Open Access

Open Journal of Cloud Computing (OJCC)
Volume 1, Issue 2, 2014

www.ronpub.com/journals/ojcc
ISSN 2199-1987

A Self-Optimizing Cloud Computing System for Distributed Storage and Processing of Semantic Web Data

Sven Groppe, Johannes Blume, Dennis Heinrich, Stefan Werner

Institute of Information Systems, University of Lübeck, Ratzeburger Allee 160, D-23562 Lübeck, Germany
groppe@ifis.uni-luebeck.de, jblume@jblume.com, { heinrich, werner }@ifis.uni-luebeck.de

ABSTRACT

Clouds are dynamic networks of common, off-the-shell computers to build computation farms. The rapid growth of databases in the context of the semantic web requires efficient ways to store and process this data. Using cloud technology for storing and processing Semantic Web data is an obvious way to overcome difficulties in storing and processing the enormously large present and future datasets of the Semantic Web. This paper presents a new approach for storing Semantic Web data, such that operations for the evaluation of Semantic Web queries are more likely to be processed only on local data, instead of using costly distributed operations. An experimental evaluation demonstrates the performance improvements in comparison to a naive distribution of Semantic Web data.

TYPE OF PAPER AND KEYWORDS

Regular research paper: *Semantic Web, Cloud Data Management, RDF, SPARQL, Distributed Storage, Distributed Query Processing.*

1 INTRODUCTION

In the last two centuries, the Internet has grown at an exponential rate. The biggest part of this growth can be attributed to the World Wide Web, both to publicly visible web pages and internal networks.

This massive amount of data is quickly becoming harder and harder to handle, as most of it is not produced in a format which can be easily processed through electronic means. Billions of pages written in natural languages, untagged pictures and graphics of many kinds, together with other media have become an as-of-yet untapped source of knowledge.

To help processing this data, the concept of a Semantic Web has been developed. The projects involved have the intention to connect unstructured data with ways to express the semantics, the meaning, of this data, in order to be able to harness it. At the forefront of this movement is the World Wide Web Consortium, which has started to develop standards for the Semantic Web.

One of the problems with processing this data, even when its semantic structure is available, is its sheer amount. Many collections of semantic web data already contain billions of data fragments like those of the Linking Open Data (LOD) project [14]. The exponential

growth of computing power cannot match the rate at which these databases are growing, so methods need to be found to split the work. Distributed computing may be the solution to handle this data. Specialized, high powered machines to build such distributed computation networks are not state-of-the-art anymore since the leading Internet search engine provider [2] has successfully shown that it is often more effective to use dynamic networks of common, off-the-shelf computers to build computation farms. The technology to do this has been named *cloud computing*.

A computing cloud consists of many comparatively low powered nodes on which data and computation can be distributed dynamically. In times of higher performance demands more nodes can be quickly added to the system and exchanged when needed. In contrast to peer-to-peer networks, these clouds are controlled by a central instance which can optimally distribute load among the system.

Our goal is to explore new ways to cope with the challenges posed by processing semantic data, especially when evaluating SPARQL [26, 27] queries over RDF [25] data. RDF data has a unique graph structure which naturally leads itself to free distribution within a distributed computing cloud. Exploiting this structure to improve processing of queries was the main idea leading to our proposed approach.

Our main contributions are the following:

- **A cloud-based system to distributed SPARQL processing** with self-optimizing capabilities.
- **Definition of locality:** We define *locality* especially to have a mean to measure how many distributed join operations are potentially necessary in the RDF graph. As smaller the locality is as more join operations can be done locally on the slave nodes avoiding high network costs.
- **An iterative optimization strategy:** One run improves locality of the RDF graph, succeeding iterations improve further the locality until an optimum is reached.
- **An experimental analysis** showing the improvements of our proposed iterative optimization strategy to distributed query processing.

The rest of this paper is organized as follows: Section 2 provides the related work concerning SPARQL processing in peer-to-peer networks, clouds and in the context of data source integration, categorizes these existing approaches and put them into relation with our proposed cloud-based system; we describe our approach in Section 3 as well as the architecture and each component of our proposed system including the discussion of updates of administrated triples and cloud topology and a new optimization strategy to speed up distributed query processing; Section 4 contains the experimental evaluation; and finally Section 5 summarizes and concludes our work.

2 RELATED WORK

We present the related work in the context of peer-to-peer networks, clouds and distributed SPARQL processing in the context of data source integration in the following subsections. Afterwards, we categorize the existing contributions and relate them to our approach.

2.1 Distributed RDF Data and SPARQL Query Processing in Peer-to-Peer Networks

One of the first decentralized approaches to store RDF data in a peer-to-peer network can be found in [4]. The implementation makes extensive use of distributed hash tables and employs a recursive algorithm for multi-predicate queries. The network is built on the grid computing framework MAAN and the service look-up framework Chord.

In [1] the implementation of distributed RDF data takes a federated approach. As an extension to the Sesame system for storing and querying, the author implements a 'mediator', which allows queries to be transparently executed over several RDF repositories at once and merge the results. Implementing the mediator with a 'brute force' approach is compared to having a possibly multi-layered index over the predicates in each of the federated repositories.

The focus in [3] lies in the evaluation of different heuristics to improve query planning in peer-to-peer RDF repositories which are based on distributed hash tables. Storing and representing triples is not treated in this paper. Introduced are local heuristics, which do not generate traffic (e.g. variable counting), dynamic heuristics, which query only meta information (e.g. interpolation of triple counts), and heuristic wrappers, which act like caches to prevent duplicate retrieval. Combinations of these heuristics are evaluated in extensive tests.

Another peer-to-peer approach can be found in [12]. Instead of common networks based on *distributed hash tables* (DHT), this implementation uses swarm algorithms based on virtual ants to cluster related RDF tuples closer together, using the Sesame2 system as the storage layer. This is one of the few approaches which try to exploit data locality in an RDF graph.

2.2 Cloud-based SPARQL Query Processing

A shared-nothing architecture consisting of off-the-shelf machines for parallel processing of RDF data is shown in [6]. This report does not go into technical details, but gives a very high-level overview of all necessary components with focus on the application of the Map-Reduce [7] pattern. Touched subjects are a scripting language used to express the computations with a parser and compiler, constructing logical execution plans, compiling the plans into physical execution plans and scheduling them on the machines. No concrete implementation of these modules is given.

