Open Access

Open Journal of Semantic Web (OJSW)
Volume 1, Issue 2, 2014

http://www.ronpub.com/journals/ojsw
ISSN 2199-336X

P-LUPOSDATE: Using Precomputed Bloom Filters to Speed Up SPARQL Processing in the Cloud

Sven Groppe [A], Thomas Kiencke [A], Stefan Werner [A], Dennis Heinrich [A], Marc Stelzner [A], Le Gruenwald [B]

[A] Institute of Information Systems (IFIS), University of Lübeck, Ratzeburger Allee 160, D-23562 Lübeck, Germany, groppe@ifis.uni-luebeck.de, kiencket@gmail.com, {werner, heinrich, stelzner}@ifis.uni-luebeck.de
[B] School of Computer Science, University of Oklahoma, Norman, OK, USA, ggruenwald@ou.edu

Abstract

Increasingly data on the Web is stored in the form of Semantic Web data. Because of today's information overload, it becomes very important to store and query these big datasets in a scalable way and hence in a distributed fashion. Cloud Computing offers such a distributed environment with dynamic reallocation of computing and storing resources based on needs. In this work we introduce a scalable distributed Semantic Web database in the Cloud. In order to reduce the number of (unnecessary) intermediate results early, we apply bloom filters. Instead of computing bloom filters, a time-consuming task during query processing as it has been done traditionally, we precompute the bloom filters as much as possible and store them in the indices besides the data. The experimental results with data sets up to 1 billion triples show that our approach speeds up query processing significantly and sometimes even reduces the processing time to less than half.

Type of Paper and Keywords

Regular research paper: *Semantic Web, Cloud, Query Processing, Query Optimization, SPARQL*

1 Introduction

The data volume world-wide will grow to 40 Zettabytes by the year 2020 according to a study [19] done in 2012, such that the data volume is doubled every two years. Handling such big data volumes is one of the current challenges.

Cloud Computing has been born partly from the need of storing and processing large data sets with dynamical reallocation of hardware resources per demand for scaling up with higher requirements and larger data processing volumes. Cloud Computing offers cost-effectiveness (as resources are not used when not needed), scalability (per dynamic reallocation of hardware resources) and a high availability (by dynamically switching from erroneous to correct running resources).

One of the main objectives of the *Semantic Web* [54] and especially of the *Linking Open Data (LOD)* project [29] is to offer and administer large machine-processable data sets which are linked with each other. Hence, Semantic Web data has a high capability of integration and is the ideal choice to be used for big data. Already in September 2011, the freely available Semantic Web data contains over 30 billions triples in nearly 300 datasets with over 500 million links between these data sets [29].

It is therefore a promising idea to marry Cloud Computing with the Semantic Web. Utilizing Cloud technologies for storing large-scale Semantic Web data and pro-

cessing Semantic Web queries on top of it is not a new idea (e.g., [40, 34, 25, 24, 44, 43, 13]). Different ways to optimize Semantic Web query processing in the Cloud have been proposed (e.g., [34, 44, 25, 44, 43]), but the research is still in its early stage.

A *bloom filter* [10] is a space-efficient probabilistic data structure for testing the membership of an element in a set. False positive matches are possible, i.e., the test of membership returns either *possibly in set* or *definitely not in set*. Bloom filters find their applications especially whenever the amount of source data would require impracticably large memory if error-free hashing techniques were applied. Bloom filters typically use bit vectors as internal data structure. This compact representation also allows saving transmission costs in distributed scenarios.

Bloom filters are traditionally computed e.g. for joins in relational databases during the index scan of one input relation by applying a hash function to the values of the join attribute(s) and setting corresponding bits in a bit vector. For the other input relation, the bit vector is used to filter unnecessary tuples as early as possible during index scans, such that succeeding operations have less intermediate results and are hence faster processed. In a Cloud system, where data transmissions in the network have high costs, bloom filters might help to minimize the data volume transferred in the network. However, the dynamic, on-demand computation of bloom filters would require additional distributed processing steps in a Cloud system and is hence very expensive. We propose to precompute bloom filters and store them like data in the Cloud to avoid these expensive distributed processing steps. We will later in Section 4.1.1 argue how updates can be processed without drawbacks in our approach.

Our main contributions are the following:

- An optimized system for Cloud-based large-scale Semantic Web data storing and query processing.
- Utilization of bloom filters to minimize intermediate results and network traffic.
- Precomputation of bloom filters to avoid expensive operations in the Cloud during query processing, but dealing also with frequent updates.
- Smooth integration of the precomputed bloom filters in the data distribution strategy.
- A comprehensive experimental analysis for large-scale data sets up to 1 billion triples showing speedups for nearly all queries. The speedups even increase with larger data volumes.

The rest of this paper is organized as follows: Section 2 provides the basic knowledge concerning Semantic Web and its languages, Cloud technologies and the related work; Sections 3 and 4 present our system architecture and our bloom filter approach; Section 5 contains our experimental performance evaluations; and finally Section 6 concludes the paper with future work directions.

2 BASICS

We introduce the Semantic Web and its basic languages, the relational algebra, and finally Cloud technologies and their languages in the following sections. An overview of existing contributions regarding processing Semantic Web data in the Cloud is also provided.

2.1 Semantic Web

The current World Wide Web is developed for the humans, which can easily understand text in natural language, take implicit knowledge into consideration and discover hidden relationships. The goal of the *Semantic Web* [54] is to enable a machine-processable web [8] allowing new applications for its users. For this purpose, the Semantic Web proposes to structure (web) information which simplifies automatic processing. The Semantic Web initiative of the *World Wide Web Consortium (W3C)* already recommends technologies and language standards in the recent years for driving the idea of the Semantic Web. Among these languages is the *Resource Description Framework (RDF)* [50] for describing Semantic Web data and the RDF query language ***S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage (SPARQL)* [51, 52]. We introduce both languages in the following sections.

2.1.1 Data Format RDF

The *Resource Description Framework (RDF)* [50] is a language originally designed to describe (web) resources, but can be used to describe any information. RDF data consists of a set of triples. Following the grammar of a simple sentence in natural language, the first component s of a triple $(s,\ p,\ o)$ is called the subject, p is called the predicate and o the object. 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 $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ 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.

In visualizations of the RDF data, the subjects and objects become (unique) nodes, and the predicates directed labeled edges from their subjects to their objects. The resulting graph is called *RDF graph*.

Listing 1 shows an example of RDF data consisting of three triples about a book published by `Publisher` with the title `"Pig"` from the author `"Expert"` in the serialization format N3 [7].

```
@PREFIX ex: <http://example.org/>.
ex:book ex:publishedBy ex:Publisher .
ex:book ex:title "Pig" .
ex:book ex:author "Expert" .
```

Listing 1: Example of RDF data

2.1.2 Query Language SPARQL

The World Wide Web Consortium proposed the RDF query language SPARQL [51, 52] for searching in RDF data sets. Listing 2 presents an example SPARQL query. The structure is similar to SQL queries for relational databases. The most important part is the *WHERE*-clause which contains triple patterns. These are used for matching triples. Known components of a triple are directly given in a triple pattern, unknown ones are left as variables (starting with a ?). A triple pattern result consists of variables bound to the values of matching triples. The results of several triple patterns are joined over common variables. All variables given behind the keyword *SELECT* appear in the final result, all others are left out. Besides SELECT-queries, also *CONSTRUCT*- and *DESCRIBE*-queries are available to return RDF data. Furthermore, *ASK*-queries can be used to check for the existence of results indicated by a boolean value. Analogous to N3, SPARQL queries may declare prefixes after the keyword *PREFIX*.

Listing 2 presents an example SPARQL query, the result of which is {(?title="Pig", ?author="Expert")} when applied to the RDF data in Listing 1.

```
PREFIX ex: <http://example.org/>
SELECT ?title ?author
WHERE {
 ?book ex:publishedBy ex:Publisher .
 ?book ex:title ?title .
 ?book ex:author ?author .
}
```

Listing 2: Example of a SPARQL query

Besides this basic structure, SPARQL offers several other features like *FILTER* clauses to express filter conditions, *UNION* to unify results and *OPTIONAL* for a (left) outer join.

SPARQL in its new version 1.1 [52] additionally supports enhanced features like update queries, paths and remote queries.

2.2 Relational Algebra

Relational algebra is used in relational databases. Basic operations of queries in such databases can be expressed in terms of (nestable) operators of the relational algebra [15, 16]. Also SPARQL operators can be transformed [53] to a relational expression (which can be represented by an operator tree or by its more general form, an operator graph). An additional operator is the triple pattern scan, which is a special form of an index scan operator, yielding the result of a triple pattern. Table 1 contains a brief overview of the operators which we use in the following sections.

Figure 1 presents an example of a SPARQL query and its transformation to an operator graph. The result of the triple patterns 1, 2 and 3 are first joined over the variable[1] $?a$, and the triple patterns 4 and 5 over the variable $?b$. Afterwards their results are joined over the variable $?x$.

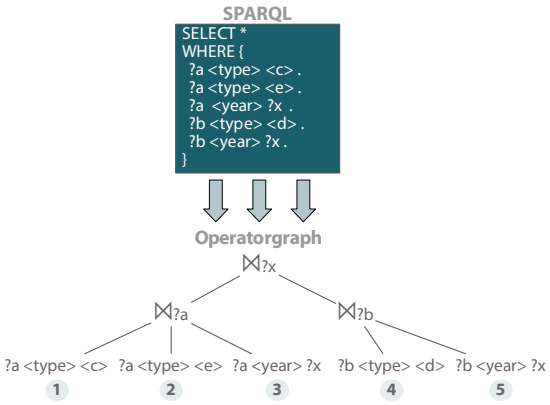

Figure 1: Transformation of a SPARQL query to an operator graph

2.3 Cloud Technologies

The idea of *Cloud Computing* is to allow users to outsource their own software or even IT hardware in order to dynamically react to current workload or hardware requirements. For this purpose, Cloud providers often use virtualization. For the user, the Cloud is often visible as one logical unit, although if necessary, several physical computers are processing the user's Cloud. The Cloud

[1] Whereas in the relational model the terms *attribute* or *column* are used to express schema information, for SPARQL operators the term *variable* is typically used, which we will also use in this paper.

Operator	Notation	Result
Selection	$\sigma_C(R)$	Relation containing only tuples of the relation R for which the formula C is fulfilled
Projection	$\pi_{v_1,...,v_n}(R)$	Relation containing all tuples of R, where only the variables $v_1, ..., v_n$ appear
Union	$R \cup S$	Relation with all tuples of R and S
Cross-product	$R \times S$	Relation concatenating each tuple of R with each tuple of S
Equi-Join	$R \bowtie_{v_1,...,v_n} S$	$\sigma_{R.v_1=S.v_1 \wedge ... \wedge R.v_n=S.v_n}(R \times S)$
Left Outer Join	$R \mathbin{⟕}_{v_1,...,v_n} S$	Relation containing the join result as well as those tuples from R which have not been joined
Distinct	$\delta(R)$	R without any duplicates
Order By	$\tau_{v_1,...,v_n}(R)$	R sorted according to $v_1, ..., v_n$
Limit	$limit_n(R)$	Relation with the first n tuples of R
Triple pattern scan	(i_1, i_2, i_3)	Relation containing the result of a triple pattern. This operator is special for SPARQL queries.

Table 1: Used operators of the relational algebra

user's benefits are that the user does not need to maintain any hardware resources, which are not fully used or even idle most of the time, and the Cloud user only pays what (s)he uses (*pay-by-use*). Usually the Cloud technologies scale well and are arbitrarily extendible [20].

Many free and commercial Cloud technologies are available. The Cloud service model *Platform as a Service (PaaS)* [30] offers a programming application interface or an execution environment to its users. Among the Platform as a Service frameworks, *Apache Hadoop* [3] is the most widely used de facto standard for processing big data. Apache Hadoop is open source and hence can be installed and executed on the user's own infrastructure (in a *Private* or *Community Cloud* [30]) or (s)he can buy an access to a commercial environment (in a *Public Cloud* [30] like Amazon EMR [1]). We introduce Apache Hadoop and its further relevant technologies in the following subsections.

2.3.1 Hadoop

Apache Hadoop [3] is a freely available Cloud framework which offers distributed processing of big data with high availability and a good scaling performance in a cluster of processing nodes. In contrast to other cluster systems which often require specialized hardware, Hadoop runs on a cluster of (heterogeneous) conventional PCs. High availability of the whole system is achieved by using data replication and automatic detection of errors.

Hadoop consists of two main components:

- The *Hadoop Distributed File System (HDFS)* which can be used to store and access data in a distributed and scalable fashion in the cluster.
- The *MapReduce* programming model and its framework in Hadoop which define the way to process the data stored in the cluster.

Additional extensions of Hadoop are built on top of these two main components. We use the database *HBase* and the data analysis framework *Pig* [4], which offers the high-level language *Pig Latin* to describe MapReduce jobs for data processing.

