME. Furthermore, they have introduced algebraic equivalence rules that can be used for SPARQL query optimization, and, extensions of classical chase algorithm for optimization of AND queries.

Cyganiak proposed in [6] the use of relational algebra for SPARQL query processing. He presents the transformation from SPARQL into abstract relational algebra and shows differences between semantics of SPARQL and relational model. This approach allows for direct use of relational query optimization and query evaluation techniques for processing SPARQL queries. The transformation from relational algebra to SQL is defined. In comparison to Cyganiak's proposal, our approach focuses on distributed implementation of algebra of graphs while, in the similar manner, we use knowledge and technology gathered in area of relational systems for the implementation of triple-store database system.

## 3 Algebra of RDF graphs

Algebra of RDF graphs is a functional language defined on sets of RDF graphs. Inputs and outputs of algebra operations are sets of RDF graphs that are linked to other operations forming in this way a tree. As we will show later, algebra expressions i.e. trees of algebra operations are converted to trees of Erlang processes that can be located on different data servers.

Let us first define the basic terminology used in presentation. Let $I$ be the set of URI-s, $B$ the set of blanks and $L$ be the set of literals. Let us also define sets $S = I \cup B$, $P = I$, and $O = I \cup B \cup L$.

*RDF triple* is a triple $(s, p, o) \in S \times P \times O$. *RDF graph* $g \subseteq S \times P \times O$ is a set of triples. Set of all graphs will be denoted as $G$. We suppose the existence of a set of variables $V$ and the set of *terms* $T = O \cup V$. Term $t \in T$ is ground if $t \in O$.

We say that RDF graph $g_1$ is *sub-graph* of $g_2$, denoted $g_1 \sqsubseteq g_2$, if all triples in $g_1$ are also triples from $g_2$.

## 3.1 Ground graphs and graph patterns

*Triple pattern* $(s, p, o) \in (S \cup V) \times (P \cup V) \times (O \cup V)$ is a triple that can include variables as components. *Graph pattern* $gp \subseteq (S \cup V) \times (P \cup V) \times (O \cup V)$ is a set of triple patterns, i.e., graph defined as set of triples that can include variables as components. The set of all graph patterns is in the sequel denoted as $G_P$.

We will separate between ground and abstract entities. Ground triples are triples that include ground terms. Abstract triples, that can include variables, are triple patterns. Similarly, ground graphs are graphs that include triples composed of ground values, and, graph patterns represent abstract graphs that stand for a set of graphs from a given triple-store.

To be able to determine set of variables included in graph pattern $gp$ we define function $vars : G_P \rightarrow \mathcal{P}(V)$.

**Matching of graphs.** Let us now define relationship "match", denoted as $\sim$, between graphs including graph patterns. Graphs $g_1$ and $g_2$ match, denoted $g_1 \sim g_2$, iff the following conditions hold.

1. Two terms $t_1, t_2 \in T$ match, written $t_1 \sim t_2$, if either $t_1$ and $t_2$ are ground and $t_1 = t_2$, or, one of values is variable and the other is ground value.
2. Matching between two triples $r_1$ and $r_2$ exists, written $r_1 \sim r_2$, if all components of $r_1$ and $r_2$ match.
3. Graph $g_1$ matches graph $g_2$, written $g_1 \sim g_2$, when there exists bijection *alpha* : $g_1 \rightarrow g_2$ so that each triple $t_1 \in g_1$ matches $alpha(t_1) = t_2 \in g_2$.

Let $gp$ be graph pattern. Function $val : V \times G_P \times G \rightarrow O$ maps variables $v \in vars(gp)$, graph patterns $gp \in G_p$ and ground graphs $g \in G$ that match $gp$ to values $o \in O$. Let $t_1 \in gp$ be triple that includes variable $v$, then $val(v, gp, g) = o$ is component of triple $alpha(t_1) = t_2 \in g$ that corresponds to $v$ in $gp$.