The master's thesis [13] shows a complete implementation of a Map-Reduce based RDF store with SPARQL queries. The RDF data is held in an Apache Cassandra key-value store. SPARQL queries to be processed are mapped to the Hadoop Pig language framework. Query execution is separated into a "select" and a "join" phase, which are implemented as separate, sequential Map-Reduce jobs. A focus of the thesis is an algorithm for Basic Graph Pattern matching utilizing Hadoop. Additionally, several optimizations are presented, e.g. indexes for RDF data filtering and preprocessing of joins.

In [5], a very simple implementation of a distributed RDF store is presented based on the Map-Reduce pattern implemented in the Hadoop framework. It employs its facilities for storing large tables (HTable) and implements simple partial RDF query processing in its nodes. As the HTable framework gives little control over the concrete storage place, locality of RDF data cannot be fully exploited, which leads to inefficient query processing.

A more complex implementation based on Hadoop can be found in [11]. It does not rely on HTable, but uses the distributed file system HDFS directly, introducing index-like files which reduce the total size of stored data and improves query execution times. This increases data locality, but still heavily restricts proper distribution of RDF data because of the opaque nature of HDFS. The paper also describes algorithms to split a query into multiple jobs and shows how to do effective joins with this setup.

The subject of [16] is processing of a subset of SPARQL queries in a Map-Reduce based RDF store. The focus lies on providing an efficient algorithm for multi-way joins instead of the commonly employed multiple individual joins. The paper shows how to process basic graph patterns in SPARQL queries and shows strategies for join-key selection.

The authors of [19] analyze pattern matching, grouping and aggregation in Map-Reduce based systems and develop extensions to Hadoop's "Pig" high level parallel processing language system to facilitate these operations. The paper also shows query optimization via operator-coalescing and look-ahead processing in RDF queries.

2.3 Distributed SPARQL Processing in the Context of Data Source Integration

An extensive analysis of a specific kind of index for distributed RDF stores is given in [21]. The authors propose an index based on paths in the RDF graph and develop algorithms to optimize queries by matching on these paths. The index is also used to provide heuristics for join ordering in multi-join queries.

An overview of different index structures for distributed RDF stores is presented in [22], including indices which represent structural similarities in the RDF graph and indices which map triples to the dimensions of spatial index structures. The paper also describes data structures for efficient local join processing and the handling of broken links.

Distributed SPARQL queries and their optimization are the subject of [23]. The paper focuses on ordering tuple retrieval for SPARQL queries by applying a minimum spanning tree algorithm to the query graph together with RDF predicate statistics. The technique is not applicable to queries where the predicate is a variable.

2.4 Categorization of Existing Contributions

All of these approaches can be categorized in one of the three following categories:

Imitating Relational Databases: Many approaches take the already extensive research in the area of relational databases and try to apply it to semantic data. The data is treated as a very simple instance of a relational database with a relation consisting of only three attributes: Subject, predicate and object. Then, the data is distributed with the well-known techniques of horizontal or vertical distribution. This approach is taken by [16], [19] and [23].

Specialized Indices: Some systems do not try to find an efficient way to store the semantic data, but instead try to generate efficient indices into already existing storage forms. These systems are often part of federated systems. Examples of this approach are [21] and [22].

Peer-to-Peer Systems: These systems utilize more the graph structure semantic data can be represented as. The peers then analyze parts of the graph and order the peer network accordingly. Systems doing this are [3], [4] and [12].

All these systems have in common that they do not optimize the data distribution such that local joins can

be processed as much as possible and distributed joins are avoided in order to save transmission costs.

2.5 Relation to our Approach

In our proposed approach some of the ideas from these previous works are followed, e.g., specialized horizontal distribution and reordering of data among similar peers, and taken into a new direction. This new approach focuses on the following concepts:

Focus on Graph Structure: Instead of treating the semantic data as specialized relational database, we focus on its unique structure in the form of a directed graph.

Centralized Optimization: Until today, only peer-to-peer approaches really focus on the graph structure. In contrast to peer-to-peer approaches, we propose a centralized approach, as the data distribution can be more controlled in this way.

Cloud based: The system should easily adjustable to data growth, so it will have a cloud-based structure of many nodes with same capabilities.

3 RDFCLOUD SYSTEM

Our system, which we call the RDFCloud system, can be characterized in the following way:

Self-optimizing: During times of little activity the system can rearrange the data distribution in order to improve performance (by redistributing data such that local joins can be processed as much as possible and distributed joins are avoided during query processing). This can be done repeatedly until an optimum is reached.

Cloud computing: The system consists of an arbitrary number of storage and computation nodes. These nodes all have the same capabilities and may be removed or added to the system at any time. In contrast to peer-to-peer systems, node activity is concerted by a central master node.

Distributed storage: Data stored within the system is evenly distributed between the nodes.

Distributed query processing: Queries targeting the data are processed in parallel on all relevant nodes.

Semantic Web data: The technologies used for storing and processing data are RDF and SPARQL, respective standards for semantic web data.

3.1 Overview

The RDFCloud system is essentially a two-tiered network with exactly two different kinds of nodes: A single master node and an unrestricted number of slave nodes. External access to the system is made via a master client, which connects to the master node to request modification of the stored RDF data or query for the result of a single SPARQL query.

Actual RDF data is only stored at the slave layer, while global information about the network structure and meta-data about the RDF graph is stored at the master node. The master node is responsible for the distribution of new RDF data and combination of the individual responses to SPARQL queries by the slave nodes. Furthermore, the master node can instruct slave nodes to transfer parts of the stored RDF data to other slave nodes in order to optimize the performance of SPARQL queries.

There are two kinds of connections within the nodes of the RDFCloud system: The first kind is that between the distinguished master node and a slave node, the second kind is between two nodes in the slave layer. While the master-slave connection is permanent and never closed during the lifetime of the system, with the exception of an irreversible planned termination of a slave node, the connections between two slave nodes are temporary. Connections between slave nodes are not supported by the Map-Reduce framework. Hence we do not use the Map-Reduce framework and implement our own Cloud infrastructure.

A slave node only connects to another slave node in one of two cases:

- Either during the execution of a SPARQL query, during which the RDF graph is traversed, and a handover to other nodes is necessary to complete the computation, or
- During an optimization of the RDF graph to exchange RDF nodes in order to increase the performance of SPARQL queries.

Inter-slave connections for SPARQL processing are unidirectional, while connections initiated for optimization can be bidirectional.