2.3.2 MapReduce Programming Model

In the MapReduce programming model, the in- and output of a user-defined map function is a set of key-value pairs. A MapReduce job consists of two phases: In the first phase the input of the map-function is mapped, such that all intermediate values with the same key are combined using an user-defined reduce function in the second phase. For this purpose, the key-value pairs may be redistributed in the cluster allowing distributed and parallel processing of both functions, the map as well as the reduce function. For example, a query processor may use the map function to map the results of triple patterns (by accessing an index) to the values of their join variables and the reduce function for finally joining these results (see Figure 2).

Several MapReduce jobs can form a MapReduce program, such that several iterations of redistributions may be necessary. Iterators are used within the processing phases supporting pipelining of intermediate results to overcome expensive materializations on external storage media like hard disks and instead process large-scale intermediate results in main memory.

2.3.3 HBase

The predecessor of HBase is Google BigTable [12]. HBase and Google BigTable share their main principles: HBase is a column-oriented database built on top of the

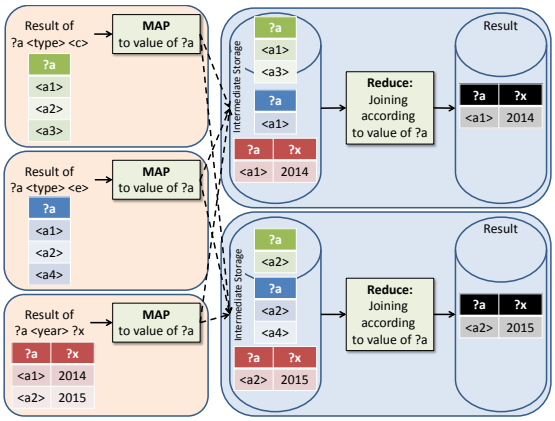

Figure 2: MapReduce example for joining the results of three triple patterns

Hadoop framework and has therefore the same advantages like high availability and scalability. Leading companies of the web era like Facebook, Twitter and eBay overcome their big data problems by running HBase.

HBase stores a table (also called multidimensional map) of key-value pairs. The key consists of several components: *row key*, *column-family*, *column-name* and *timestamp*. Values with the same column-family are stored in the same physical file, such that the distribution can be controlled for supporting local operations in order to avoid expensive network traffic.

Figure 3 presents an HBase table with two rows. One of the two rows has the row key *key1* and the other row has the row key *key2*. There are three columns for the row key *key1*: *red* (with value "#F00"), *blue* (with value "#00F") and *square* (with value "4"). As the columns *red* and *blue* are in the same column-family, both are stored in the same physical file [38].

An HBase table is not the same as a relational table. In fact an HBase table can be understood as a nested key-value-container [46] as follows:

- **Row container:** For each table row there is a container with the row key as key and the value is an own key-value container containing the columns and their values (maybe over different column families) for this row.
- **Column container:** The column name is the key of this container and its value is a value container.
- **Value container:** The value container contains a time-versioned value.

Each column family is stored in its own separate container.

The keys are internally indexed in a *Log Structured Merge Tree (LSM-Tree)* [36]. This data structure for big data is well known to be efficient, especially for frequent insertions as well as for searching.

2.3.4 Pig

Apache Pig [4] provides the high-level language *Pig Latin* to specify big data processing tasks and an execution environment to run those tasks.

The creation of conventional MapReduce tasks is often very time-consuming. First the map and reduce functions must be defined, the code compiled, the job transmitted and finally the result received.

The advantage of Pig Latin is that for the same work only a few lines of code are necessary. Furthermore, Pig offers already data operations on big data like *join*, *filter* and *sort*. Pig itself is easily extensible, e.g., user defined functions may be defined for writing, executing or reading data. The generated Pig Latin-programs are translated and executed in the Pig environment, which optimizes the execution for highly-efficient processing. The optimizations themselves are hidden from the user [49].

2.3.5 LUPOSDATE

LUPOSDATE [21] is an open source Semantic Web database which uses different types of indices for large-scale datasets (disk based) and for medium-scale datasets (memory based) as well as processing of (possibly infinite) data streams. LUPOSDATE supports the RDF query language SPARQL 1.1, the rule language RIF BLD, the ontology languages RDFS and OWL2RL, parts of geosparql and stparql, visual editing of SPARQL queries, RIF rules and RDF data, and visual representation of query execution plans (operator graphs), optimization and evaluation steps, and abstract syntax trees of parsed queries and rules. The advantages of LUPOSDATE are the easy way of extending LUPOSDATE, its flexibility in configuration (e.g. for index structures, using or not using a dictionary, ...) and it is open source [22]. These advantages make LUPOSDATE best suited for any extensions for scientific research.

2.4 Related Work of Data Management of Semantic Web Data in the Cloud

Several approaches exist in the scientific literature which address utilizing Cloud technologies for Semantic Web databases. In the following sections we will characterize these contributions according to the following design strategies:

- **Storage strategy:** Which data structure is used for storing the triples on the nodes?
- **Distribution strategy:** How are triples distributed among the nodes?

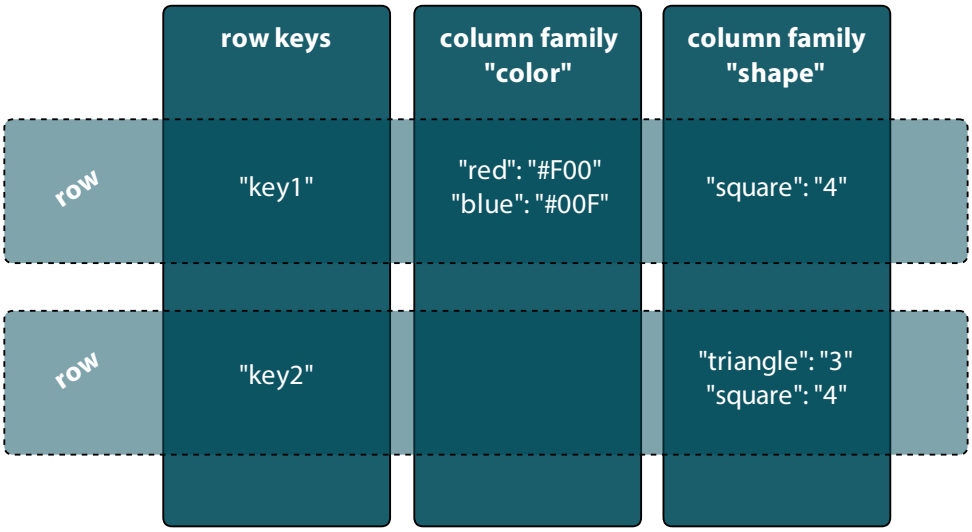

Figure 3: Example of an HBase table [38]

- **Query strategy:** How is the data queried?

2.4.1 Storage Strategy

The triples can be either stored in one or more files (e.g. using HDFS) or in a complex data structure like a database (e.g. HBase).

Storing in one file: The simplest way is to store all the triples in *one file* [40]. Hadoop and HDFS automatically take care of the distribution of the file among the different nodes. The disadvantage is that the way of distribution cannot be influenced. Also for querying the data, all the triples in this file need to be scanned to achieve the result of each triple pattern. In comparison to a centralized solution, this is very time-consuming, although scanning can be done in parallel on each node, and inefficient because the query optimizer has no possibility to restrict the number of fragments that will be scanned.

Storing in several files: Most existing work proposes to distribute the triples among *several files* [34, 25, 24, 44]. Because of the data fragmentation the result of a triple pattern can be scanned faster by addressing a fragment directly. The effectiveness depends on the *distribution strategy*, i.e., the way the triples are distributed over the different files. After a certain size each file will again be distributed over several nodes, such that the parallelization during query processing is increased for larger files.

Using a complex data structure (like HBase): Some existing work (e.g., [43, 13]) describes how to utilize HBase for storing the triples. HBase is a distributed, column-based database (see Section 2.3.3) built on top of Hadoop. The advantage of utilizing HBase is its well structured environment that is optimized for data distribution. By using column families it is possible to directly influence the data distribution to the different nodes. This is very important for the distribution strategies and their corresponding specialized query optimizations.

Centralized Semantic Web databases like *RDF-3X* [32] and *Hexastore* [48] often use six different indices according to the six different collation orders *SPO, SOP, PSO, POS, OSP* and *OPS* of RDF triples, such that the result of a triple pattern can be achieved with one index access. The result is also sorted because the indices of these centralized databases are stored in B^+-trees, such that often the fast merge joins [32] can be applied for joining the results of several triple patterns. As HBase uses *LSM-trees* [36], which have a B^+-tree similar structure (but are faster for insertions), it seems to be reasonable to choose a similar distribution strategy.

2.4.2 Distribution Strategy

The choice of the distribution strategy is one of the most important design decisions and influences the used storage space as well as, more importantly, query processing performance (with trade-off between both). We discuss existing approaches in the following paragraphs.

Horizontal partitioning: Horizontal partitioning [34] distributes the triples according to their subjects. If the subject is used as file name, this will lead to many too small partitions as there are typically a lot of subjects. Hence the result of a hash function applied to the subject determines the HDFS file, which yields a much better uniform distribution. Each resulting partition is then stored in a HDFS file. For all triple patterns with a fixed

subject (or a variable already bounded with the result of another triple pattern), it is possible to scan only the relevant file for their results instead of all triples. However, for all other cases, it is still necessary to scan all the triples, which means this approach is not efficient for many queries.

Vertical partitioning: Vertical partitioning [34, 44], which is also sometimes called *Predicate Split* [25], distributes the triples according to their predicates. As usually there are only few predicates in real-world data sets [18], it is *not* necessary to hash the predicates for computing the HDFS file in which the triple should be stored. Instead just the predicate label itself is used as the name of the HDFS storage file for this triple. Furthermore, only the subjects and objects of the triples are storage-efficiently stored, which avoids redundancy for storing the predicates. Unfortunately, the sizes of the partitions vary much as the predicates in triples do not uniformly occur, e.g., the predicate *rdf:type* occurs the most in many real-world data sets [18].

A similar problem to that in horizontal partitioning also exists for this approach: if the predicate of a triple pattern is known, then its result can be found quickly; otherwise the whole data must be scanned. In many typical application scenarios and for most triple patterns, the predicates are known [6, 31], such that this distribution strategy works well. However, not all types of queries can be answered efficiently.

Predicate Object Split/Clustered Property Partitioning: Predicate Object Split [25] is sometimes also called *Clustered Property Partitioning* [34]. This approach avoids the large partitions of the vertical partitioning by splitting and distributing the partitions further. It is assumed that those partitions of vertical partitioning are larger which contain triples with an *Internationalized Resource Identifier (IRI)* as object. Those partitions are split according to the object and distributed to smaller partitions. If the object is a literal, then just vertical partitioning is applied, as literals are mostly unique or occur seldom. In this way too small partitions are avoided. Like vertical partitioning, Predicate Object Split cannot handle all types of triple patterns efficiently. However, if predicate and object are known, a smaller partition than that for vertical partitioning is to be scanned to compute the result of the triple pattern. Storage space can be reduced up to 70% [25] by just storing the subjects in files containing the predicates and objects in their names.

Column-based Partitioning: In Column-based Partitioning [44] each triple is distributed to three different tables according to the subject, predicate and object. Instead of storing the whole triple inside, only one of them, respectively, is stored together with a triple id in the table. If the subject, predicate or object of a triple pattern is known, then the triple ids of its result can be accessed with one index scan and the triples with another index scan: At least two expensive MapReduce jobs are necessary to retrieve a triple pattern result. For most other distribution strategies only one MapReduce job is needed.

Ordered-Sorting Partitioning: In a variant of Column-based Partitioning, Ordered-Sorting Partitioning [44] stores the complete triples (instead of only triple ids) in a sorted way. In this way, only one MapReduce job is necessary to retrieve a triple pattern result, but also the storage of the triples needs much more space. Large tables are splitted into subtables, such that often only a few subtables and not the whole table must be scanned.

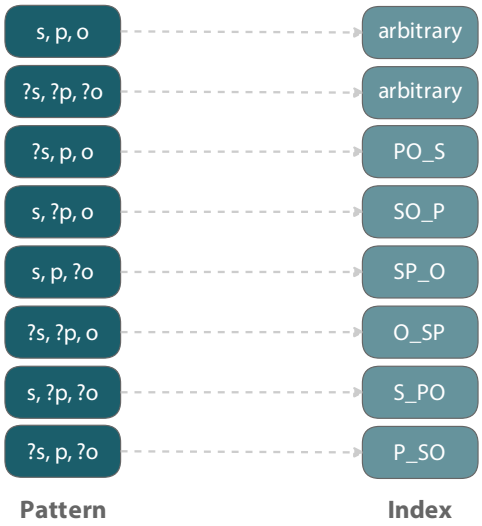

Figure 4: Triple pattern and its corresponding index when using Hexastore distribution (see [43])

Hexastore Distribution with HBase: The contribution [43] proposes a distribution strategy similar to the one of Hexastore [48] and RDF3X [32] for centralized Semantic Web databases. The triples are distributed to six different indices: *S_PO, P_SO, O_SP, SP_O, SO_P* and *PO_S*, where the name before the underscore represents the *rowkey* and that after the underscore the value. By choosing the correct index (see Figure 4), we can retrieve the result of any triple pattern within one index access.