**Interpretation of graph pattern.** Interpretation of graph pattern $gp$ in database of triples storing graph $db$ is a set of all sub-graphs $g$ of $db$ that match $gp$.

$$[\![gp]\!]_{db} = \{g \mid g \sqsubseteq db \wedge g \sim gp\}$$

Special case of graph pattern is triple pattern $tp$ where complete graph is one single triple that can include variables possibly in all three positions. Interpretation of $tp$ is a set of all triples from $db$ that match $tp$.

Triple patterns represent graph counterpart of relational access methods [4]. They are always the leafs of query tree. Implementation of query node for a given triple pattern can use SPO indexes to access ground triples.

## 3.2 Definition of algebra

Let us now present algebra of RDF graphs. We denote graph query as $Q$, triple pattern as $TP$, selection condition as $C$, condition operations as $OP$, sets of variables as $SV$, and, variables as $V$. Syntax of algebra is defined as follows.

$$\begin{array}{ll}
Q & ::= TP \mid \mathit{select}(Q,C) \mid \mathit{project}(Q,SV) \mid \mathit{join}(Q,Q) \mid \mathit{union}(Q,Q) \mid \\
 & \quad \mathit{intsc}(Q,Q) \mid \mathit{diff}(Q,Q) \mid \mathit{leftjoin}(Q,Q) \\
TP & ::= (S \mid V, P \mid V, O \mid V) \\
C & ::= V\ OP\ V \mid V\ OP\ O \mid C \wedge C \mid C \vee C \mid \neg\, C \\
OP & ::= = \mid \neq \mid > \mid \geq \mid < \mid \leq \\
SV & ::= \{V+\} \\
S & ::= \text{URI} \mid \text{Blank-Node} \\
P & ::= \text{URI} \\
O & ::= \text{URI} \mid \text{Blank-Node} \mid \text{Literal} \\
V & ::= ?a\ ..\ ?z
\end{array}$$

We extend previously defined function $vars$ to queries. Let $(Q)$ be the set of all queries. The function $vars : \mathcal{Q} \rightarrow \mathcal{P}(V)$ maps each query to the set of variables that are included in the query. Let us now present the denotational semantics of RDF algebra by defining the interpretation of each particular operation.

Access paths to database of triples storing graph $db$ are defined using *triple patterns* $(t_1, t_2, t_3)$ where $t_1 \in (S \cup V)$, $t_2 \in (P \cup V)$ and $t_3 \in (O \cup V)$.

$$[\![(t_1, t_2, t_3)]\!]_{db} = \{\ (s, p, o) \mid (s, p, o) \sqsubseteq db \wedge (s, p, o) \sim (t_1, t_2, t_3)\ \}$$

SPARQL operation FILTER is represented by means of operation $\mathit{select}(q, C)$ where $q$ is query and $C$ is condition expression.

$$[\![\mathit{select}(q, C)]\!]_{db} = \{\ g \mid g \in [\![q]\!]_{db} \wedge C(g) = \mathit{true}\ \}$$

The evaluation of condition $C$ on graph $g$ is defined by the following rules. Value of $C(g)$ is presented by cases of $C$ structure.

- $C = ?a\ OP\ o$, where $?a \in V$ and $o \in O$: if $\mathit{val}(?a, q, g)\ OP\ o = \mathit{true}$ then $C(g) = true$, else $false$.
- $C = ?a\ OP\ ?b$, where $?a, ?b \in V$: if $\mathit{val}(?a, q, g)\ OP\ \mathit{val}(?b, q, g) = \mathit{true}$ then $C(g) = true$, else $false$.
- $C = C_1 \wedge C_2$: if $C_1(g) = \mathit{true}$ and $C_2(g) = \mathit{true}$ then $C(g) = true$, else $false$.
- $C = C_1 \vee C_2$: if $C_1(g) = \mathit{true}$ or $C_2(g) = \mathit{true}$ then $C(g) = true$, else $false$.
- $C = \neg C_1$: if $C_1(g) = false$ then $C(g) = true$, else $false$.