3.2 Representation of RDF data

Semantic Web data is a set of triples, where the first component is called the subject (S), the second the predicate (P) and the third the object (O) of the triple. More formally:

Definition (RDF triple): Assume there are pairwise disjoint infinite sets I, B and L, where I represents the set of IRIs, B the set of blank nodes and L the set of literals. We call a triple $(s, p, o) \in (I \times B) \cup I \cup (I \times B \times L)$ an RDF triple, where s represents the subject, p the

predicate and *o* the object of the RDF triple. We call an element of $I \cup B \cup L$ an RDF term. □

Semantic Web query evaluators such as RDF3X [17, 18] and Hexastore [24] as well as our RDFCloud system use dictionary indices to map RDF terms into integer ids. One advantage of ids is lower space requirements in the evaluation indices storing the input RDF triples as an integer is stored instead of a possibly large string. Solutions using ids consume less space such that the memory footprint is smaller and/or more solutions can be processed without swapping to hard disks and thus improving the performance. Finally, in a distributed scenario the transfer costs over network connections are significantly decreased because much less bytes must be transferred. Our RDFCloud system maintains the dictionary indices on the master node, such that the slave nodes only have to process integer ids instead of space-consuming string representations and only integer ids are transferred over network.

Using ids has disadvantages in seldom cases when operations like sorting or relational comparisons like <, ≤, ≥, and > require the RDF terms instead of the ids causing high costs for large intermediate results because of the materializations of the RDF terms. Furthermore, displaying the final query result has also high costs whenever the query result is large. However, the advantages typically outweigh the disadvantages of using ids for large-scale datasets.

One dictionary index maps RDF terms into integer ids; one translates integer ids back into RDF terms. The dictionary indices do not fit into main memory for large-scale datasets such that our RDFCloud system uses a B+-tree for the dictionary index for the mapping from the string representation to the integer id, and a disk-based array for the other direction from integer ids to the string representation. When storing RDF terms in the dictionaries, we use difference encoding in order to save storage space: we determine common left substrings of the current and previously stored strings and store only the length of the common left substring together with the remaining right substring of the current string. Furthermore, after transforming id values of query results back to RDF terms, we cache the RDF terms with their ids together in order to avoid multiple materializations. We use the strategy of least recently used (LRU) caches for the accesses to the B+-tree nodes in order to further improve the performance of these materializations.

The dictionary indices are used to transform RDF triples into id triples, which are consisting of ids instead of RDF terms and are then stored in the cloud: Id triples are obtained from RDF triples by using the dictionary index from strings to ids and mapping the RDF terms of the subject, predicate, and object from the triples to their ids.

3.3 Master Node

The master node has several key responsibilities in the RDFCloud system:

Maintaining Topology: The master node must keep track of all participating slave nodes in the system. This includes maintaining permanent connections and alerting clients when slave nodes are missing.

Dictionary: As the RDF data is not stored on the slave nodes in textual form, the master node has to maintain a mapping between numerical and textual representations.

Construct Query Graphs: SPARQL queries received from clients need to be parsed by the master node, transformed to a form usable for distributed computations and transfer them to the slave nodes.

Result Processing: Once all slave nodes returned their results to the SPARQL query, these results need to be combined by the master node and mapped back to their textual representation using the maintained dictionary.

Initiate Optimization: If the previous optimization iteration already yielded good results (and hence the optimum is not yet reached), if the topology has changed or many updates have been processed in the meanwhile, an optimization iteration is expected to improve the performance much and hence scheduled. When an optimization iteration is scheduled and the system is idle, the master node starts optimization phases involving all slave nodes and coordinates RDF data movement among the slave nodes in order to improve performance.

3.4 Slave Node

The main purposes of a slave node are the storage of parts of the RDF graph and the processing of SPARQL queries on the stored parts. One of the main decisions which need to be made in a distributed storage system is the distribution method and the way of locating data. In the RDFCloud system all slave nodes have equal capabilities and may store arbitrary data. A way to do this would be the completely unrestricted distribution of triples among all slave nodes. The advantage of this approach would be the ability to achieve perfectly balanced nodes, as the smallest possible storage units can be freely moved around.

However, this would lead to serious problems: It would be very difficult to locate specific data and only the most primitive queries could be evaluated efficiently, as any data could be located on any node. Data structures storing this information would be as big as the data itself. For this reason, the RDFCloud system takes a different approach: Each slave node in the system can take control over certain subsets of the RDF

data. Information about which slave controls which subset is distributed by the master node.

The shape of these data subsets follows naturally from the graph representation of RDF data: A slave node can take control of a vertex within the graph, i.e. over a specific RDF subject.

Definition (Authority): A slave node is the authority for an RDF subject. ⇔ The slave node is the only node storing triples having this subject. □

Table 1: Example RDF Data

Triple	S	a	c	b	a	a
	P	p	p	p	p	p
	O	c	d	c	b	d

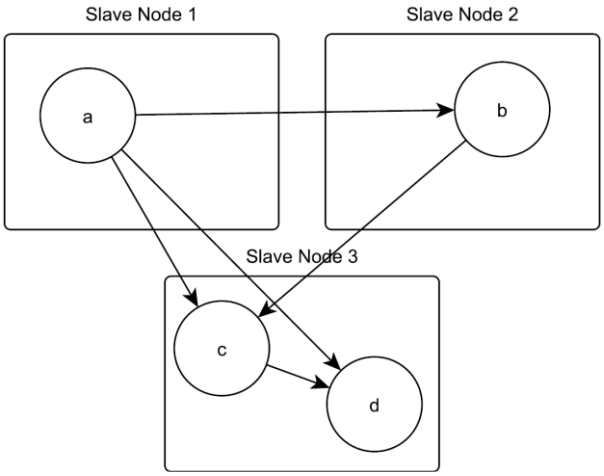

Figure 1: Example of subject authorities

As an example, the RDF data of Table 1 is distributed among three nodes. Figure 1 visually presents the distribution of these triples in slave nodes numbered 1 to 3. Here, subjects and objects of triples become (unique) nodes in the RDF graph, and the predicate a directed edge from the subject to the object node. In this example, node 1 is the authority for subject "a", node 2 the authority for subject "b" and node 3 for subject "c". Important to note here is the fact that node 3 is not the authority for subject "d", even though the graph representation might suggest this. No triple containing "d" as a subject exists, so there is no authority for "d". Furthermore, a slave node may be the authority of several subjects, not only one.

3.5 Modification of RDF Data and Topology

In this section it is shown how to modify the data stored in the distributed RDFCloud system. Operations explained here include:

- Insertion of RDF triples for storage on the slave nodes,
- deletion of data,
- removal of a slave node without changing stored data, and
- inclusion of an additional slave node into the system.

3.5.1 Insertion of RDF Data

When new RDF information needs to be inserted into the distributed storage, the client first parses the local source data and transforms it into string based triples. These triples are then transferred to the master node over a temporary connection, where they are processed further.