Each triple is stored six times, which implies a much higher space consumption than the previously described approaches. According to the Cloud paradigm the Cloud can dynamically offer more space when needed, such that a better performance outweighs the disadvantage of higher space consumption.

2.5 Query Strategy

In the Hadoop environment, queries must be processed in one or several MapReduce jobs. For this purpose, the

SPARQL query must be first transformed to these jobs. If there are several triple patterns inside the query, more than one job is necessary to fulfill the task of query processing. If one job (e.g., for joining) needs the results of other jobs as an input, then these results must be temporarily stored in the distributed file system. As this causes high costs in terms of space and time, the goal of optimization is to create as few jobs as possible, such that query processing needs less data traffic and processing time.

Creation of "pure" MapReduce jobs: Several proposals (e.g., [43, 24]) describe approaches that create "pure" MapReduce jobs and store intermediate results of these jobs in HDFS, which are then used as input for further processing steps. The algorithm presented in [24] also minimizes the number of MapReduce jobs.

Creation of MapReduce jobs with the help of an abstraction language: Most of the time a lot of efforts are needed to develop "pure" MapReduce jobs, where even for simple queries often several hundred lines of code are necessary. Hence, several abstraction languages have been developed in order to indirectly create suitable MapReduce jobs.

The contribution [34] describes how triples can be stored as JSON objects. Furthermore, SPARQL queries are translated into *JAQL* [9] queries which work on JSON files. *JAQL* supports core functionalities like JOIN, FILTER and SORT which abstract from the MapReduce Framework.

Another work [40] uses the abstraction language Pig Latin from the Pig framework in order to create MapReduce jobs for tasks such as joining, filtering or processing other set operations like union. Here, the SPARQL query is first transformed into an operator graph which is afterwards optimized and finally transformed into the Pig Latin program to be executed.

2.5.1 Relation of our work to existing contributions

Our approach combines the Hexastore distribution strategy [43] for HBase with the creation of the MapReduce jobs with Pig Latin from an SPARQL operator graph (similar to the approach presented in [40]). However, we extend the existing approaches by using a variant of *bloom filter* [10]. Bloom filters are often used in databases to minimize the intermediate query results early in the query execution process [37, 33, 23]. The contributions [56] and [27] already describe the usage of bloom filter for SPARQL processing within the MapReduce framework. The approach in [56] is quite rudimentary as the bloom filter can only be created if the triples to be processed fit completely in main memory. The other approach [27] uses bloom filters only during joining two triple patterns. Furthermore, the bloom filter of a triple pattern is computed in both approaches during query processing, which is time consuming. Contrary to these approaches, our proposed variant already precomputes and stores the bloom filters for all possible triple patterns during the import phase of the triples. In this way the bloom filter of the whole query can be faster determined as additional iterations through triple pattern results are avoided during query processing. We describe more details of our approach in Section 4.

3 Proposed Architecture

We first clarify the requirements of our Cloud-based Semantic Web database. Afterwards the motivations for some of our design decisions, like addressing the storage, distribution and query strategies, are given. Finally, we describe the main phases of our approach during query processing.

3.1 Requirements

Our architecture is developed to fulfill the following requirements for a Cloud-based SPARQL processing engine:

- **Supporting full SPARQL 1.1:** The system should support the whole SPARQL 1.1.
- **Processing as many operations in the Cloud as possible:** The system should only schedule the operations that are definitely not supported by the Cloud technologies to be processed outside the Cloud.
- **Using proven Cloud-technologies:** The system should not re-invent the wheel, but use proven and efficient Cloud-based technologies as much as possible.
- **Using indices:** The system should use the best-known indices for RDF data and SPARQL query processing to achieve efficient query processing.
- **Precomputing as much as possible:** The system should precompute meta-data as much as possible, such that this information does not need to be computed for each query.
- **Supporting updates:** The system should be able to handle updates efficiently, even though the meta-data needs to be (partly) recomputed.

3.2 Storage Strategy

Our system stores the triples in HBase instead of using the distributed file system HDFS because of its advantages concerning efficient storage, fast access according to rowkeys and the possibility to influence the distribution to nodes by using column families.

Furthermore, HBase supports compression of its contained data and the compression rate of RDF data is very high due to its contained IRIs that have many long common prefixes. Although the Cloud paradigm offers the illusion of infinite storage space, in concrete systems especially in private Clouds, it may be an advantage to reduce the stored data. HBase supports different compression algorithms like *GZIP* [17], *LZO* [35] and *SNAPPY* [55]. GZIP has a good compression rate, but the compression itself is slower in comparison to LZO and SNAPPY. As we want to offer high performance in query processing, we hence choose LZO as the compression algorithm for HBase data.

3.3 Distribution Strategy

We choose the Hexastore distribution strategy (see Section 2.4.2) because the result of a triple pattern can be accessed with one index access. The result of a triple pattern is also on one node in the Cloud, such that the triple pattern result is at once available *without* an additional Reduce phase.

The disadvantage of this distribution strategy is the additional space requirements as the triples are replicated in six different indices. However, this problem is minimized by using a suitable compression algorithm. Like in [47] we choose LZO as the compression algorithm. In this way the overall space consumption (of all six indices) is reduced to 15% of the original data.

We use Bulk Load [5] for importing data. For this purpose, triples are block-wise imported using MapReduce jobs. In this way we import 10 million triples (which lead to 60 million HBase entries because of the chosen data distribution strategy) in approximately 11 minutes in our experimental setup (see Section 5). In other words, we import 1 million triples (leading to 6 million HBase entries) per minute.

3.4 Query Strategy

For full support of SPARQL 1.1, we decided to extend an existing SPARQL engine, such that parts of the query can be executed in the Cloud and the rest locally at the node initiating query processing. Because of its advantages like the easy way of extending LUPOSDATE, its flexibility in configuration and its being open source as described in Section 2.3.5, we have chosen LUPOSDATE [21] as basis for our new engine called P-LUPOSDATE.

For the query strategy we have the possibilities to directly implement MapReduce jobs or to use a higher abstraction language (see Section 2.5). While we can implement exactly what is needed using efficient algorithms when directly coding MapReduce jobs, we have less implementation effort when using an abstraction language. Abstraction languages already offer most needed functionalities and are often extendible for currently unsupported parts. Furthermore, we will directly benefit from future improvements of supported algorithms when using an abstraction language. Also temporarily storing large intermediate results in the Cloud is already supported by the use of an abstraction language, which otherwise would have to be re-implemented. Pig (see Section 2.3.4 and Section 2.5) with its abstraction language Pig Latin to express data processing jobs is proven as Cloud-based framework for big data [4]. Hence, we decided to use Apache Pig as most functionalities needed for SPARQL query processing are already supported by Pig. Actually the name P-LUPOSDATE is a combination of Pig and LUPOSDATE to reflect this design decision. Pig also supports compression of intermediate results with GZIP or LZO;[2] we choose LZO because of its fast compression and decompression. Besides saving storage space, using compression also reduces network load whenever intermediate results need to be transferred to other nodes.

All the SPARQL operators of Table 1 can be processed in the Cloud with P-LUPOSDATE. However, it is necessary to extend Pig Latin for the support of some built-in functions for the Selection operator. For example, our implementation extends Pig Latin by the *bound* built-in function, which checks whether or not a variable is bound to a value.

3.5 Phases during Query Processing

Figure 5 shows the main phases for SPARQL query processing in P-LUPOSDATE. The phases 1a and 1b are processed on the client side and phase 2 in the Cloud. First the SPARQL query is converted into a Pig Latin program (phase 1a) by transforming it into an optimized operator graph using LUPOSDATE. The Appendix A describes the transformation steps for the different operators in more detail. In phase 1b for computing the bloom filter(s) (see Section 4), the bit vectors for each triple pattern are loaded from HBase and combined according to the SPARQL operators to retrieve bloom filter(s) for the whole SPARQL query. The bloom filter(s) is/are used in phase 2 to filter out unnecessary intermedi-

[2] Pig version 0.11.1 does not support SNAPPY [2].

ate query results of each triple pattern. The final result of the whole SPARQL query is computed by executing the Pig Latin program (using one or more MapReduce jobs) in the Cloud.

4 Proposed Bloom Filter Approach

Databases often use bloom filter [10] in order to reduce the number of intermediate query results especially before expensive operations like join for speeding up these operations. Figure 6 presents an example for bloom filter application. First a bit vector (initialized with all bits cleared) for table B is generated by applying a hash function to the join attribute value of each tuple of B. The result is an index position used to set the bit in the bit vector. Afterwards this bit vector (also called bloom filter) is compared with the tuples of table A. Each tuple, for which the corresponding bit (determined by applying the same hash function as before to the join attribute value) in the bloom filter is unset, is ignored and the other tuples form a new table A'. Finally, the smaller table A' is joined with B instead of the whole table A. This concept is very interesting for distributed approaches as for a bloom filter of 1024 bits, only 128 bytes needs to be transferred.

Hence, in our system we also use bloom filters to restrict the input to the necessary minimum and avoid superfluous data transfers and computations. Typically bloom filters are computed before or during the query processing phase. This would require at least one more costly MapReduce job on large data, which can be avoided by precomputing bloom filters during importing data. Due to the simple nature of RDF consisting of only three components and because of the used Hexastore distribution strategy, we can precompute *all possible* bloom filters and store them in line with the Hexastore distribution strategy. The following Section 4.1 describes this in more detail. We will later argue (see Section 4.1.1) that even frequent updates are no problems to the system. Section 4.2 describes a hybrid storage format for the bloom filter, and Section 4.3 deals with the computation of bloom filters for the different SPARQL operators.

4.1 Precomputation of the Bloom Filters

We adapt the concept of bloom filters for Cloud-based SPARQL processing. Apache Pig already provides a bloom filter implementation. However, applying this implementation in our Cloud-based SPARQL processing engine leads to a recomputation of the bit vector for each triple pattern. It takes a long time if the triple pattern result is large, as the complete result must be iterated and one additional MapReduce phase is necessary. Hence,

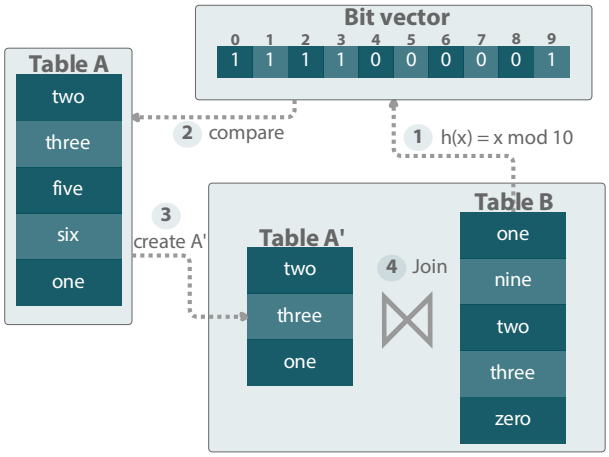

Figure 6: Bloom filter example

we developed our own bloom filter variant (and implementation), which only computes the bit vectors once during the data import. For each possible triple pattern we store the corresponding bit vector besides the precomputed triple pattern result in HBase, such that these are incorporated in line with the Hexastore distribution strategy. The bit vectors are stored in a Column-family. Figure 7 presents the final storage scheme inclusive bit vectors. The storage scheme is exemplified for the S_PO table and is similar for the other tables.

For each rowkey three column-families exist:

- **Hexa:** The values of the rowkey are stored here. For the table S_PO, this is, for example, the concatenated elements of predicates and objects separated by a comma.
- **Bloomfilter1**: This column-family stores the set bits of the bit vector of the first element. The first element is the predicate in the case of table S_PO.
- **Bloomfilter2 (optional)**: This column-family stores the set bits of the bit vector of the second element. The second element is the object in the case of table S_PO. Some tables like SP_O do not have any second element, such that for these tables only one bit vector is stored and Bloomfilter2 is empty.

By using several column-families, the triple pattern results as well as the bit vectors can be accessed separately. HBase stores three different key-value- container for each rowkey and a container for each column-family. Within the containers, the column name is stored as a key and, in our system, the corresponding value remains empty. Figure 7 marks the different elements by *Key* and *Value*.