Operation $\mathit{project}(q, s)$ projects graphs $g \in [\![q]\!]_{db}$ to graphs composed of triples that include values of variables from set $s$. Let $\mathit{tr\text{-}vars} : db \times G_P \rightarrow \mathcal{P}(V)$ denote function that maps triples $t \in [\![q]\!]_{db}$ and query $q$ to set of variables $vs \in \mathcal{P}(V)$ such that for each $var \in vs$ value of variable $var$ is a component of $t$.

$$[\![project(q,s)]\!]_{db} = \{\ g_1 \mid g \in [\![q]\!]_{db} \wedge \forall t \in g(\mathit{tr\text{-}vars}(t,q) \subseteq s \Longrightarrow t \in g_1)\ \}$$

Operation $join(q_1, q_2)$ joins two sets of graphs that are interpretations of queries $q_1$ and $q_2$. Let $vs$ be a set of variables $vars(q_1) \cap vars(q_2)$. The result of $join$ includes union of graphs $g_1 \in [\![q_1]\!]_{db}$ and $g_2 \in [\![q_2]\!]_{db}$ such that they agree in the values of all common variables from $vs$. Observe also that joining two graphs is obtained by making union of graph triples from both graphs. Semantics of operation $join$ can be defined as follows.

$$\begin{aligned}[\![join(q_1,q_2)]\!]_{db} = \{\ g_1 \cup g_2 \mid g_1 \in [\![q_1]\!]_{db} \wedge g_2 \in [\![q_2]\!]_{db} \wedge \\ \forall v \in vs : val(v,q_1,g_1) = val(v,q_2,g_2)\ \}\end{aligned}$$

Set operations are defined in a usual way except that argument sets can include graphs that have heterogeneous structure. Union, intersection and difference of $q_1$ and $q_2$ is defined as union, intersection and difference of their interpretations $[\![q_1]\!]_{db}$ and $[\![q_2]\!]_{db}$. Set operations of RDF algebra are defined as follows.

$$\begin{aligned}[\![union(q_1,q_2)]\!]_{db} &= \{\ g \mid g \in [\![q_1]\!]_{db} \vee g \in [\![q_2]\!]_{db})\ \} \\ [\![intsc(q_1,q_2)]\!]_{db} &= \{\ g \mid g \in [\![q_1]\!]_{db} \wedge [\![q_2]\!]_{db})\ \} \\ [\![diff(q_1,q_2)]\!]_{db} &= \{\ g \mid g \in [\![q_1]\!]_{db} \wedge g \not\in [\![q_2]\!]_{db}\ \}\end{aligned}$$

Finally, to implement SPARQL operation OPTION we define operation $leftjoin(q_1, q_2)$, that is, left outer join of two sets of graphs which are elements of the interpretations of queries $q_1$ and $q_2$. For each pair $g_1 \in [\![q_1]\!]_{db}$ and $g_2 \in [\![q_2]\!]_{db}$ the result of $leftjoin(q_1, q_2)$ includes either $g_1 \cup g_2$ in the case that $g_1$ can be joined with $g_2$, or $g_1$ if $g_1$ can not be joined with $g_2$. Let $vs$ be a set of variables $vars(q_1) \cap vars(q_2)$. Operation $leftjoin(q_1, q_2)$ can be defined as follows.

$$\begin{aligned}[\![leftjoin(q_1,q_2)]\!]_{db} = \{\ g \mid g_1 \in [\![q_1]\!]_{db} \wedge g_2 \in [\![q_2]\!]_{db} \wedge \\ ((\mathit{is\text{-}join}(g_1,g_2) \wedge g = g_1 \cup g_2) \vee (\neg \mathit{is\text{-}join}(g_1,g_2) \wedge g = g_1))\ \},\end{aligned}$$

where $\mathit{is\text{-}join}(g_1, g_2)$ is defined as

$$\mathit{is\text{-}join}(g_1,g_2) = \forall v \in vs : val(v,q_1,g_1) = val(v,q_2,g_2)$$

### 3.3 Physical algebra of RDF graphs

The design of physical algebra of RDF graphs follows the ideas used for implementation of relational algebra in the frame of relational database management systems [10, 11]. Previously presented operations of RDF algebra are converted into three physical operations: physical access method (AM) `AM`, physical join denoted `join`, and, physical set operations `union`, `diff` and `intsc`.

All physical operations now include besides the functionality of their logical counterparts also the functionality of operations *select* and *project* operations. Each physical operation therefore includes also *select list* and *project list*. There are more reasons for folding more logical operations into single physical operation.