As a first step, the triples are converted into a more efficient representation by replacing each literal into an integral number uniquely representing the original string. In order to achieve this, the master node makes subsequent lookups into the *dictionary*, a data structure storing bi-directional mappings between literal strings and numbers. This data structure is extended each time a string is encountered which cannot be mapped yet. Already existing mappings may not be modified when doing this, as this would invalidate the data structure on the slave nodes storing the already existing RDF data.

After converting all RDF data into numerical triples, the master node has to distribute it among all connected slave nodes. If a slave node is already the authority for the subject of triple to be inserted, the triple is sent to this slave node. However, in order to save transmission costs, the triples are first collected for each slave node and sent to them in bulk.

If no slave node has the authority for the subject of a triple, the slave node with lowest fill rate, which is defined as

$$\frac{\text{number of triples of the slave node}}{\text{total number of triples}},$$

could become the authority. However, in order to boost local joins, the authority should be assigned to a slave node containing already triples having the subject of the triple to be inserted as object. The algorithm to do this is as follows:

1. Contact each slave node and:
   a. Transfer a distinct list of "subject" literals of the new data to the slave node.
   b. The slave node checks if it has at least one entry in its local indices containing the subjects.
   c. For all subjects found this way the slave node is the authority and the master node is informed of this.

   d. The master node responds with the full triples containing the subjects to the slave node, which adds the data to its local indices.
   e. This data is removed from the list of triples to be added to the system.
2. All data still left is distributed among the slave nodes according to their respective fill rates.
3. Authority information of the new subjects is broadcast to all slave nodes by the master node.

The performance of this operation is highly dependent on the distribution of data among the slave nodes and the new data. In the optimal case, the very first node is subject authority for all triples and the algorithm can immediately terminate. The worst case is that the slave node checked last is the authority for all subjects.

3.5.2 Deletion of RDF Data

Removing data from the system works in a similar way to adding data:

1. After conversion of triples to be deleted to numerical representation contact each slave node and:
   a. Transfer triples to be deleted to the slave node.
   b. The slave node looks up its local indices and removes entries if necessary.
   c. If the last entry containing a specific subject is deleted inform the master node of lost authority.
2. Broadcast removed authority to all slave nodes by the master node.

3.5.3 Removal of Slave Node

The removal of a slave node is possible as long as there is at least one node available which can take over the RDF data currently stored on the slave node to be removed.

First, all subjects the slave node is an authority for are transferred to the master node. Using information about the fill level of all remaining slave nodes, the master node now evenly distributes the RDF subjects to available recipients. After the redistribution is complete, the master informs all nodes of the changed subject authorities.

As only the fill levels are taken into account during redistribution, it is not guaranteed that the distribution is optimal for the system's performance. In the example shown (Figure 2), the number of edges crossing slave node boundaries remains equal, even though there are better distributions. Subsequent optimization steps can further optimize the distribution.

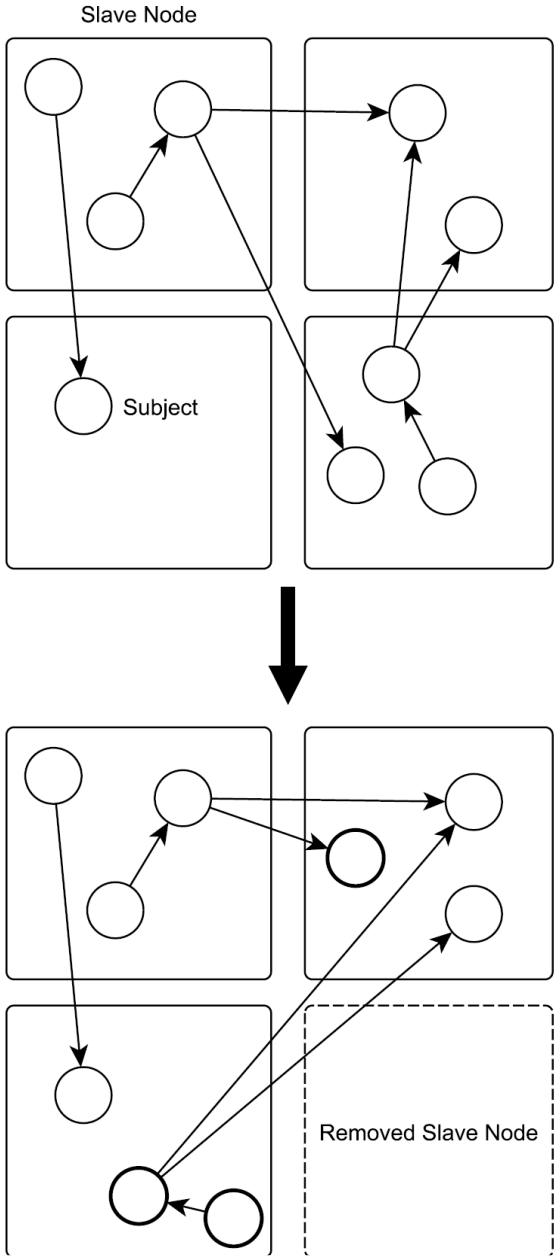

Figure 2: Removal of a slave node

3.5.4 Adding new Slave Node

In case the slave nodes in the system cannot accept more RDF data because of memory constraints or because calculation load during queries becomes too high, more nodes can be included as well.

The master node initiates the communication with a new slave node and asks for its storage capabilities. The master node contacts the already existing slaves, which then offer a certain amount of RDF subjects depending on their current fill rate. The total amount of RDF transferred to the new slave node is dependent on the

average fill rate of the whole system to ensure optimal load distribution. In contrast to the removal of nodes, the optimality of the final distribution can be influenced by the slave nodes. Instead of randomly choosing subjects they are the authority of, they offer subjects with lowest count of total local subject edges. This ensures that the total number of inter-slave edges increases by at most this count of local edges, as external edges either remain external, or become internal edges in the new slave node.

An example for the inclusion of a new slave node is shown in Figure 3. Here, the number of inter-slave edges increases by the minimum possible of 2, while remaining balance in node fill level.