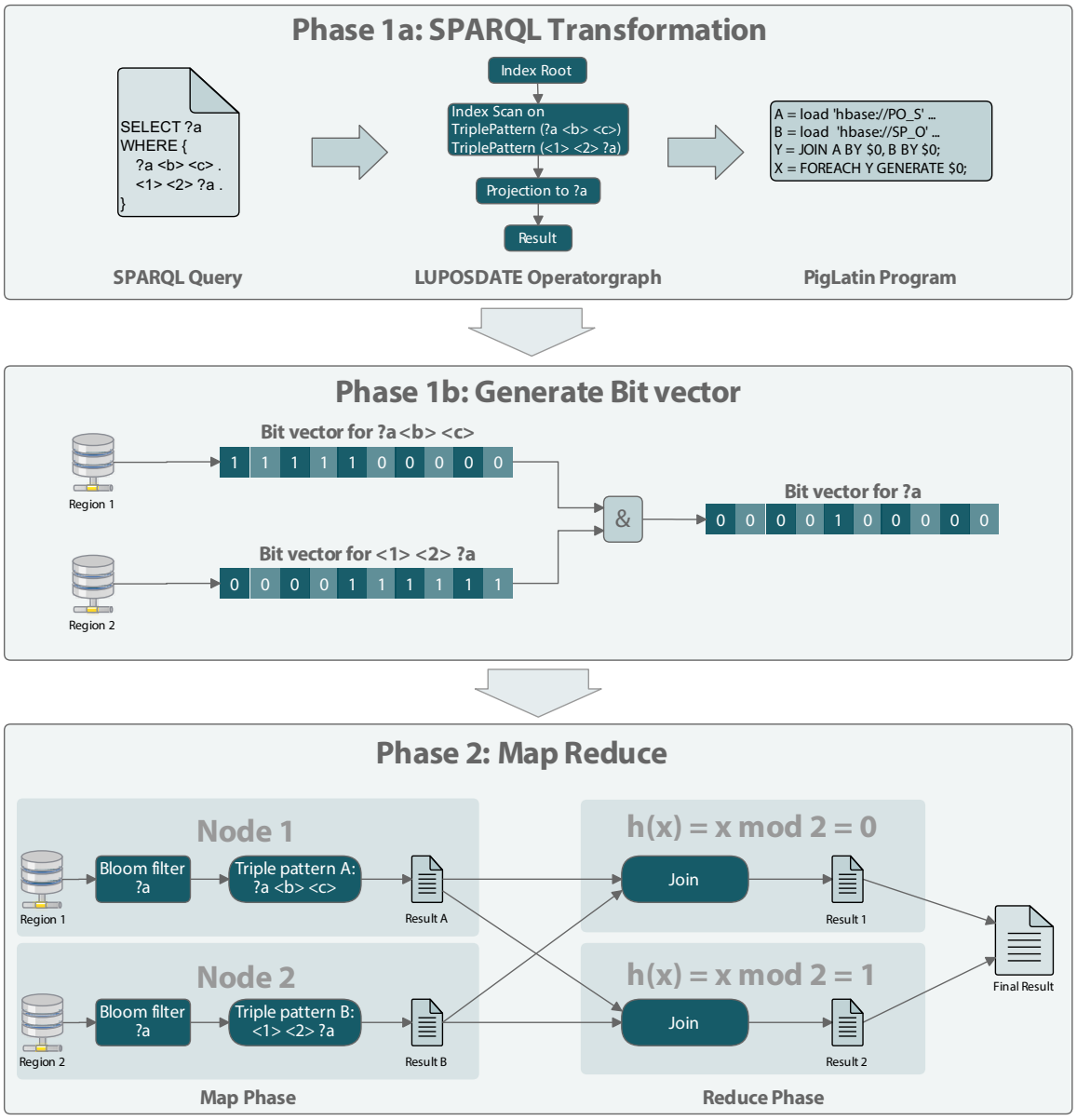

Figure 5: Phases of SPARQL Query Processing in the Cloud

4.1.1 Updates

We discuss updating the data (i.e., adding, deleting and modifying triples) and the necessary updates in the bloom filters in the following paragraphs.

Adding triples: For each added triple the corresponding bit vectors must be updated, too, by just setting the corresponding bit. This can be done quickly as for each index to which the triple is added, only one or two bit vectors need to be accessed. The system only has to ensure that the bit vectors are updated in the same transaction as the one for adding the triple; otherwise the bloom filters are incorrectly filtering out too much input.

Deleting triples: Whenever a triple is deleted from an index, it may lead to errors if we just clear the corresponding bits in the bit vectors. The corresponding bits may still need to be set because of other remaining triples in the index. We may scan all triples in the index for such a triple, but this takes similar time as building the bloom filter from scratch. Building the affected bloom filters again is, on the other hand, too inefficient for each inserted triple.

We therefore propose to delay building the bloom filters to times when the workload is low or build the bloom filters after a certain number of deletions based on the

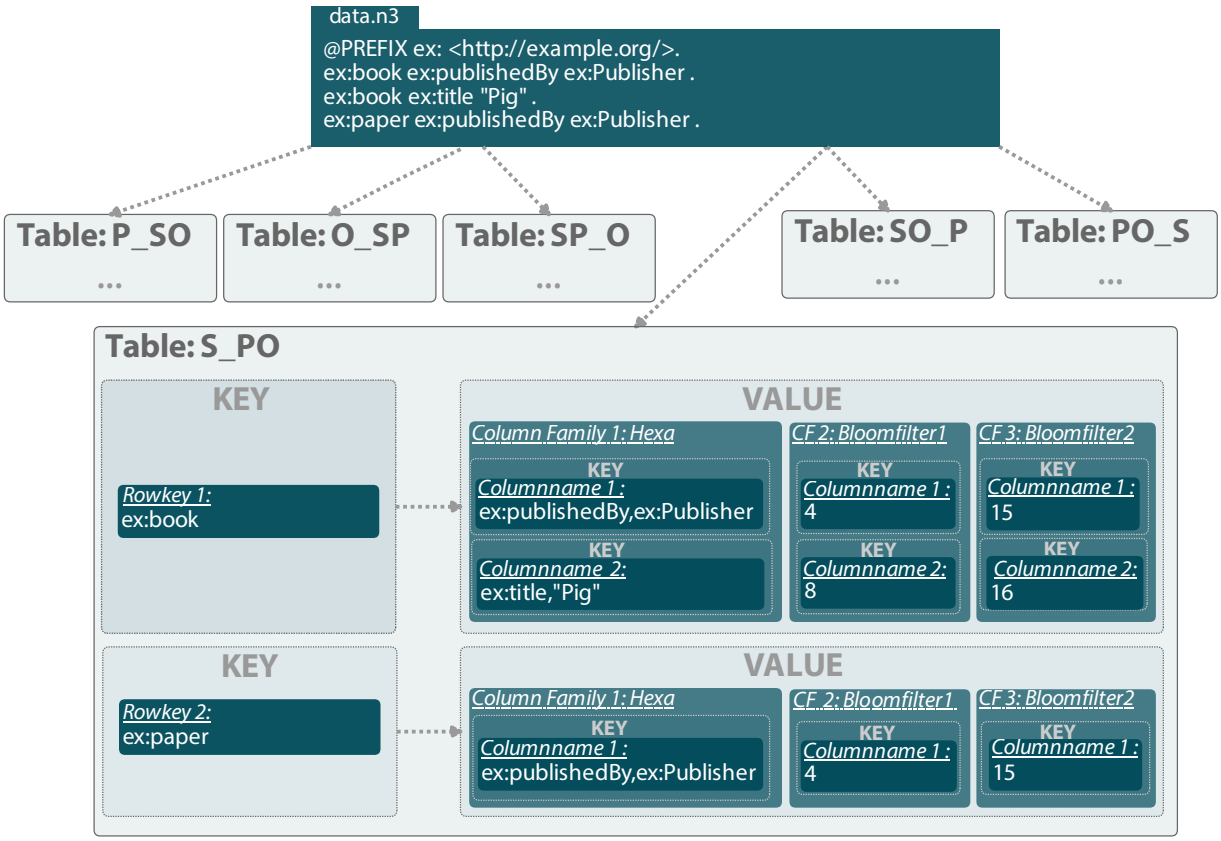

Figure 7: HBase Storage Scheme with Bloom Filter

following observation: If a few bits are unnecessarily set in a bloom filter, then the bloom filter does not filter out too much and the approach still works correctly. Only few more *false drops* may be generated, i.e., some unfiltered triples, which could be filtered out based on an up-to-date bloom filter, may lead to a higher number of unnecessary intermediate query results, which are later actually not joined and are not necessary for the final query result. An upper bound for the number of additional false drops can be obviously determined by

$$D \times \frac{\#\text{triples of data set}}{\#\text{bits of bloom filter}}$$

where D is the number of triples without bloom filter update. According to [11] the number of false drops is low for choosing the same number of bits for the bloom filter as the number of triples in the data set. Hence we expect the number of additional false drops to be at most the same as the number of deleted triples, which is manageable.

Counting bloom filters [11] which counts the occurrences of a bit position may be another solution to this problem. However, counting bloom filters needs much more storage space and thus we are not in favor for this solution.

Modifying triples: Modifying a triple can be seen as an atomic operation of deleting the old and adding the modified triple. The previously discussed methods for adding and deleting triples thus apply also here.

In summary, handling updates (adding, deleting as well as modifying triples) can be done efficiently in our system.

4.2 Storage Formats

For sparse bit vectors with only few bits set, we store the index positions of these bits. This has a big disadvantage for bit vectors with many bits set in that each index requires additional 4 bytes space as the indices are stored as integers. In fact it is no problem for HBase as the data is efficiently compressed, but the phase 1b (in Figure 5) takes too long for this kind of bit vectors. This is because in these circumstances, many indices must be loaded from HBase and transformed into a real bit vector. Experiments show that bit vectors with more than 100,000 bits set are unpractical as the calculation takes several minutes.

Hence, we searched for an alternative storage format

for bit vectors. Indeed we could save the real bit vectors instead of the indices. However, this would require overall more space for the following reasons: For the Hexastore distribution strategy, HBase stores many rows containing the results of a triple pattern, and for each of these rows, one or two bit vectors are created. In order to illustrate the problem we imported 50,027 triples and printed the number of rows (see Table 2).

Table	Number of Rows
S_PO	9,788
P_SO	57
O_SP	25,559
SP_O	48,671
SO_P	50,027
PO_S	25,850

Table 2: Number of rows for 50,027 triples

Additionally, the following properties are to be considered:

- Each table contains 50,027 triples. This means if a table contains 10,000 rows, on average 5 results are stored for each row.
- For each column name, in every row, there will be at most *one* new bit set in the bit vector. If several results have the same hash value, then the number of bits is lower than the number of results.
- For each of the tables S_PO, P_SO and O_SP, two bit vectors are stored.
- The sum of the rows is approximately 150,000.

A *real* bit vector with 10 million bits represented by bytes has a size of 1.2MB if, for each index, only 1 bit is stored. We call such a real bit vector a *byte-bit vector*. When storing the byte-bit vectors, for each row, 1.2MB is stored for SP_O, SO_P and PO_S, and 2.4MB for S_PO, P_SO and O_SP. Altogether we would have to store *(48,671 + 50,027 + 25,850)* $\times$ *1.2MB + (9,788 + 57 + 25,559)* $\times$ *2.4MB* $\approx$ **228GB** (uncompressed). If only the indices are stored as integers, then we would, in the worst case, have only *(48,671 + 50,027 + 25,850)* $\times$ *4 Byte + (9,788 + 57 + 25,559)* $\times$ *8 Byte* $\approx$ **0.75MB**. This example demonstrates that for the most rows, the storage of the byte-bit vectors is unnecessary and wastes too much storage space. We hence developed a hybrid approach which unifies the advantages of (a) storing byte-bit vectors and (b) storing the indices of set bits.

In this hybrid approach, there is an additional phase that checks each of the rows after importing the triples. If there are more than 25,000 results in a row (corresponding to the number of set bits), then additionally the byte-bit vector is generated and stored. The threshold of 25,000 is chosen because our experiments show that the generation of the byte-bit vector from the indices of set bits only needs few hundred milliseconds (and is therefore not significant in the whole query processing phase). These bit vectors with fewer than 25,000 set bits do not consume much storage space if they are stored as indices of set bits. On the other hand, the byte-bit vectors with more than 25,000 set bits can be fast accessed. Therefore overall the hybrid approach saves storage space and processing time.

4.3 Bloom Filters for a whole Query

Bloom filters are computed after a SPARQL query has been parsed, transformed to a relational expression and optimized, but before the first index scan starts. Such resulting bloom filters can be used to filter out irrelevant input.

Figure 8 provides the computation formulas for computing the bloom filters of a relational expression to which SPARQL queries are transformed. The bloom filters of each triple pattern for a specific variable can be accessed by querying the corresponding index in HBase where the precomputed bloom filters are stored. The right index is chosen based on the constants in the triple pattern and the position of the considered variable (see formula (1) in Figure 8). For example, the first bloom filter of the index SP_O with key (c_1, c_2) is chosen for a triple pattern (c_1, c_2, v) and a variable v, where c_1 and c_2 are constants. The second bloom filter of the index P_SO with key c is chosen for a triple pattern (v_1, c, v_2) and a variable v_2, where c is a constant and v_1 is a variable. See Table 3 for the used index and key for the different types of triple patterns. If we want the bloom filter of v or v_1, we choose the first bloom filter, for v_2 we choose the second bloom filter. We cannot determine a bloom filter for a triple pattern consisting only of constants. For the bloom filter for triple patterns with only variables (v_1, v_2, v_3) we have many possibility to determine the bloom filter. One possibility is to choose the first bloom filter of P_SO for v_1, the first bloom filter of S_PO for v_2 and the second bloom filter of S_PO for v_3.