Firstly, it makes sense to perform selection of triples immediately after data needed for selection is available. For instance, immediately after obtaining triples by means of a given triple-pattern access method, they are filtered using selection conditions.

For similar reason operation *project* is performed as soon as possible. Immediately after some triple in a result RDF graph is not useful, it is dropped. For instance, after using particular triple for performing *join* operation, it can be omitted from result graph, if of course it is not needed as the result of query, or, for some other operation higher in the query.

The above two rules resemble "pushing" selections and projections down towards the leafs of query tree in relational database systems.

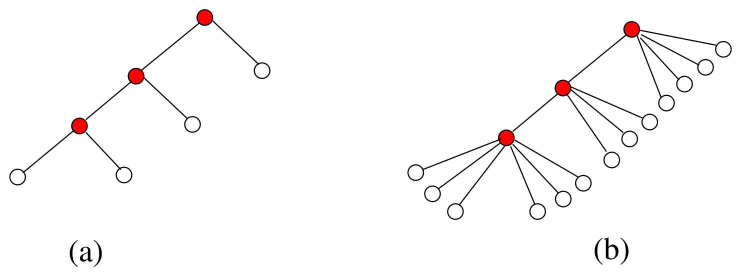

**Fig. 1.** (a) Left-deep query tree (b) Left-deep query tree with multiple `AM` operations

Secondly, the reasons for folding selections and projection into `AM` and `join` are: 1) the possibility to use join reordering algorithm for query optimization, and, 2) the possibility to implement *left-deep* as well as *bushy query trees* [10]—both of them have operations `AM` as leafs and operations `join` as the inner nodes of query trees.

big3store is currently using left-deep query trees. An example of left-deep query tree with 3 `join` operations and 4 `AM` operations is given in Figure 3.3(a). The most important advantage of using left-deep trees is the *pipeline* that is formed by physical `join` operations. The results of retrieving graphs from outer query node of `join` operation is used for index-based access to the inner query node. The graph that is constructed as the result of `join` operation is then sent to the parent query node i.e. `join` query node. Consequently, there is no need to store intermediate results during query evaluation.

Triples related to some class with very large number of instances are, by using of semantic distribution algorithm, distributed to more data servers. Therefore, physical operation `AM`, defined using some triple-pattern, may be executed on number of data servers. Indeed, it is desirable that triple-pattern based operations `AM`, that tackle large number of triples, are distributed to more servers. The number of servers depends on the size of targeted set of triples. Left-deep query trees can therefore have multiple `AM` query nodes as presented in Figure 3.3(b).

## 4 Distributed query execution system

Storing and querying huge volumes of data efficiently is currently possible by using shared-nothing cluster architecture [16]. Efficient data servers with huge amount of RAM and disk storage are available as inexpensive commodity hardware. This allows heavy distribution and replication of data as well as massive distribution of query processing on servers forming very large clusters.

Big3store is a *data-flow system* [3] where triple-store is composed of an array of data servers arranged into columns and rows. The complete triple-store is partitioned and distributed into *columns* based on semantic information attached to triples via triple-store schema. Each column stores a partition of triple store that is replicated to the column *rows*. Rows of the column therefore contain replicas of triple-store partitions assigned to columns.

While triple store partitioning affects significantly the performance of query executions, it is not the focus of this paper. Detailed presentation of big3store partitioning algorithm is given in [14]. Let us here present only some important ideas that have guided the design of triple-store (graph) partitioning.

Hash-based partitioning can not be employed for storing huge triple datasets that are expected to grow significantly in the following decade. Splitting data into a large number of partitions based on hashing can increase significantly the communication traffic among the data servers, especially, when large number of transactions is executed in parallel.

Big3store uses *semantic distribution algorithms* to partition triple-store into chunks that are suitable for distribution and that are related to a set of schema entities which serve as the key for distribution. Since distribution is based on rich taxonomy of classes spanning more then ten hierarchical levels we can achieve *well-defined distribution* in the sense that triples defined for classes including large number of instances are split into larger number of chunks. Triples defined for a class that has small number of instances is stored in one chunk. Query distribution must follow data distribution: larger the class of triples addressed by query, larger the number of columns where query will be executed.