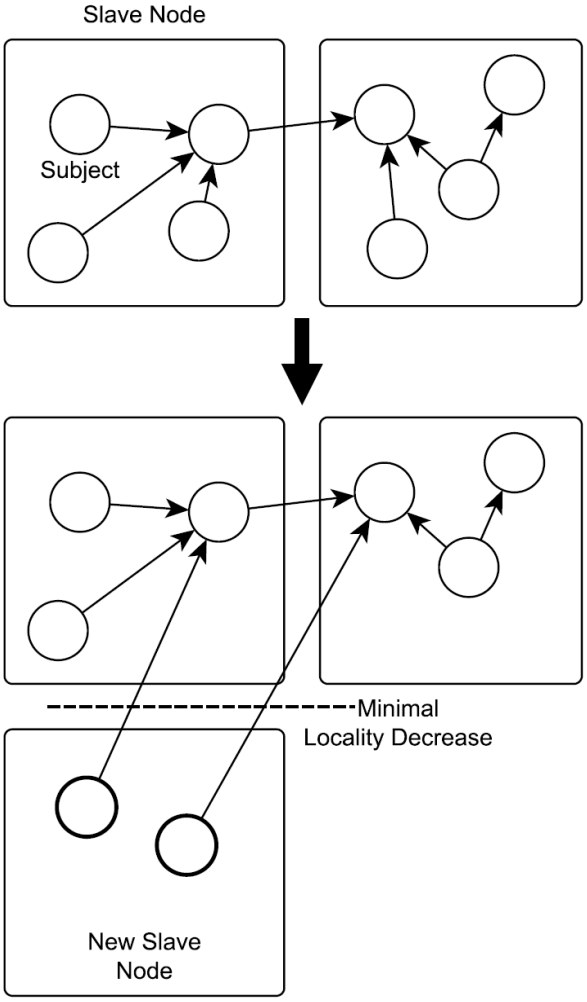

Figure 3: Insertion of a new slave node

3.6 Optimization of Data Distribution

One of the characteristics of the RDFCloud system is the ability to continually optimize itself in order to improve its performance. By not having a fixed mapping of specific data to specific nodes through a distribution function, data can continually be reordered among the slave nodes. The key element to achieve better performance is choosing a well-defined characteristic of the system to alter and find a method which has a high chance of optimizing this characteristic. In the RDFCloud system, this characteristic is the locality of data.

3.6.1 Locality

The main idea behind the optimization process is that the most expensive operation in a distributed database is the transfer of data from one node to another. In non-distributed systems, access to permanent storage is the most expensive operation, but this takes a much lower priority in a distributed system like RDFCloud. Especially join operations during query executions should be executed preferable only on local data.

Definition (Locality between two slave nodes): Let e be the number of edges crossing node boundaries. The locality of RDF data of two slave nodes is 0 if e is 0, otherwise $\frac{1}{e}$. □

Definition (Global Locality): The global locality of RDF data in an RDFCloud system is the sum of localities of all disjoint pairs of slave nodes. □

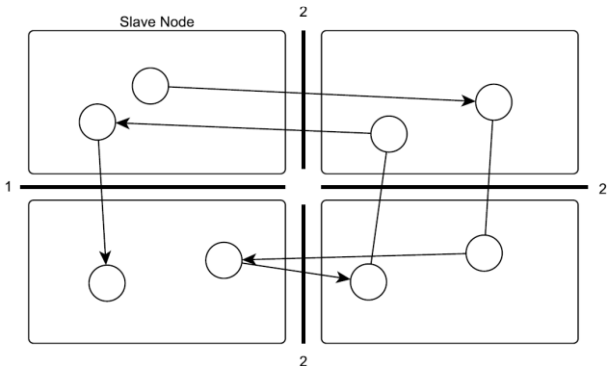

Figure 4: Locality

For example in Figure 4 one edge is crossing between the slave nodes on the left: Their locality is $\frac{1}{1} = 1$. Two edges are crossing each of the following three pairs of slave nodes: the pair of slave nodes on the top, the pair of slave nodes on the right as well as the pair of slave nodes at the bottom. Hence, each of these pairs of slave nodes has the locality $\frac{1}{2}$. No other pairs of slave nodes have edges crossing them, i.e. the locality of all other pairs of slave nodes is 0. Therefore, the global locality in Figure 4 is $1 + 3 * \frac{1}{2} = \frac{5}{2} = 2.5$.

3.6.2 Optimizing Locality

Clustering: One possible way to maximize the locality of data is the employment of a class of algorithms used for cluster analysis. In order to apply these algorithms, the graph representing the RDF data first has to be converted into a form these algorithms can work on. This means that the graph has to be projected to a system in which two subjects have a well-defined distance, and subjects with short predicate-paths between them are placed in close proximity.

Definition (Distance): The distance between two subjects in the RDF graph is defined as the length of the shortest path between them. If no path connects the subjects, the distance is the length of the longest possible path in the graph. □

General clustering algorithms do not have a concept of "balance". This leads to a bad distribution of computation load, even though the data locality is optimal. Another problem is that a clustering algorithm which finds optimal clusters even for problem instances as simple as points in a 2-dimensional Cartesian space is always NP-complete (see [15]). This makes such algorithms unusable for the RDFCloud system, as potentially very large RDF data sets are stored and the algorithm would also have to run every time the data is changed. This makes it prohibitively expensive to use clustering algorithms.

Heuristics: The problem of NP-completeness of all algorithms finding an optimal solution to maximizing data locality lead to the development of heuristics which scale better with the amount of data, such that handling big data becomes realistic.

Instead of immediately finding an optimal solution, an iterative approach to reducing inter-slave edges is taken. The algorithm is designed to quickly return possible improvements through subject authority changes, which is important for a self-optimizing system as optimization phases must be short in order to keep the system responsive for queries.

The basic idea is that the slave nodes are not only used to divide computation load of queries, but also take part in distributed optimization. To this end, they analyze the local subset of the RDF graph.

The heuristic algorithm is as follows:

1. The master node signals all slave nodes the start of an optimization phase.
2. Each node generates two lists. The number of entries in these lists is limited by the master node, making them independent of the total amount of RDF data:
   a. One list contains the most referenced remote subjects on other slave nodes.
   b. Another list includes authority subjects which have the least count of incoming edges (i.e. referenced with a predicate).
3. Both lists are transferred to the master node by all slave nodes.
4. The master node compares these lists and tries to find matches between those lists. These matches are stored in a candidate list.
5. If a previously defined amount of candidates was not met yet, candidates are added from the list containing often referenced foreign subjects. As a priority criterion the fill rate of slave nodes is used. The lower the fill rate, the lower the chance that a subject authority will be removed from it.
6. The candidate lists are sent to the receiving slave nodes, which contact their respective partner. The partner informs the slave node of local edges which would become cross-slave edges in case of moving the authority. If the move is an improvement, the authority transfer is initiated. Otherwise, the subject is put on a temporary black list together with the number of local triples having this subject as object, i.e., the authority of this subject will not be moved any more until the authorities movements may become beneficial again.
7. All slave nodes are informed by the master node of authority changes. Slave nodes modify their local list of remote authorities accordingly.