Operators not changing the bloom filter: Most relational operators like projection, sorting, duplicate elimination and limit do not affect filtering and, hence, the bloom filter remains as it is (see formula (2)).

Bloom filter of selection: For many cases of the selection operator, we can compute a more precise bloom filter than just using the bloom filter of the input relation (see formula (3)). If there is a boolean AND-operation inside the filter, the conditions of its operands must be both fulfilled and, hence, we can combine the bloom filters of the operands' conditions applied to the input re-

Unary operators:

$$bf_v(tp) = index\text{-}access(tp, v) \tag{1}$$

$$bf_v(\Phi(R)) = bf_v(R) \mid \Phi \in \{\pi_{v_1,...,v_m}, \tau_{v_1,...,v_m}, \delta, limit_n\} \tag{2}$$

$$bf_v(\sigma_C(R)) = \begin{cases} bf_v(\sigma_{C_1}(R)) \wedge bf_v(\sigma_{C_2}(R)) & \text{if } C = (C_1 \wedge C_2) \\ bf_v(\sigma_{C_1}(R)) \vee bf_v(\sigma_{C_2}(R)) & \text{if } C = (C_1 \vee C_2) \\ bf_{v_1}(R) \wedge bf_{v_2}(R) & \text{if } v \in \{v_1, v_2\} \wedge C = (v_1 = v_2) \\ gen(c) & \text{if } C = (v = c) \\ bf_v(R) & \text{otherwise} \end{cases} \tag{3}$$

Binary operators:

$$bf_v(R_1 \bowtie_{v_1,...,v_m} ... \bowtie_{v_1,...,v_m} R_n) = \begin{cases} bf_v(R_1) \wedge ... \wedge bf_v(R_n) & \text{if } v \in \{v_1, ..., v_n\} \\ bf_v(R_i) & \text{if } v \notin \{v_1, ..., v_n\} \wedge v \in R_i \wedge i \in \{1, ..., n\} \end{cases} \tag{4}$$

$$bf_v(R_1 \times ... \times R_n) = bf_v(R_i) \mid v \in R_i \wedge i \in \{1, ..., n\} \tag{5}$$

$$bf_v(R_1 \bowtie_{v_1,...,v_m} R_2) = \begin{cases} bf_v(R_1) & \text{if } v \in R_1 \\ bf_v(R_2) & \text{if } v \notin R_1 \wedge v \in R_2 \end{cases} \tag{6}$$

$$bf_v(R_1 \cup ... \cup R_n) = bf_v(R_1) \vee ... \vee bf_v(R_n) \tag{7}$$

Legend:

bf_v	Function with signature $Relation \rightarrow bitvector$ to compute the bloom filter of variable v
tp	Triple pattern
$v, v_1, ..., v_m$	Variables
$index\text{-}access$	Function with signature $Triple\ Pattern \times Variable \rightarrow bitvector$ to retrieve the bloom filter of a variable for a given triple pattern from HBase
$R, R_1, ..., R_n$	Relations
$v \in R$	Variable v occurs in the relation R
C, C_1, C_2	Conditions
c	Constant (an RDF term, i.e., an iri, blank node or literal)
$gen(c)$	Function with signature $Constant \rightarrow bitvector$ returning a bit vector, where all bits are unset except the one for the constant c. The bit-position for c is determined by a hash function applied to c.

Figure 8: Computation of bloom filters based on relational expression

lation by a bitwise AND-operation. For a boolean OR-operation, one of the conditions of its operands must be fulfilled, and hence the bloom filter of the operands' conditions applied to the input relation must be combined by a bitwise OR-operation. If two variables are checked for equality, then this is an equi-join: after the selection operator only those tuples remain, which have the same values for both variables, i.e., the intersection of values for the two variables remain. Hence, the bloom filter for these two variables is the same after the selection and we can build this bloom filter by a bitwise AND-operation on the bloom filters of these two variables of the input relation. If a variable is compared with a constant value, then the resulting bloom filter contains only one bit set at the position for the constant.

Bloom filter of join: For joining operations, the bloom filters of their operands are bitwise AND-combined for join variables; otherwise the bloom filter of the input relation from which the variable originates is taken (see formula (4)).

Bloom filter of Cartesian product: There are no common variables between the input relations of Carte-

Triple pattern type	Index	Key
(c, v_1, v_2)	S_PO	c
(v_1, c, v_2)	P_SO	c
(v_1, v_2, c)	O_SP	c
(c_1, c_2, v)	SP_O	(c_1, c_2)
(c_1, v, c_2)	SO_P	(c_1, c_2)
(v, c_1, c_2)	PO_S	(c_1, c_2)
(c_1, c_2, c_3)	e.g. SP_O	(c_1, c_2) additional filtering step necessary
(v_1, v_2, v_3)	e.g. SP_O	full scan

Table 3: Used index and key for the different types of triple patterns (v and v_i are variables, c and c_i are constants)

sian products, such that also here the bloom filter of the input relation from which the variable originates is chosen (see formula (5)).

Bloom filter of Left outer join: There are two possibilities for the left outer join (see formula (6)): If the variable occurs in the left operand, then its bloom filter must be chosen as all tuples from the left operand are considered, but not additional ones from the right operand in its final result. If the variable occurs only in the right operand, then its bloom filter must be chosen as values for this variable might only come from this operand.

Bloom filter of union: For the union operation (see formula (7)), the bloom filters are bitwise OR-combined as any tuple from its operands is considered. However, many of the bits of the bloom filters may be set in this way leading to worse filtering rates. If there are no further operations like joins on the result of an union-operation, it is often better to just compute the bloom filters within the operands and apply these *local* bloom filters only within the operands. An even better way is to bitwise AND-combine the bloom filter for the whole query and the bloom filter of the considered union operand and use the result as the bloom filter for this union operand. These optimizations are not expressed in the formulas, but should be implemented in real systems.

5 Experimental Evaluation

This section presents the results of the experiments using P-LUPOSDATE[3] and their analysis. We have imported data of different sizes into the Cloud database and compare the execution times of different queries. Each query is evaluated with and without using bloom filters in order to compare both approaches.

[3]The source code of P-LUPOSDATE is open source and freely available at `https://github.com/luposdate/P-LUPOSDATE`.

5.1 SP^{2}Bench: SPARQL Performance Benchmark

We have used the SP^{2}B benchmark [42] of the University Freiburg. The advantage of this benchmark is that a generator can be used to generate different data sizes. The structure of the generated data is based on the Digital Bibliography & Library Project (DBLP) [45], such that the data and its properties are very similar to data sets of the real world. The authors of [42] also propose 14 SELECT queries and 3 ASK queries, which are developed in such a way that all important language constructs of SPARQL are covered. We do not consider the 3 ASK queries in our experiments as they do only differ from some of the SELECT queries by being an ASK query and not a SELECT query.

All queries are executed completely in the Cloud except of query 11 which contains an OFFSET-clause (`OFFSET 50`), which is not supported by Pig Latin. Hence, the OFFSET operator is executed at the master node after all the other computations have been done in the Cloud. As only 50 intermediate results are additionally transmitted to the master node, this does not significantly influence the total query execution time.

The original query 6 in this benchmark contains an OPTIONAL clause with no common variables with the outer triple patterns. We have to specify the join variables for left outer joins in Pig Latin. Another work [39] solved this problem by modifying the query slightly, such that executing the modified query does neither change the result nor the execution order. Hence, we use the same modified version (Listing 3) of the original query 6.[4]

5.1.1 Experimental Environment

We have used Cloudera CDH 4.4.0 [14] for the installation of the Hadoop (version 2.0.0-mr1) and the HBase (version 0.94.6) cluster.

The hardware of the cluster consists of 8 different nodes with the following configuration:

- **1× Master Node:** Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz, 12GB DDR3 1333 MHz RAM, 1TB hard disk and 1GBit network connection.
- **5× Worker Nodes:** Intel(R) Core(TM)2 Quad CPU Q9400 @ 2.66GHz 4GB DDR3 1333 MHz RAM, 250GB to 500GB hard disk and 1GBit network connection.
- **2× Worker Nodes:** Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz, 4GB DDR3 1333 MHz RAM, 300GB

[4] In future releases we will solve this problem by implementing variable propagation (considering the filter condition `?author=?author2`) in the logical optimization phase.

```
...
SELECT ?yr ?name ?document
WHERE {
 ?class rdfs:subClassOf foaf:Document.
 ?document rdf:type ?class.
 ?document dcterms:issued ?yr.
 ?document dc:creator ?author.
 ?author foaf:name ?name.
 OPTIONAL {
   ?class2 rdfs:subClassOf foaf:Document.
   ?document2 rdf:type ?class2.
   ?document2 dcterms:issued ?yr2.
   ?document2 dc:creator ?author.
   FILTER (?yr>?yr2)
 }
 FILTER (!bound(?document2))
}
```

Listing 3: SP²B Query 6 (Modified)

and 400GB hard disk and 1GBit network connection.

Hence, we run 1 Namenode and 7 Datanodes for HDFS, 1 JobTracker and 7 TaskTracker for MapReduce as well as 1 HBase Master and 7 RegionServer for HBase. We use the standard configuration with few modifications: we change the replication factor of HDFS from 3 to 1 in order to save space. We choose a heap size of 2GB for the MapReduce-TaskTracker-Service and 1GB for the MapReduce Child process. The number of MapReduce slots corresponds to half of the number of CPU cores in the worker nodes, i.e., 12 MapReduce slots in our cluster. We choose one Reduce slot for each worker node, i.e., our cluster contains 7 Reduce slots.

5.1.2 Benchmark Results and Analysis

The results of the experiments and their analysis are presented in the following sections. We first discuss the import of the data and its space requirements, then the experiments for query evaluation, the scalability of query evaluation as well as the determination of the bloom filters.

5.1.2.1 Import and Byte-Bit Vector Calculation

We use Bulk Load [5] for importing the data. The times and times per 1 million triples $\text{Ø}_{\text{per 1m}}$ for importing the data in the sizes of 1 million, 10 million, 100 million and 1000 million triples and the corresponding byte-bit vector calculation are presented in Table 4.

The times per 1 million triples $\text{Ø}_{\text{per 1m}}$ are nearly constant for 10 million triples and larger datasets. Note

# Triples	Import		Bit Vector-Calculation	
	Time	$\text{Ø}_{\text{per 1m}}$	Time	$\text{Ø}_{\text{per 1m}}$
1 m	00:01:40	00:01:40	00:00:44	00:00:44
10 m	00:12:34	00:01:15	00:02:42	00:00:16
100 m	02:04:05	00:01:14	00:28:40	00:00:17
1,000 m	21:32:40	00:01:18	04:19:20	00:00:16

Table 4: Times for import and byte-bit vector calculation in the format [hours:minutes:seconds]

that the same numbers apply for subsequent updates because the way of adding triples is the same for importing data from scratch and for adding additional triples. Overall, the experiments show that the times to import and update data are almost proportional to the number of triples and are therefore well scalable.

5.1.2.2 Space Requirements

The data in the HBase tables are compressed before being stored on hard disk. We present the storage space for the different indices and their sum in Table 5.

Index \ # Triples	1m	10m	100m	1000m
S_PO	0.041	0.417	4.1	41.4
SP_O	0.041	0.421	4.1	41.0
SO_P	0.048	0.517	5.0	49.6
P_SO	0.031	0.306	3.6	39.6
PO_S	0.047	0.439	4.5	46.6
O_SP	0.059	0.548	5.6	56.6
Sum	0.267	2.590	27.0	274.9
Original Data	0.107	1.100	11.0	103.0

Table 5: Space of HBase tables in comparison to original data in GB

The disadvantage of storing the triples six times (in the six indices) is quite well compensated by compression. Furthermore, the space for the six indices is almost proportional to the number of triples. Table 5 contains also the storage space for the original data in N3 format. Contrary to the storage format used in our implementation for the HBase data, the N3 data makes extensive use of prefixes which minimizes the sizes drastically.

5.1.2.3 Query Evaluation

In order to show the benefits of the bloom filter approach, we import 1 billion triples and compare the execution times of query evaluation with and without using the bloom filter approach. Table 6 and Figure 9 present the

experimental results. In the experiments, we use a bit vector size of 1 billion in order to minimize the number of false drops. We run each experiment three times. Here, in Table 6 we present the average execution times and the uncorrected sample standard deviation σ relative to the absolute execution times in percent in order to show that the execution times do not differ much.