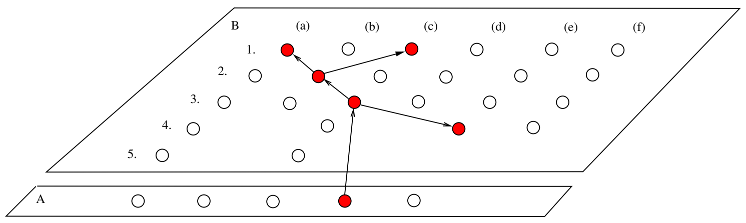

**Fig. 2.** Configuration of servers for a particular query

Figure 4 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 into *columns* labeled from (a) to (f). A complete database is distributed to columns such that each column stores a portion of the complete database.

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, larger 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.

## 4.1 Architecture of query execution system

Erlang programming environment [2] is used for the implementation of big3store as an alternative to Hadoop-like systems [17]. It provides remarkably simple and effective parallel programming model based on lightweight processes. Erlang processes use "shared nothing" philosophy where the communication among processes is realized solely by means of synchronous and asynchronous messages.

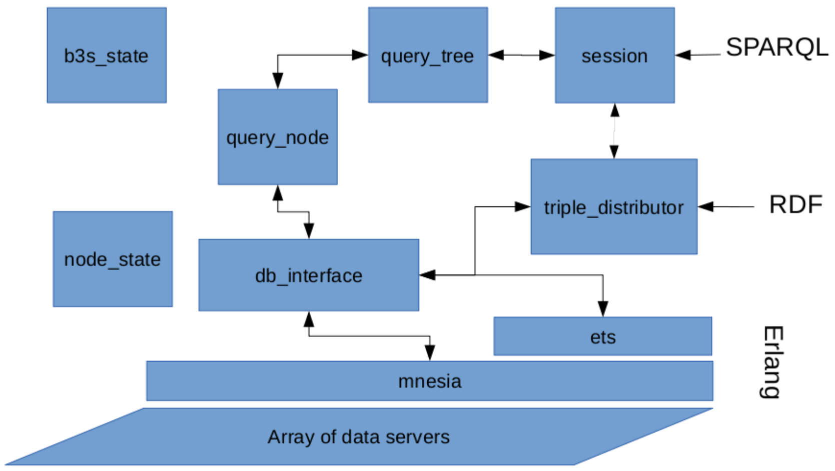

**Fig. 3.** Architecture of big3store query executor

Query execution system of big3store is composed of modules presented in Figure 4.1. Each module includes the implementation of particular type of process.

State modules `b3s_state` and `node_state` are used for efficient sharing of big3store configuration data structures as well as for storing and querying current state of system, such as for instance number of processes running at each particular data server.

Each data server runs one instance of Erlang Mnesia database system that serves as *local triple-store*. Triple-store is realized by means of a single table `triple_store` that is accompanied with 6 indexes for all combination of SPO attributes. Mnesia provides transaction-based access to local triple-store through module `db_interface`. However, since `db_interface` provides only very simple cursor based access to a single table, local triple-store can be easily replaced by other database engine, and, even file-based access to RDF triples.

Module `triple_distributor` implements various schema-based algorithms for the distribution of triple-store into a set of cluster columns [14].

Session processes are implemented in module `session`. They serve as user-interface for interaction with users, initiate creation of query-tree processes, control the execution of query tree, and collect the results of query execution. One session can spawn many query trees in parallel.

Module `query-tree` implements query tree processes that run on front-servers. The main task of query-tree process is to prepare, schedule and initiate the execution of query in the form of query tree composed of query-node processes interconnected by means of streams. Therefore, each query-tree process controls one or more query node processes that constitute query. This is presented in more detail in Sub-section 4.2.

Physical algebra operations `AM`, `join`, `union`, `intsc` and `diff` are implemented in `query-node` modules. Each physical algebra operation is realized as independent Erlang query-node process that runs on one of data servers. All operations are implemented as state machines executing particular protocol: access method to local triple-store, indexed nested-loop join algorithm, or, particular set operation. Sub-sections 4.4 and 4.5 give more detailed description of operations `AM` and `join`.