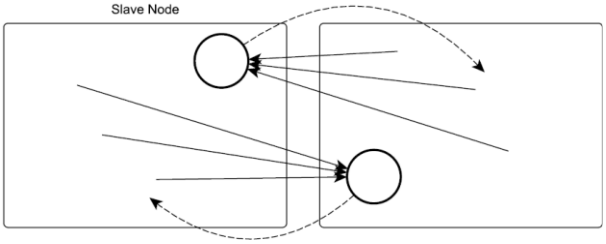

Figure 5: Perfect Authority Exchange

There are several possible outcomes when an authority exchange is tested for locality improvement. The best outcome is shown in Figure 5; none of the subjects have local incoming edges, so switching them between the slave nodes is a clear improvement in locality. As both nodes gain and lose an authority, no imbalance is created either.

Another possibility is the attempt of a one-sided transfer. This is depicted in Figure 6, where the right slave node requests the subject authority because there are many predicates pointing at it. In the shown example, the transfer fails, because the internal edges to this subject outnumber the external ones by one. A transfer would result in lowered locality, so the subject

(together with some of its context information like the number of local triples containing it as object) is black-listed for future attempts. Only if an authority movement may become beneficial again (i.e., the context has changed and we have a lower number of local triples containing it as object), an authority movement of the considered subject is checked again.

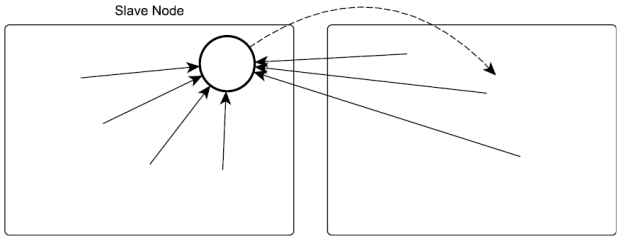

Figure 6: Failed Authority Transfer

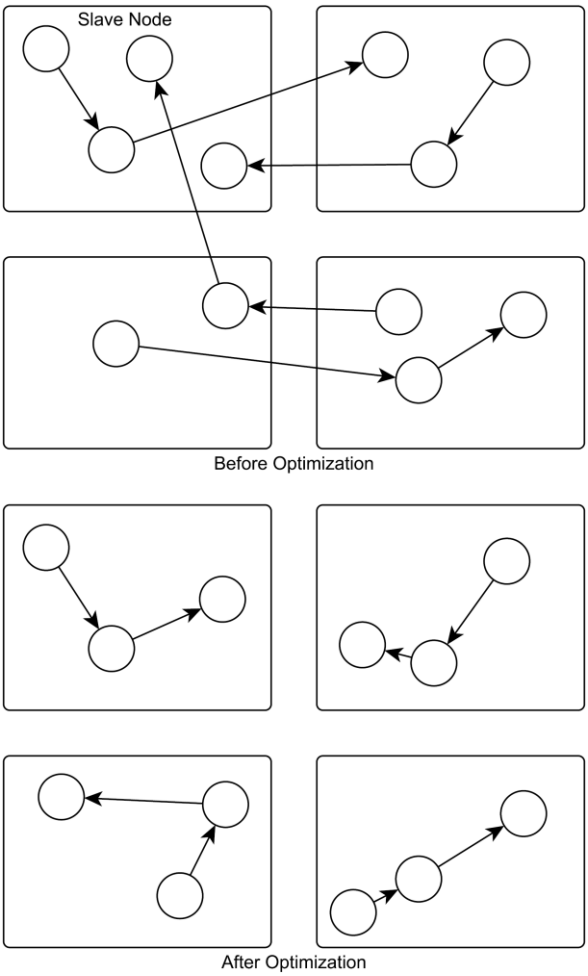

Figure 7: Optimization of RDF Graph with Short Paths

Effects of the Heuristics: To make the effects of optimizing the locality of RDF data visible, this section gives two examples of how the data distribution among the slave nodes changes for specific highly structured types of RDF graphs, which often occur in real-world RDF data.

The type of RDF datasets for which the optimization process of RDFCloud works extremely well has the graph representation of set of trees with short paths. For an example, see Figure 7. It can be seen that data with short paths can be optimized very well and the system finally reaches a state where there are no edges between slave nodes.

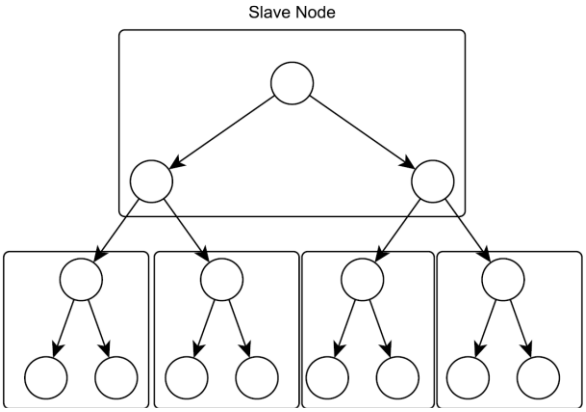

Figure 8: Optimization of RDF Tree Graph

In the case of long chains of predicates leading from very few "root" subjects to a large number of "leaf" subjects, the resulting graph takes on the shape of a large tree (in the case of a single root) (see Figure 8) or a forest (for multiple roots). The main problem for this type of graphs lies in the balance of both, minimizing the number of edges between slave nodes and evenly distributed fill levels among all slaves.

Most queries follow the graph structure of its queried RDF data (e.g., the queries of the SP2B benchmark [20]). Thus the proposed distribution approach is ideally suitable for executing queries. Only exotic queries like those having a join between a predicate and a subject or object cannot be handled efficiently. As a heuristic for data distribution is used, it is possible that the system ends up in a local minimum instead of finding the global minimum. However, standard approaches like simulated annealing can be also applied here to overcome these problems.

3.7 Discussion of Possible Bottlenecks

The **bottleneck during query processing** is the speed of the network, as the main query processing tasks are distributed among the slave nodes. The master node *only* initiates the query processing and collects the results from the slave nodes. The additional task of the master node, the mapping between the integer ids and the

textual representations with the help of the dictionary, is also fast. On the other side huge amount of intermediate results need to be transferred between the slave nodes during query processing.

The **bottleneck during optimizing the data distribution** is the network speed as well as the speed of the master node. Again a lot of data need to be transferred for collection of statistics and informing slave nodes with candidate lists of authorities, which are analyzed for movement. However, also the determination of the candidate lists on the master node itself is a costly and time-consuming task. We will deal with a distributed computation of the candidate lists in our future work.