Table 6 contains also the speedup of each query when using the bloom filter approach. Hence, the speedup is computed in the following way:

$$Speedup = \frac{Execution\ Time\ without\ Bloom\ Filter}{Execution\ Time\ with\ Bloom\ Filter}$$

This means that a speedup of 2 indicates that the bloom filter approach needs half of the time in comparison to not using the bloom filter approach.

Query	With Bloom Filter		Without Bloom Filter		Speedup	Number of Query Results
	Time	σ[%]	Time	σ[%]		
Q1	00:10:03	2.97	00:14:45	3.61	1.47	1
Q2	01:21:38	0.91	01:45:28	2.21	1.29	95,476,064
Q3a	00:12:54	1.37	00:19:14	0.62	1.49	10,157,284
Q3b	00:01:53	1.89	00:02:37	1.55	1.39	70,612
Q3c	00:01:51	0.65	00:02:36	1.53	1.40	0
Q4	04:49:10	1.64	05:20:57	1.52	1.11	4,509,222,768
Q5a	01:38:30	1.57	01:42:08	1.26	1.04	9,197,169
Q5b	01:37:46	1.76	02:41:22	0.56	1.65	9,197,169
Q6	03:32:24	0.76	03:32:45	0.83	1.00	120,403,318
Q7	03:46:51	2.71	04:05:17	0.81	1.08	20,440
Q8	04:23:34	3.29	09:50:13	2.67	2.24	493
Q9	02:20:41	0.39	02:30:12	0.55	1.07	4
Q10	00:00:19	1.15	00:00:19	0.10	1.00	656
Q11	00:13:00	0.91	00:13:11	1.01	1.01	10
Q1-Q11	24:10:35	0.46	32:21:03	0.89	1.34	-

Table 6: Results for SP^{2}B benchmark for 1 billion triples (times in [hours:minutes:seconds])

Considering Table 6, nearly all queries are running faster when using the bloom filter approach. We will discuss the results for each query in the following paragraphs in more detail:

Q1: The first query returns every time exactly one result, independent of the size of the input data. The query contains three triple patterns, the results of which are joined over one common variable. The result of this query can be determined by only one MapReduce job by using a multi-join. Due to the bloom filter, unneeded triples can be filtered out during the map phase, such that the number of intermediate results can be reduced. This can be also seen in the log files: Only 3 intermediate results with a size of 233 bytes are transmitted after the map phase to the reduce nodes when using bloom filters, but 62,088,782 intermediate results with a size of 8GB are transmitted without bloom filters, which explains the high speedup of 1.47 for query 1.

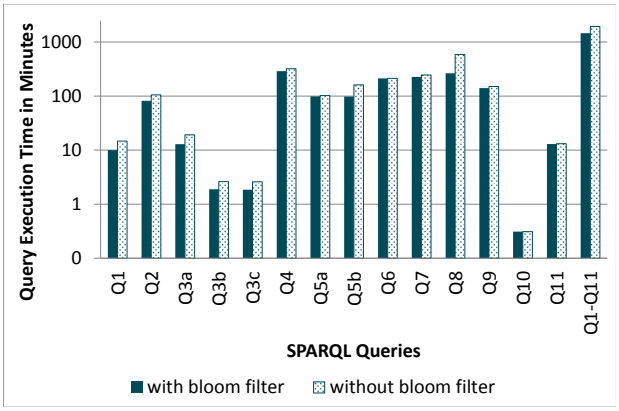

Figure 9: SP^{2}B benchmark results for 1 billion triples

Q2: Query 2 contains an OPTIONAL clause with one triple pattern. Outside the OPTIONAL clause 9 different triple patterns are joined over one common variable. Additionally, the result should be sorted. Hence, 4 MapReduce jobs are necessary: The first job loads the results of the 9 triple patterns and joins them. The bloom filter approach transmits for this purpose 351,778,497 intermediate results (47GB) to the Reduce-nodes, whereas without bloom filters 721,912,971 intermediate results with a size of 90GB are transmitted. The second MapReduce job loads the triple pattern of the OPTIONAL clause and builds the left outer join with the result of the first job. Finally, in the third and fourth jobs the data is sorted and the projection is computed. These huge intermediate and final results lead to long running queries. The obtained speedup of 1.29 shows significant performance improvements also for this kind of queries.

Q3 (a,b,c): The third query has three variants. Each variant has two triple patterns, which are joined over one common variable. The variants differ only in the predicate of the second triple pattern (after constant propagation in the logical optimization phase). Only one MapReduce job was necessary for all query variants. For query 3a with bloom filter 21,015,573 intermediate results (2GB) and without 78,672,889 intermediate results (9,1GB) are transmitted to the Reduce nodes. For query 3b with bloom filter 141,965 intermediate results (13MB) and without 11,039,357 (780MB), and for query 3c with bloom filter 19,039 intermediate results (1.8MB) and without 11,825,000 (881MB). For all variants, the speedup is between 1.39 and 1.49.

Q4: For the fourth query, 8 triple patterns are loaded and joined. Contrary to the previous queries, not all

triple patterns can be joined over one common variable. Hence, several MapReduce jobs are necessary for joining. The triple patterns must be joined in 5 MapReduce jobs and one additional MapReduce job is necessary for a succeeding duplicate elimination. Table 7 presents the jobs and their resultant intermediate results as well as their sizes. In the first two jobs the data size is reduced from 25.1GB to 5.58GB. The total execution time is reduced by 9.9%. This query contains a Cartesian product over two bags with the same values (except of variable names). Hence, the same data sizes are loaded in the jobs 1 and 2, and also in jobs 3 and 4. Jobs 3 and 4 dominate the query evaluation time, their intermediate results cannot be reduced much by our bloom filter approach. However, the reduction of intermediate results in jobs 1 and 2 to 23% still leads to a speedup of 1.11. The final result set after duplicate elimination contains 4,509,222,768 results. This shows that our system can handle big data during query evaluation.

Job	With Bloom Filter		Without Bloom Filter	
	#Intermediate Results	Size [GB]	#Intermediate Results	Size [GB]
1	58,863,644	5.58	251,411,636	25.10
2	58,863,644	5.58	251,411,636	25.10
3	150,516,077	20.25	151,468,117	20.34
4	150,516,077	20.25	151,468,117	20.34
5	69,480,176	8.43	69,480,176	8.43
6	4,513,878,642	601.75	4,513,878,642	601.75

Table 7: Space requirements after the map function for SP^{2}B query Q4

Note that the database tables contain the sizes of the uncompressed data, but we have used SNAPPY for compressing the intermediate query results during transmission, which reduces the space to 15-50% of the uncompressed data.

Q5a: In query 5a, six different triple patterns are joined. Here, 5 MapReduce jobs are necessary for these joins as well as 1 MapReduce job for duplicate elimination. Table 8 contains the properties of these jobs. Except of job 1, the data sizes are not significantly reduced when using bloom filters. The first two jobs are executed in parallel and are the input of the third job, which must wait for the results of both jobs. Hence, the reduction of the data size in job 1 does not improve the overall execution time. We obtain an improvement of only about 3.55%.

Q5b: The result of query 5b is the same as that of query 5a. However, query 5b contains only 5 triple patterns, which must be joined in 3 MapReduce jobs followed by an additional MapReduce job for duplicate elimination. Table 9 lists the number of intermediate query results and their sizes. Especially in the second job the bloom filter reduces the intermediate results much from 1.7 billion to 70 million triples, such that the total execution time is reduced to 39.41% of the time without bloom filters.

Job	With Bloom Filter		Without Bloom Filter	
	#Intermediate Results	Size [GB]	#Intermediate Results	Size [GB]
1	47,895,418	4.22	289,724,174	27.66
2	255,333,923	24.46	240,443,060	23.74
3	231,551,928	20.94	233,455,696	21.20
4	150,516,282	18.79	151,468,329	18.89
5	227,627,971	37.99	227,629,074	37.99
6	39,206,794	3.66	39,206,794	3.66

Table 8: Space requirements after the map function for SP^{2}B query Q5a

Job	With Bloom Filter		Without Bloom Filter	
	#Intermediate Results	Size [GB]	#Intermediate Results	Size [GB]
1	347,787,102	33.72	575,676,818	56.52
2	69,761,060	14.47	1,681,136,113	397.04
3	101,567,720	11.47	134,603,979	16.67
4	39,206,794	3.66	39,206,794	3.66

Table 9: Space requirements after the map function for SP^{2}B query Q5b

Q6: Query 6 contains 5 triple patterns and additionally 4 triple patterns in an OPTIONAL clause, such that overall 10 MapReduce jobs are created. Table 10 contains the intermediate query results of the jobs. The jobs 1 to 3, 4 to 6, and 7 and 8 are executed in parallel. Except for the jobs 1 to 3, the bloom filter does not significantly reduce the number of intermediate results. The first three jobs without bloom filter additionally store about 12GB (uncompressed) for the intermediate results. A further analysis shows that the transmission of these additional 12GB takes only about one minute longer. Hence, overall the execution times when using and when not using bloom filters do not significantly change.

Q7: Query 7 requires duplicate elimination and contains 2 nested OPTIONAL clauses each of which with 4 triple patterns, 5 triple patterns outside the OPTIONAL clause and two FILTER expressions. Altogether 12 MapReduce jobs (see Table 11) run for this query in the Cloud. The bloom filter approach filters out quite many intermediate results for some of these jobs. For example, the jobs 2 and 3 transmit only 92.7MB instead of 16.19GB to the Reduce nodes. Unfortunately, the first 5 jobs are executed in parallel and are the input of the

Job	With Bloom Filter		Without Bloom Filter	
	#Intermediate Results	Size [GB]	#Intermediate Results	Size [GB]
1	319,350,924	35.41	481,737,742	47.00
2	319,350,924	35.41	481,737,742	47.00
3	319,350,924	35.41	481,737,742	47.00
4	194,726,834	38.78	194,727,925	38.78
5	194,726,834	38.78	194,727,925	38.78
6	194,726,834	38.78	194,727,925	38.78
7	310,502,789	49.25	311,455,764	49.34
8	310,502,789	49.25	311,455,764	49.34
9	389,453,511	89.99	389,454,602	89.99
10	293,677,229	97.49	293,677,229	97.49

Table 10: Space requirements after the map function for SP^{2}B query Q6

further jobs. As the bloom filter for the 5th job does not reduce the transmitted data, the reduction because of the bloom filters in the first 4 jobs does not have any effect on the query execution time (except for the overall workload reduction). In the end, a reduction of 7.52% of the execution time can be observed when using bloom filters.

Job	With Bloom Filter		Without Bloom Filter	
	#Intermediate Results	Size [GB]	#Intermediate Results	Size [GB]
1	327,998,498	59.97	1,263,226,051	236.57
2	724,774	0.09	190,317,759	16.19
3	724,774	0.09	190,317,759	16.19
4	190,319,033	32.82	1,000,667,828	218.39
5	1,000,667,822	218.39	1,000,667,828	218.39
6	13,506,566	3,65	85,697,170	12.57
7	13,506,566	3,65	85,697,170	12.57
8	67,304,597	21,63	67,304,597	21.63
9	67,979,274	23,09	67,979,274	23.09
10	25,563,556	7,23	25,563,556	7.23
11	20,776,021	6,03	20,776,021	6.03
12	25,513	0.002	25,513	0.002

Table 11: Space requirements after the map function for SP^{2}B query Q7

Q8: In query 8 two triple patterns are joined with the union of 5 and 3 triple patterns containing also filter expressions causing altogether 9 MapReduce jobs. The achieved speedup of 2.24 is the largest for this query. Applying bloom filters reduces the data size e.g. in job 8 from 1.85TB to 119.7GB (see Table 12), which are only about 6% of the data without bloom filters.

Q9: Query 9 requires duplicate elimination and consists of a union, each operand of which contains 2 triple patterns. Hence, 3 MapReduce jobs are necessary. Applying bloom filters reduces the sizes of the intermediate results in job 2 from 150.59 GB to 50.31GB (see Table 13). A speedup of 1.07 is achieved.

Job	With Bloom Filter		Without Bloom Filter	
	#Inter-mediate Results	Size [GB]	#Inter-mediate Results	Size [GB]
1	2	88 Bytes	116,727,855	3.11
2	458,947,675	45.96	458,948,970	45.96
3	458,947,675	45.96	458,948,970	45.96
4	229,475,055	22.98	458,948,970	45.96
5	753,023,885	136.34	753,979,316	136.43
6	753,023,885	89.47	753,979,316	89.56
7	637,248,267	119.69	1,274,498,947	245.57
8	637,251,403	119.7	9,839,070,935	1847.41
9	4,711	0.0003	4,711	0.0003

Table 12: Space requirements after the map function for SP^{2}B query Q8

Job	With Bloom Filter		Without Bloom Filter	
	#Intermediate Results	Size [GB]	#Intermediate Results	Size [GB]
1	1,116,720,996	168.92	1,116,720,996	168.92
2	533,546,529	50.31	1,116,720,996	151.59
3	464,993,168	21.54	464,993,168	21.54

Table 13: Space requirements after the map function for SP^{2}B query Q9

Q10 and Q11: These two queries are not relevant for the bloom filter approach as each contains only one triple pattern, such that no join is executed and hence the bloom filter does not filter out any intermediate result. Based on the 6 indices, the Cloud database can quickly find the results of the triple patterns. For example, for query 10 and its 656 results, our Cloud database needs only 19 seconds.