## 4.2 Query-tree process

Query-tree module implements processes that serve as front-end of query tree represented as tree of inter-connected processes running on array of servers. Query is received from session process in the form of a list of triple-patterns augmented with projections and selections as presented in previous section.

Query of type `qt_query()` presented to query tree process as parameter of message `start` is converted into tree data structure stored as process dictionary entry. First element of list representing `qt_query()` is triple-pattern of the lower leftmost query node. Last element of list is triple-pattern of the upper rightmost query node. All other triple-patterns are placed as inner query nodes in order between lower leftmost and upper rightmost.

Query-tree process analyzes the query, computes all components of query node processes to be started, determines cluster columns associated to each query node, and, schedules the rows of columns to be employed for running each particular query node of query tree.

Query-tree process determines the location of each query node in terms of column and row in array (cluster) of servers. Each query node is executed on location determined by query tree process. Firstly, the column of query node is computed by using *distribution function* that translates triple-patterns to columns in array of servers. Secondly, rows in given columns are scheduled dynamically based on current load of servers in columns.

We use two types of scheduling of column rows to query nodes. First type of scheduling is random assignment of rows to query nodes. The second method used for scheduling is bookkeeping the execution of each particular query node on particular server. Bookkeeping is realized by means of local `node_state` process. Besides bookkeeping `node_state` provides a function that selects row server with least load.

Both types of scheduling resemble *affinity scheduling* where we tend to select the same servers for the same session. The benefits of assigning the same servers (rows) in columns for same session is primarily in utilizing cache of local database management system Mnesia. Experiments are currently under way to present the benefits of affinity scheduling in terms of execution speed.

## 4.3 Triple-pattern query node

Triple-pattern (abbr. TP) query node is implemented as Erlang `gen-process`. It realizes access method at local triple-store implemented as Mnesia table. Access method is defined by means of triple-pattern, and, it can use index based access to triple-store.

TP query node is implemented as state machine. Input and output messages trigger coroutines that comprise protocol. The states of TP query node are: `active, db_access, eos`, and `inactive`. Message `start` initializes TP query node process and moves state to `active`. Message `eval` starts with evaluation and moves state to `read_db`.

After obtaining triples from local Mnesia database, TP query node process checks them against *selection list*. The selected triples are sent to parent of query node by using `data` messages. Protocol requires that each `data` message is sent to parent process only after receiving `empty` message from parent process. Therefore, protocol can control the number of `data/empty` messages that comprise stream. Subsequent messages `empty` retain state `read_db`.

After `end_of_stream` is obtained from function accessing triple-store, state moves to `eos`. Message `eval` can be received multiple times if in state `active` or `eos`. Finally, `stop` message puts TP query node to state `inactive`.

## 4.4 Join query node process

Join query node is implemented as independent Erlang `gen-process`. Join query node is a state-machine realizing protocol that has incoming and outcoming messages. Each message is implemented as coroutine.

Join query node state-machine has the following states: `active, wait_next_outer, wait_next_inner, eos` and `inactive`. Message `start` initializes the main data structures of join query node and sets state of protocol to `active`. Message `eval` start the evaluation of join query node by sending message `eval` to all children, and, moves state to `wait_next_outer`. After this, state alternates between `wait_next_outer` and `wait_next_inner`. State moves to `eos` after end of outer streams is detected.

Join query node implements join method which is a variant of indexed nested-loop join algorithm. However, it can have multiple outer query nodes as well as multiple inner query nodes. Since we suppose that every local triple-store indexes triple table on all possible subsets of SPO, all join variables are supported by indexes.

Algorithm of join method is defined as follows. Each graph obtained from outer query nodes causes initialization of inner query nodes by means of message `eval`. Initialization of inner query nodes uses the values of join variables obtained from outer graph. Only those graphs are retrieved from inner query nodes that match previously obtained outer graph. Each outer and inner graphs are merged into one graph which is tested against *selection list* and projected using *project list* of given query node. If selected then resulting graph is sent to parent query node.