4 EXPERIMENTAL EVALUATION

In this section experimental results based on the reference implementation of the RDFCloud system will be presented. Both, the optimization process and example queries have been executed on the system. The example system consists of the master node and six active slave nodes, processing data sets of different sizes. All nodes run Linux and Java 1.6 on a 2.33 GHz Dual Core processor with 1 Gigabytes main memory. All nodes are in a LAN with 1 Gbit/s. The RDFCloud system is implemented on top of the non-distributed Semantic Web database LUPOSDATE [8], the code of which is open source [9]. Also other Cloud extensions of LUPOSDATE exist like P-LUPOSDATE [10].

These data sets have been generated with the tools provided by the SP2Bench SPARQL Performance Benchmark [20]. This benchmark was created by observing statistical characteristics of scientific publications since the year 1940 and can generate data sets of arbitrary size which imitate the characteristics. The used set sizes were 250,000, 1,000,000 and 2,500,000 RDF triples.

4.1 Optimizing Data Distribution

First, the distribution of RDF data and optimization over several iterations of the described algorithm is tested. To this end, the optimization process is started repeatedly until a stable configuration is found, i.e., the number of moved authorities is 0, or only negligible changes occur from one iteration to the next, i.e., the edge difference divided by the number of inter-slave edges is below a threshold to be specified. Considering the experiments for query execution, a threshold of 5% already yields sufficient results, which is reached after an *optimization rate* of about 33%. We define the optimization rate to be the number of optimization iterations divided by the number of total optimization iterations until a stable configuration is reached. After each iteration, the number of moved subject authorities and the inter-slave edge count are calculated (see tables 2, 3 and 4).

Table 2: Optimizing Data Distribution (250k triples)

Iteration	0	1	2	3	4	5	6
Inter-slave edges	75453	56752	55326	54882	54769	54747	54742
Moved Authorities	16300	1527	323	88	13	4	0
Edge Difference	-18701	-1426	-444	-113	-22	-5	0

Table 3: Optimizing Data Distribution (1,000k triples)

Itera-tion	0	1	2	3	4	5	6	7	8
Inter-slave edges	303456	258692	233523	220 784	220 108	220 006	219 961	219 905	219 903
Moved Autho-rities	31566	23092	14747	1526	336	250	56	2	0
Edge Diffe-rence	-44764	-25169	-12739	-676	-102	-45	-56	-2	0

Table 4: Optimizing Data Distribution (2,500k triples)

Iteration	0	1	2	3	4	5	…	13
Inter-slave edges	771684	707108	673453	648773	625166	599756		532064
Moved Authorities	30442	24943	23272	23963	25331	25732	…	64
Edge Difference	-64576	-33655	-24680	-23607	-25410	-25732		-70

These results show that the heuristic algorithm can effectively improve the locality of data. In relatively few iterations the edge count can be reduced by roughly a third in these test cases. The number of optimizing movements of subject authorities is decreasing in a strongly logarithmic fashion, as fewer nodes with high inter-slave connectivity are found.

4.2 Query Execution

The SP2Bench project also provides several queries to be executed with the data set. However, the optimization algorithm has hardly any effect on the runtime of most of the given SP2Bench queries. The reason for this lies in the focus of the provided queries on sub-graphs within the RDF graph, which take the form of a star, centered among certain subjects. Because of this, no inter-slave connectivity is needed and the optimization algorithm has almost no influence. The almost insignificant increase in performance can be attributed to a lower

count of remote bindings the slave nodes have to process.

In a sense, the RDF graph in the RDFCloud system is already in its final optimized form from the very beginning, as triples with the same subject are stored on the same node by design. Because of this, the queries are only testing the performance of the underlying query engine, LUPOSDATE, and the speed of network communication. Note that these numbers as well as the query execution times of other types of queries for the initial data distribution (where no optimization iteration has been done already), are comparable to the times achieved in peer-to-peer systems distributing the data according to the subject of the triples.

```
SELECT ?name ?document WHERE {
    ?document dc:creator ?author .
    ?author foaf:name ?name. }
```

Figure 9: Query E1 (Find all documents and their respective authors)

Table 5: Query Execution Times of E1 (Avg. of 10 executions)

Optimization rate / #Triples	0%	33%	67%	100%
250k	8.9 s	6.9 s	6.3 s	6.1 s
1,000k	21.5 s	18.7 s	17.6 s	16.9 s
2,500k	53.4 s	41.3 s	39.2 s	38.3 s

```
SELECT ?name ?document ?jname WHERE {
    ?document dc:creator ?author .
    ?author foaf:name ?name .
    ?document swrc:journal ?journal .
    ?journal dc:title ?jname. }
```

Figure 10: Query E2 (Find all journal articles, their respective authors and the name of the journal.)

Table 6: Query Execution Times of E2 (Avg. of 10 executions)

Optimization rate / #Triples	0%	33%	67%	100%
250k	13.4 s	8.2 s	8.1 s	8.1 s
1,000k	39.4 s	26.7 s	22.1 s	21.9 s
2,500k	71.4 s	59.3 s	53.2 s	52.9 s

The queries in Figure 9 and Figure 10 are better suited to show the effects of graph optimization done by the RDFCloud system. The effects of reduced inter-slave edges are clearly visible for the query in Figure 9 (see Table 5), as the graph needs to be traversed for each result. The optimization algorithm can quickly improve the query by moving the author subjects belonging to a document onto the same slave node. This effect is even more pronounced when the path taken in the RDF graph is increased further to at least two edges (see Figure 10). It can be seen (see Table 6) that the first iterations of the optimization algorithm have the greatest effect on query runtime. A possible explanation for this is that the first subject authority movements have a high chance of bringing leaf nodes to the node their respective RDF tree is located at.

4 SUMMARY AND CONCLUSIONS

The goal of our research is the development of a distributed system which exploits the characteristics of RDF data. The methods developed to optimize the RDF graph were shown to measurably improve performance of SPARQL queries, although this is dependent on the type of queries given to the system.

The concept of subject authority in a distributed system is a good compromise between the freedom of data distribution and keeping the optimization and query algorithms comparatively simple and effective. This approach still has potential for improvement, e.g. in form of combination with effective indices to reduce network load.

REFERENCES

[1] G. Adamku, H. Stuckenschmidt: Implementation and Evaluation of a Distributed RDF Storage and Retrieval System. *The IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, Halifax, Canada, pages 393 - 396, 2005.

[2] L. A. Barroso, J. Dean, U. Hölzle: Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23 (2), pages 22-28, 2003.