5.1.2.4 Scalability for Big Data

Besides the dataset of 1 billion triples, also other data set sizes are imported into the Cloud database in order to investigate on their impacts on the bloom filter approach. We present the results in the following paragraphs.

1 million triples: The experiments with 1 million triples are repeated 5 times. The size of the bit vector is 100 million. Table 14 summarizes the results for this data set. Except for query 8, there is no significant difference, such that the overhead for the calculation and application of the bloom filters outrun the runtime improvements for reducing the intermediate results for rel-

atively *small* data sets. However, there is still a speedup of 1.02 when running all queries.

Query	With Bloom Filter		Without Bloom Filter		Speedup	Number of Query Results
	Time	σ[%]	Time	σ[%]		
Q1	00:00:18	12.58	00:00:18	9.79	0.98	1
Q2	00:01:47	5.29	00:01:48	0.24	1.01	32,764
Q3a	00:00:34	7.28	00:00:33	5.30	0.98	52,676
Q3b	00:00:30	1.99	00:00:28	6.93	0.94	379
Q3c	00:00:27	10.47	00:00:29	0.18	1.05	0
Q4	00:03:00	2.65	00:03:03	1.32	1.01	2,586,645
Q5a	00:02:14	3.36	00:02:14	1.48	1.01	35,240
Q5b	00:02:05	0.69	00:02:06	1.84	1.01	35,240
Q6	00:03:38	1.56	00:03:38	3.17	1.00	62,790
Q7	00:03:48	2.14	00:03:54	0.82	1.03	292
Q8	00:03:09	2.74	00:03:24	3.60	1.08	400
Q9	00:01:34	3.94	00:01:35	0.23	1.01	4
Q10	00:00:18	12.58	00:00:18	9.79	0.98	572
Q11	00:01:21	4.11	00:01:22	4.83	1.02	10
Q1-Q11	00:24:43	0.83	00:25:10	0.79	1.02	-

Table 14: SP^{2}B benchmark results for 1 million triples (times in [hours:minutes:seconds])

10 million triples: The 10 million experiments are executed 5 times with a bit vector size of 100 million bits. According to Table 15 the bloom filter approach saves execution time of already over 5% for the queries Q1, Q3 (a,b,c), Q5b, Q8 and Q9. The query 8 is already speeded up much (1.32) for this data size.

Query	With Bloom Filter		Without Bloom Filter		Speedup	Number of Query Results
	Time	σ[%]	Time	σ[%]		
Q1	00:00:35	2.20	00:00:37	6.60	1.13	1
Q2	00:02:52	1.73	00:02:57	1.16	1.03	613,683
Q3a	00:00:41	4.86	00:00:44	0.47	1.08	323,452
Q3b	00:00:29	2.74	00:00:33	5.98	1.14	2,209
Q3c	00:00:30	2.12	00:00:33	6.08	1.14	0
Q4	00:06:00	1.54	00:06:03	1.15	1.00	40,085,208
Q5a	00:03:13	3.43	00:03:13	1.79	1.00	404,896
Q5b	00:03:04	2.26	00:03:25	2.37	1.08	404,896
Q6	00:06:01	3.05	00:05:58	1.60	1.01	852,624
Q7	00:10:06	2.34	00:10:30	2.78	1.07	2,336
Q8	00:05:42	2.37	00:07:26	0.85	1.32	493
Q9	00:07:04	3.33	00:07:27	2.78	1.04	4
Q10	00:00:19	2.88	00:00:19	0.46	0.98	656
Q11	00:01:34	6.12	00:01:34	5.76	1.02	10
Q1-Q11	00:48:10	1.22	00:51:19	0.75	1.07	-

Table 15: SP^{2}B benchmark results for 10 million triples (times in [hours:minutes:seconds])

100 million triples: For 100 million triples we run the experiments 3 times with a bit vector size of 100 million bits. Table 16 shows that the improvements by the bloom filter approach are high. For the queries Q1, Q3 (a,b,c), Q5b and Q8 we save an enormous amount of time of up to 53%.

Query	With Bloom Filter		Without Bloom Filter		Speedup	Number of Query Results
	Time	σ[%]	Time	σ[%]		
Q1	00:02:11	4.69	00:02:40	4.53	1.22	1
Q2	00:11:20	12.6	00:11:26	4.39	1.01	9,049,975
Q3a	00:01:57	2.56	00:02:27	1.59	1.26	1,466,388
Q3b	00:00:47	5.22	00:00:52	4.53	1.11	10,143
Q3c	00:00:45	1.75	00:00:54	0.48	1.21	0
Q4	00:31:08	0.54	00:32:44	0.32	1.05	460,116,279
Q5a	00:10:07	1.54	00:10:22	0.36	1.02	2,016,938
Q5b	00:10:44	0.61	00:15:33	0.66	1.45	2,016,938
Q6	00:24:30	4.46	00:24:03	0.90	0.98	9,811,861
Q7	00:30:16	0.85	00:32:02	3.11	1.06	14,645
Q8	00:24:56	2.20	00:53:48	10.49	2.16	493
Q9	00:19:44	4.45	00:22:13	4.00	1.13	4
Q10	00:00:19	3.72	00:00:19	0.14	0.96	656
Q11	00:03:05	0.80	00:02:56	0.37	0.95	10
Q1-Q11	02:51:48	0.85	03:32:19	2.56	1.24	-

Table 16: SP^{2}B benchmark results for 100 million triples (times in [hours:minutes:seconds])

Figure 10 contains a comparison of the total execution times of all queries altogether for the different data sizes. The increase of the execution times is slightly more than proportional to the number of imported triples for both approaches (with and without bloom filter). However, the experiments still show that the Cloud database is able to handle large data sets in a scalable way. Especially the improvements when applying the bloom filters increase remarkably for larger data sets.

5.1.2.5 Determination of the bloom filters

Finally, we present the calculation times for determining the bloom filters for the different queries. Table 17 contains these calculation times for each query dependent on the data size. For the data set sizes 1, 10 and 100 million triples, we use a bit vector size of 100 million bits, which can be stored in 12MB. For the data set size 1 billion triples, the bit vector size is 1 billion bits, which has a size of 120MB. The experimental results show that the time overhead incurred in this phase of query processing is not significant compared to those incurred in the other phases. For the data set sizes of up to 100 million, the bloom filter calculation can be done for most queries in under 1 second, and for the 1 billion triples data set in

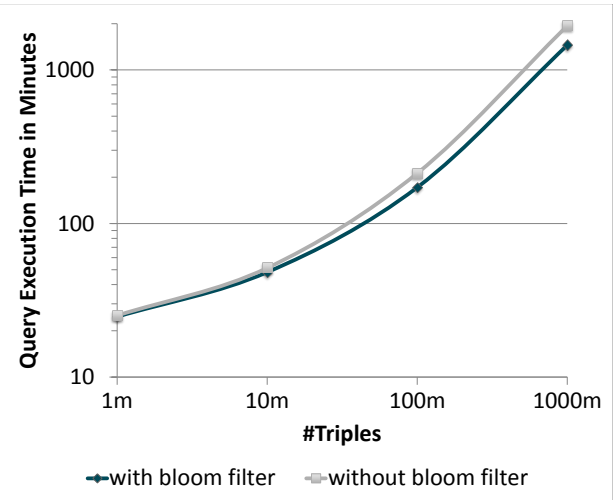

Figure 10: Comparison of the total execution times

only a few seconds. If the calculation needs longer time, like for query 8 with 70 seconds, then it does not significantly extend the total execution time, which is over 4 hours for query 8, and the achieved speed up of 2.24 is quite high. Queries 10 and 11 contain only 1 triple pattern; thus the bloom filter calculation does not need to be done.

# Triples / Query	1 m	10 m	100 m	1,000 m
Q1	0.04s	0.03s	0.05s	0.04s
Q2	0.72s	1.07s	3.57s	22.57s
Q3a	0.30s	0.37s	0.69s	4.40s
Q3b	0.28s	0.32s	0.59s	1.90s
Q3c	0.23s	0.36s	0.43s	1.92s
Q4	1.06s	1.53s	2.96s	26.39s
Q5a	0.98s	1.55s	3.27s	37.41s
Q5b	0.72s	1.10s	2.07s	23.00s
Q6	2.04s	2.96s	6.80s	61.63s
Q7	1.38s	2.07s	2.50s	21.66s
Q8	1.85s	2.98s	6.16s	69.91s
Q9	0.40s	0.61s	1.17s	9.63s
Q10	-	-	-	-
Q11	-	-	-	-
Q1-Q11	10s	14.96s	30.25s	280.45s

Table 17: Bloom filter calculation time before executing the SPARQL query

Figure 11 shows the bloom filter calculation times of all queries together dependent on the data set sizes. Again, the increase is slightly more than proportional to the number of imported triples. This is caused by the different sizes of the bloom filters for the different data set sizes. For the first 3 data set sizes, the bloom filter size is 100 million bits, and thus the calculation times do not differ much. For the 1 billion triples data set size we need a bloom filter size of 1 billion bits, as otherwise much worse (and hence useless) filter effects occur. As expected, the effort for calculating the bloom filter increases by a factor of about 10 from 30.25 seconds to 280.45 seconds.

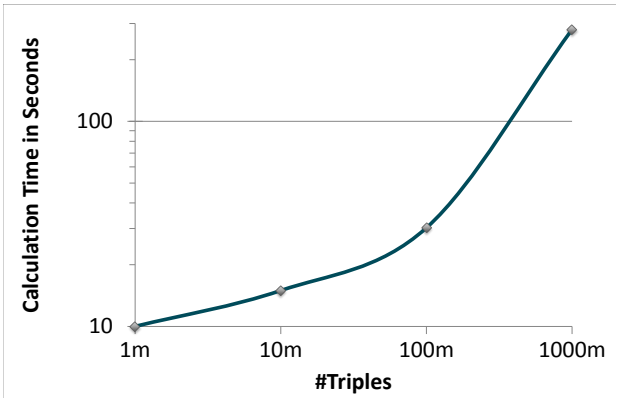

Figure 11: Comparison of the bloom filter calculation times

5.2 Real-world data: D2R Server publishing the DBLP Bibliography Database

For real-world data we choose the content[5] of the D2R Server publishing the DBLP Bibliography Database [26]. The data set is close to the one of SP2B, the results of which we discussed in the previous section, as SP2B imitates DBLP data by generating synthetic data, which has the advantage of generating data of different sizes. However, this time the used data set contains the data of a recent extraction from the DBLP Bibliography Database [45] containing a huge collection of scientific articles not only from the database and logic programming community. Furthermore, we compare our results with a local installation of a Semantic Web database as well as with the traditional bloom filter approach.

5.2.1 Experimental Environment

This time all experiments are conducted in a cluster of 8 nodes. All nodes are running Ubuntu 12.04.5 LTS. The master node uses an Intel(R) Core(TM) i5 CPU 750@2.67GHz with 16 Gigabytes main memory. All other nodes use an Intel(R) Core(TM) 2 Duo CPU E6550@2.33GHz with 4 Gigabytes main memory. HBase is configured to run 1 master and 8 region servers.

[5] available at `http://dblp.l3s.de/dblp.rdf.gz`

HDFS runs 8 data nodes, 1 name node, 1 secondary name node and 1 balancer. Join and duplicate elimination are done in parallel by 5 nodes.

Furthermore, we compare the results with a local installation of LUPOSDATE [22]. The test system for the performance analysis of the local installation uses an Intel Xeon X5550 2 Quad CPU computer, each with 2.66 Gigahertz, 72 Gigabytes main memory (from which we reserved 10 Gigabytes for the Java heap during query processing), Windows 7 (64 bit) and Java 1.7. We have used a 500 GBytes SSD for administrating the indices.

5.2.2 Benchmark Results and Analysis

In this section, we describe our results concerning the import of data, query evaluation as well as the determination of bloom filters.

5.2.2.1 Import and Space Requirements

The number of triples contained in the D2R DBLP extraction data set is 113,175,668. After duplicated triples have been eliminated, 70,929,219 triples remain.

Importing the data into the cloud took 2 hours, 49 minutes and 30 seconds. The space requirements are summarized in Table 18. This time we have chosen GZIP to compress the HBase table content, which achieves a better compression. GZIP is slower in compression, but is faster in decompression. Such configurations using asymmetric compression schemes are good for the case where data are not frequently updated having the benefit of saving more space.