### 4.5 Fault tolerance

Erlang programming environment provides the tools for the construction of fail-safe process hierarchies by means of Erlang supervision processes [2]. Important process state data structures are circulating among supervision and supervised processes. The mechanism is integrated into message sending/receiving protocol. Each message received by process A includes the current state of A that was stored by its supervision process. After completing the task, process A returns its new state as a function result. In this way, a supervision process always has up-to-date state of all processes that it manages. In the case that process failure is detected by supervision process it can be restarted using the last state. Furthermore, supervised processes can form various types of structures with specific behavior.

## 5 Experimental results

As a preliminary study, the execution time of benchmark queries in big3store are compared with the execution time obtained with Virtuoso [9].

Benchmark environment comprises six server machines. All of them have the same physical specifications. Each server has two 2.9 GHz Xeon E5-2960 CPU and 256 GB of RAM. One Erlang interpreter process was invoked on each server. Benchmark configuration uses one server as *front server* and the other five servers as *data servers*.

Virtuoso was installed on one of the servers that were used to execute big3store.

Let us first describe benchmark queries presented in Figure 5. The first group of queries are simple queries that produce small number of intermediate and final results. Query Q1 finds all triples having property `<startedOnDate>`. It returns 9 triples from *YAGO2s*. Query Q2 finds all sets (graphs) of triples sharing the same subject that has `<startedOnDate>` and `<endedOnDate>` properties in the graph. It returns 1 triple. Query Q3 finds graphs that describe Japanese computer scientists that have created a programming language. Query Q4 returns the creation dates of all things classified as wordnet_language that were created by Ericsson.

Query Q5 compares `<Slovenia>` and `<Japan>` by using the same predicate in triple patterns. While it is similar to query Q2, query Q3 returns 241,596 graphs. Query causes large number of intermediate results that are transferred as messages among data servers.

```
SELECT * WHERE {
  ?sbj <startedOnDate> ?obj.
}

Query Q1

SELECT * WHERE {
  ?sbj <startedOnDate> ?obj1.
  ?sbj <endedOnDate>   ?obj2.
}

Query Q2

SELECT * WHERE {
  <Slovenia> ?prd ?obj1.
  <Japan>    ?prd ?obj2.
}

Query Q5

SELECT * WHERE {
  ?a1 <actedIn>     ?movie.
  ?a2 <actedIn>     ?movie.
  ?a1 <livesIn>     ?c1.
  ?c1 <isLocatedIn> <England>.
  ?a2 <livesIn>     ?c2.
  ?c2 <isLocatedIn> <England>.
}

Query Q6

SELECT * WHERE {
  ?p1 <isMarriedTo> ?p2.
  ?p1 <wasBornIn> ?city.
  ?p2 <wasBornIn> ?city.
}

Query Q8
```

```
SELECT * WHERE {
   ?p rdf:type
   <wikicategory_Japanese_computer_scientists> .
   ?p <created> ?o .
   ?o rdf:type <wordnet_programming_language> .
}

Query Q3

SELECT * WHERE {
   <Ericsson> <created> ?pl.
   ?pl rdf:type <wordnet_language>.
   ?pl <wasCreatedOnDate> ?dt.
}

Query Q4

SELECT * WHERE {
   ?p <hasGivenName>  ?gn.
   ?p <hasFamilyName> ?gn.
   ?p rdf:type <wordnet_scientist>.
   ?p <wasBornIn>     ?c1.
   ?c1 <isLocatedIn> <Switzerland>.
   ?p <hasAcademicAdvisor> ?a.
   ?a <wasBornIn>          ?c2.
   ?c2 <isLocatedIn> <Germany>.
}

Query Q7

SELECT * WHERE {
   <Tim_Burton> <directed> ?movie1.
   <Johnny_Depp> <actedIn> ?movie1.
   ?p1 <directed> ?movie1.
   ?p2 <influences> ?p1.
   ?p3 <actedIn> ?movie1.
   ?p3 <actedIn> ?movie1.
   ?p4 ?prd1 ?p3.
   ?p4 <actedIn> ?movie2.
   ?p1 ?prd1 ?p4.
}

Query 9
```