[3] D. Battré: Query Planning in DHT Based RDF Stores. *The IEEE International Conference on Signal Image Technology and Internet Based Systems (SITIS)*, Bali, Indonesia, pages 187 – 194, 2008.

[4] M. Cai, M. Frank: RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. *Proceedings of the 13th international conference on World Wide Web (WWW)*, New York, USA, pages 650 - 657, 2004.

[5] H. Choi, J. Son, Y. Cho, M. Kyoung Sung, Y. D. Chung: SPIDER: a system for scalable, parallel / distributed evaluation of large-scale RDF data. *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, Hong Kong, China, pages 2087 - 2088, 2009.

[6] P. Castagna, A. Seaborne, C. Dollin: A Parallel Processing Framework for RDF Design and Issues. *Technical report*, HP Laboratories, pages 1 -14, 2009.

[7] J. Dean, S. Ghemawat: MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), pages 107-113, 2008.

[8] S. Groppe: *Data Management and Query Processing in Semantic Web Databases*, Springer, 2011. ISBN 978-3-642-19356-9

[9] S. Groppe: LUPOSDATE Open Source, https://github.com/luposdate/luposdate.

[10] S. Groppe, T. Kiencke, S. Werner, D. Heinrich, M. Stelzner, L. Gruenwald: P-LUPOSDATE: Using Precomputed Bloom Filters to Speed Up SPARQL Processing in the Cloud, *Open Journal of Semantic Web (OJSW)*, RonPub, 1(2):25-55, 2014. http://www.ronpub.com/publications/ojsw/OJSW-v1i2n02_Groppe.html.

[11] M. F. Husain, P. Doshi, L. Khan, B. Thuraisingham: Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce. *Proceedings of the 1st International Conference on Cloud Computing (CloudCom)*, Beijing, China, pages 680–686, 2009.

[12] M. Harasic, A. Augustin, P. Obermeier, R. Tolksdorf: RDFSwarms: selforganized distributed RDF triple store. *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Sierre, Switzerland, pages 1339 - 1340, 2010.

[13] P. Kulkarni: Distributed SPARQL query engine using MapReduce. *Master's thesis*, University of Edinburgh, 2010.

[14] Linked Data, "Linked Data - Connect Distributed Data across the Web," 2014. Online available: http://www.linkeddata.org

[15] M. Mahajan, P. Nimbhorkar, K. Varadarajan. The Planar k-Means Problem is NP-Hard. *Proceedings of the 3rd International Workshop on Algorithms and Computation (WALCOM)*, Kolkata, India, pages 274 - 285, 2009.

[16] J. Myung, J. Yeon, S. Lee: SPARQL basic graph pattern processing with iterative MapReduce. *Proceedings of the Workshop on Massive Data Analytics on the Cloud (MDAC)*, Raleigh, North Carolina, Article No. 6, 2010.

[17] T. Neumann, G. Weikum: Scalable join processing on very large RDF graphs. *Proceedings of the ACM SIGMOD International Conference on Management of data (SIGMOD)*, Providence, Rhode Island, USA, pages 627 - 640, 2009.

[18] T. Neumann, G. Weikum: RDF3X: a RISCstyle engine for RDF. *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, Auckland, New Zealand, pages 647 - 659, 2008.

[19] P. Ravindra, V. V. Deshpande, K. Anyanwu: Towards scalable RDF graph analytics on MapReduce. *Proceedings of the Workshop on Massive Data Analytics on the Cloud (MDAC)*, Raleigh, North Carolina, Article No. 5, 2010.

[20] M. Schmidt, T. Hornung, G. Lausen, C. Pinkel: SP2Bench: A SPARQL Performance Benchmark, *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, Shanghai, China, pages 222 - 233, 2009.

[21] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, J. Broekstra: Index structures and algorithms for querying distributed RDF repositories. *Proceedings of the 13th international conference on World Wide Web (WWW)*, New York, USA, pages 631 - 639, 2004.

[22] M. Svoboda, I. Mlýnková: Efficient querying of distributed linked data. *Joint EDBT/ICDT Ph.D. Workshop*, Uppsala, Sweden, pages 45-50, 2011.

[23] B. P. Vandervalk, E. L. McCarthy, M. D. Wilkinson: Optimization of Distributed SPARQL Queries Using Edmonds' Algorithm and Prim's Algorithm, *International Conference on Computational Science and Engineering (ICCSE)*, Vancouver, Canada, pages 330 – 337, 2009.

[24] C. Weiss, P. Karras, A. Bernstein: Hexastore: sextuple indexing for semantic web data management. *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, Auckland, New Zealand, pages 1008 - 1019, 2008.

[25] World Wide Web Consortium, Resource Description Framework, W3C Recommendation, http://www.w3.org/RDF, 2004.

[26] World Wide Web Consortium, SPARQL Query Language for RDF, W3C Recommendation, http://www.w3.org/TR/rdf-sparql-query, 2008.

[27] World Wide Web Consortium, SPARQL 1.1 Query Language, W3C Recommendation, http://www.w3.org/TR/sparql11-query, 2013.

AUTHOR BIOGRAPHIES

Sven Groppe earned his diploma degree in Informatik (Computer Science) in 2002 and his Doctor degree in 2005 from the University of Paderborn. He earned his habilitation degree in 2011 from the University of Lübeck. He worked in the European projects B2B-ECOM, MEMPHIS, ASG and TripCom. He was a member of the DAWG W3C Working Group, which developed SPARQL. He was the project leader of the DFG project LUPOSDATE, and is currently the project leader of two research projects, which research on FPGA acceleration of relational and Semantic Web databases. His research interests include Semantic Web, query and rule processing and optimization, Cloud Computing, peer-to-peer (P2P) networks, Internet of Things, data visualization and visual query languages.

Johannes Blume received his M.Sc. in Computer Science in 2012 from the University of Lübeck, Germany, where he wrote his master thesis at the Institute of Information Systems (IFIS). He currently works as a software engineer in the financial sector.

Dennis Heinrich received his M.Sc. in Computer Science in 2013 from the University of Lübeck, Germany. At the moment he is employed as a research assistant at the Institute of Information Systems at the University of Lübeck. His research interests include FPGAs and corresponding hardware acceleration possibilities for Semantic Web databases.

Stefan Werner received his Diploma in Computer Science (comparable to Master of Computer Science) in March 2011 at the University of Lübeck, Germany. Now he is a research assistant/PhD student at the Institute of Information Systems at the University of Lübeck. His research focuses on multi-query optimization and the integration of a hardware accelerator for relational databases by using run-time reconfigurable FPGA's.