**Fig. 4.** Benchmark queries

**Table 1.** Benchmark Results (in seconds)

| Query | **big3store** | **Virtuoso** |
|---:|---:|---:|
| Q1 | 0.015 | 0.149 |
| Q2 | 0.086 | 0.133 |
| Q3 | 0.033 | 0.159 |
| Q4 | 0.009 | 0.608 |
| Q5 | 95.594 | 0.054 |
| Q6 | 3.652 | 0.262 |
| Q7 | 7.549 | 0.279 |
| Q8 | 23.512 | 0.182 |
| Q9 | 104.364 | 0.558 |

Queries Q6, Q7, and Q8 correspond to YAGO queries B1, A1, and B2 from [13], respectively. Because YAGO and YAGO2s [12] have different schema structures, queries were rewritten to have similar meaning.

Query Q6 returns pairs of actors that were playing in the same film and live in the same city in England. Query Q7 returns graphs describing scientists that were born in a city in Switzerland, and have academic advisor who was born in a city in Germany. Query Q8 returns all married couples that were born in the same city.

Query Q9 was constructed to test circular queries. While current version of query Q9 is specific and executes fast, a circular query can be constructed by removing the first two triple patterns of `<Tim_Burton>` and `<Johnny_Depp>`.

Let us now give some comments on comparison presented in Table 1. System big3store executed queries Q1, Q2, Q3 and Q4 faster than Virtuoso. One reason for this is that Mnesia copies complete database in main memory, if it is possible.

It is also apparent that queries that do not produce a lot of traffic execute in big3store much faster that queries that produce a lot of traffic among the servers. There are more reasons for this. Firstly, we currently do not use any data compression, so data is stored in raw form. Secondly, streams are implemented by sending one message for one graph.

The improved version of big3store will map IRIs to integers to optimize storage and transfer speed. Furthermore, the speed of stream transfer will be improved by packing more graphs into bundles that will serve as unit of transfer.

Another reason for slow performance of some queries is in the implementation of cursors in Mnesia. Index-based access to table always returns all results in one package. Consequently, there is almost no parallelism in the execution of queries. The improved version of big3store will replace Mnesia with BerkeleyDB.

## 6 Conclusions

Algebra of RDF graphs and its implementation on shared-nothing clusters is presented. Algebra is described by first defining denotational semantics of abstract algebra. Physical algebra corresponding to its abstract counterpart is based on technology of relational and parallel database systems. The architecture of distributed query processing system based on the presented algebra is described. Finally, some preliminary experimental results are discussed.

We have a list of tasks that remain to be completed. Among the most important are: distributed implementation of mapping from strings (URIs) to integers and its inverse mapping, more deep study of the effects of structure and distribution of query trees to the execution speed, experimental study that will give more insight into interrelations between data and query distribution, and, improving the communication speed among cluster servers by packing triples into bundles.

## 7 Acknowledgments

We thank Andreas Schmidt for valuable suggestions on draft version of this paper. 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. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.
3. S. Babu and H. Herodotou. Massively parallel databases and mapreduce systems. *Foundations and Trendsin Databases*, 5(1):1–104, 2012.
4. M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Storage management for objects in exodus. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Applications, and Databases*. Addison-Wesley Publishing Co., 1988.
5. E. F. Codd. A relational model of data for large shared data banks. *Communication of ACM*, 13(6):377–387, June 1970.
6. R. Cyganiak. A relational algebra for sparql, 2005.
7. D. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of ACM*, 36(6), 1992.
8. X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *KDD-2014*, KDD. ACM, 2014.
9. 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.
10. G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.
11. G. Graefe. Dynamic query evaluation plans: Some course corrections? *IEEE Data Eng. Bull.*, 23(2):3–6, 2000.
12. 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.
13. T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proc. VLDB Endow.*, 1(1):647–659, Aug. 2008.
14. I. Savnik and K. Nitta. Semantic partitioning method for very large rdf graphs. Technical Report (In preparation), FAMNIT, University of Primorska, 2014.
15. 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.
16. M. Stonebraker. The case for shared nothing. *Database Engineering Bulletin*, 9(1):4–9, 1986.
17. T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.