# Triples / Index	approx. 100m
S_PO	1.623
SP_O	1.757
SO_P	1.925
P_SO	1.420
PO_S	1.934
O_SP	1.855
Sum	10.516
Original Data	uncompressed: 17.660, compressed (gz): 0.722

Table 18: Space of HBase tables for the D2R DBLP extraction in comparison to original data in GB

The local installation took only 28 minutes and 22 seconds for importing the data, but we reserved 32 GBytes of the main memory for this purpose such that not much data needed to be swapped during the import phase. The local installation uses a dictionary for coding RDF terms during query evaluation. The size of the dictionary on disk is 3.18 Gigabytes. The other indices consume 6.92 Gigabytes, where about half of the indices are dedicated to the data indices and the other to the histogram indices of LUPOSDATE. Further information of the LUPOSDATE indices is provided in [21].

5.2.2.2 Query Evaluation

This time we want to compare query execution times not only for using our approach to bloom filters or using no bloom filters at all, but with also filtering with dynamically generated bloom filters like in traditional approaches.

The first option for the traditional approach is to use the built-in bloom filter approach in HBase itself. However, we do not have the possibility in HBase to access the bloom filter itself directly (such that we cannot build the bloom filter of the whole query as described in Section 4.3). Furthermore, HBase computes for each table only one bloom filter. For some tables like S_PO, we need two separate bloom filters for P and O for effective filtering according to the single components, but not a combined one like what HBase provides.

The second option is to use bloom filters as provided in the Pig Latin language. In our scenario, we can build the bloom filter from the result of one triple pattern and filter the result of the other triple patterns before joining. Again similarly to the HBase built-in bloom filter approach, there is no possibility for a Pig Latin script to build the bloom filter of the whole query as described in Section 4.3. However, the simple approach of building the bloom filter based on the result of one triple pattern and filtering the others' results with it is more close to the traditional approach. In our experiments we modified the Pig Latin scripts generated by our approach to use the Pig Latin built-in bloom filter approach and ran these scripts directly. We always chose to build the bloom filter from the most selective triple pattern.

We have chosen four queries (see Appendix B) with an increasing number of results. Appendix B also contains the Pig Latin scripts implementing the traditional bloom filter approach. The results are shown in Table 19. Again using bloom filters greatly improves query processing time in the cloud. More complex queries can benefit mostly from the bloom filter approach: For example, our bloom filter approach has a speedup of more than 7 for query Q3 and of more than 8 for query Q4. Due to the time needed for dynamic bloom filter computation as well as not a more precise bloom filter of the *whole* query is computed, the traditional approach (here called *Pig Latin Bloom Filter*) is always slower than our ap-

proach. The difference grows for more complex queries: Our bloom filter approach is about 3.5 times faster than the Pig Latin Bloom Filter approach for query Q3 and 4.1 times faster for query Q4. However, the local installation still beats the cloud technologies by several magnitudes, although the gap becomes closer when using bloom filters in the cloud.

Query	Local		With Bloom Filter		Without Bloom Filter		Pig Latin Bloom Filter	
	Time	σ [%]	Time	σ [%]	Time	σ [%]	Time	σ [%]
Q1	0.2 sec	14	00:02:28	2.7	00:03:16	0.1	00:09:27	1.7
Q2	00:00:08	0.6	00:03:07	3.1	00:03:17	1.2	00:09:23	0.6
Q3	00:00:05	0.3	00:04:21	3.4	00:31:59	5.59	00:15:13	1.7
Q4	00:00:10	0.4	00:04:55	3.1	00:42:36	3.3	00:20:26	0.6
Q1-Q4	00:00:05	3.8	00:03:42	3.08	00:20:09	2.55	00:13:37	1.15

Table 19: Results for the D2R DBLP extraction (times in [hours:minutes:seconds])

5.2.2.3 Determination of the bloom filters

Table 20 lists the computation times for our bloom filters. Again, they are quite slow in comparison to the total query execution times.

Query	Computation Time
Q1	10
Q2	39
Q3	25
Q4	24

Table 20: Computation time of bloom filters in seconds for the D2R DBLP extraction

6 Summary and Conclusions

In this paper we propose the Semantic Web Cloud database, P-LUPOSDATE. We combine the advantages of two existing systems: (a) We use the Hexastore distribution strategy [43], such that the result of any triple pattern can be accessed within one index access; and (b) we furthermore use the Pig framework with its abstraction language Pig Latin to express the processing steps for SPARQL queries like in [40]. We use LUPOSDATE as the basic Semantic Web engine to inherit its support of SPARQL 1.1 by processing operations not supported by Pig in LUPOSDATE.

To reduce network traffic and improve query processing time, we further propose a bloom filter variant which can be smoothly integrated into the Hexastore distribution strategy. In order to avoid an additional MapReduce step, we precompute and store all possible bloom filters during the import of the data and argue that updates are no problem for our approach. We propose to store bloom filters in a hybrid way. Bloom filters with few bits can be fast transformed into byte-bit vectors and hence are stored by enumerating the indices of the set bits in order to save storage space. The other bloom filters are precomputed and stored as byte-bit vectors during the import of the data in order to save processing time.

A comprehensive experimental evaluation shows that our system can handle big data with data sets containing up to 1 billion triples efficiently. Our bloom filter variant always speeds up query processing. Our experiments show that a speedup factor of more than 8 is possible. Our bloom filter approach is also faster than the traditional bloom filter approach: In our experiments we obtain a speedup of more than 4 for some queries.

Our future work will investigate other storage formats for bloom filter like compressed bit vectors [28] or *counting bloom filters (CBF)* [11], which may have advantages in scenarios with frequent updates. Also the usage of dictionaries [41] in order to save storage space and network traffic and to lower the main memory footprint is a promising future direction.

References

[1] Amazon.com, Inc., "Amazon Elastic MapReduce (Amazon EMR)," http://aws.amazon.com/de/elasticmapreduce, 2014.

[2] Apache Software Foundation, "Apache Pig: Performance and Efficiency - Compress the Results of Intermediate Jobs," http://pig.apache.org/docs/r0.11.1/perf.html#compression, 2013.

[3] ——, "Hadoop Homepage," http://hadoop.apache.org, 2014.

[4] ——, "Welcome to Apache Pig!" http://pig.apache.org, 2014.

[5] ——, "HBase Bulk Loading," http://hbase.apache.org/book/arch.bulk.load.html, accessed on 4.9.2014.

[6] M. Arias, J. D. Fernández, and M. A. Martínez-Prieto, "An empirical study of real-world sparql queries," in *1st International Workshop on Usage Analysis and the Web of Data (USEWOD 2011), Hyderabad, India*, 2011. [Online]. Available: http://dataweb.infor.uva.es/wp-content/uploads/2011/06/usewod2011.pdf

[7] T. Berners-Lee and D. Connolly, "Notation3 (N3): A readable RDF syntax," W3C, W3C

Team Submission, 2008. [Online]. Available: http://www.w3.org/TeamSubmission/n3/

[8] T. Berners-Lee and M. Fischetti, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*, 1st ed. Harper San Francisco, 1999.

[9] K. Beyer, V. Ercegovac, E. Shekita, N. L. Jun Rao, and S. Tata, "JAQL: Query Language for JavaScript(r) Object Notation (JSON)," https://code.google.com/p/jaql, accessed on 4.9.2014.

[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[11] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Algorithms–ESA 2006*. Springer, 2006, pp. 684–695.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[13] H. Choi, J. Son, Y. Cho, M. K. Sung, and Y. D. Chung, "SPIDER: a system for scalable, parallel/distributed evaluation of large-scale RDF data," in *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 2009, pp. 2087–2088.

[14] Cloudera Inc., "Cloudera Project Homepage," http://www.cloudera.com, 2014.

[15] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[16] T. Connolly and C. Begg, *Database systems: a practical approach to design, implementation, and management*, ser. International computer science series. Addison-Wesley, 2005.

[17] P. Deutsch, "GZIP file format specification version 4.3," http://tools.ietf.org/html/rfc1952, 1996.

[18] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea, "Apples and oranges: A comparison of rdf benchmarks and real rdf datasets," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 145–156. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989340

[19] EMC Corporation, "New Digital Universe Study Reveals Big Data Gap: Less Than 1http://www.emc.com/about/news/press/2012/20121211-01.htm, 2012.

[20] B. Furht and A. Escalante (editors), *Handbook of Cloud Computing*. Springer, 2010.

[21] S. Groppe, *Data Management and Query Processing in Semantic Web Databases*. Springer, 2011.

[22] ——, "LUPOSDATE Semantic Web Database Management System," https://github.com/luposdate/luposdate, 2013, [Online; accessed 16.10.2013].

[23] F. Heine, "Scalable p2p based RDF querying," in *Proceedings of the 1st international conference on Scalable information systems*. ACM, 2006, p. 17.

[24] M. F. Husain, P. Doshi, L. Khan, and B. Thuraisingham, "Storage and retrieval of large rdf graph using hadoop and mapreduce," in *Cloud Computing*. Springer, 2009, pp. 680–686.

[25] M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham, "Data intensive query processing for large RDF graphs using cloud computing tools," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, 2010, pp. 1–10.

[26] L3S Research Center, "D2r server publishing the dblp bibliography database," http://dblp.l3s.de/d2r, 2014, [Online; accessed 9.9.2014].

[27] T. Lee, K. Kim, and H.-J. Kim, "Join Processing Using Bloom Filter in MapReduce," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, ser. RACS '12. New York, NY, USA: ACM, 2012, pp. 100–105. [Online]. Available: http://doi.acm.org/10.1145/2401603.2401626

[28] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," *Data & Knowledge Engineering*, vol. 69, no. 1, pp. 3–28, 2010.

[29] Linked Data, "Linked Data - Connect Distributed Data across the Web," 2013. [Online]. Available: http://www.linkeddata.org

[30] P. Mell and T. Grance, "The nist definition of cloud computing," National Institute of Standards and Technology (NIST), Gaithersburg, MD, Tech. Rep. 800-145, September 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[31] K. Möller, M. Hausenblas, R. Cyganiak, and G. A. Grimnes, "Learning from linked open data usage: Patterns & metrics," in *Proceedings of the WebSci10: Extending the Frontiers of Society On-Line.*

Web Science Conference (WebSci), April 26-27, Raleigh,, North Carolina, USA, 2010.

[32] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 647–659, 2008.

[33] ——, "Scalable join processing on very large RDF graphs," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 627–640.

[34] Z. Nie, F. Du, Y. Chen, X. Du, and L. Xu, "Efficient SPARQL query processing in mapreduce through data partitioning and indexing," in *Web Technologies and Applications*. Springer, 2012, pp. 628–635.

[35] M. F. Oberhumer, "Lempel-Ziv-Oberhumer Project Site," http://www.oberhumer.com/opensource/lzo, 2014.

[36] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[37] E. Oren, C. Guéret, and S. Schlobach, "Anytime query answering in RDF through evolutionary algorithms," in *The Semantic Web-ISWC 2008*. Springer, 2008, pp. 98–113.

[38] E. Redmond, J. Wilson, and J. Carter, *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*, ser. Oreilly and Associate Series. Pragmatic Bookshelf, 2012. [Online]. Available: http://books.google.de/books?id=lA3QygAACAAJ

[39] A. Schätzle, "PigSPARQL: Eine Übersetzung von SPARQL nach Pig Latin," *Masterthesis*, University Freiburg, 2010.

[40] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen, "PigSPARQL: Mapping SPARQL to Pig Latin," in *Proceedings of the International Workshop on Semantic Web Information Management*. ACM, 2011, p. 4.

[41] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel, "An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario," in *The Semantic Web-ISWC 2008*. Springer, 2008, pp. 82–97.

[42] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench: A SPARQL Performance Benchmark," *CoRR*, vol. abs/0806.4627, 2008.

[43] J. Sun and Q. Jin, "Scalable RDF store based on HBase and MapReduce," in *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, vol. 1. IEEE, 2010, pp. V1–633.

[44] Y. Tanimura, A. Matono, I. Kojima, and S. Sekiguchi, "Storage Scheme for Parallel RDF Database Processing Using Distributed File Ssytem and MapReduce," in *Proceedings of HPC Asia*, 2009, pp. 312–319.

[45] University Trier, "The DBLP Computer Science Bibliography," http://dblp.uni-trier.de, accessed on 4.9.2014.

[46] I. Varley, "HBaseCon 2012 Presentation: HBase Schema Design," http://ianvarley.com/coding/HBaseSchema_HBaseCon2012.pdf, 2012.

[47] J. Weaver and G. T. Williams, "Reducing I/O Load in Parallel RDF Systems via Data Compression," in *1st Workshop on High-Performance Computing for the Semantic Web (HPCSW 2011)*, 2011.

[48] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, 2008.

[49] T. White, *Hadoop: the definitive guide*. O'Reilly, 2012.

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

[51] ——, "SPARQL Query Language for RDF," *W3C Recommendation*, http://www.w3.org/TR/rdf-sparql-query, 2008.

[52] ——, "SPARQL 1.1 Query Language," *W3C Recommendation*, http://www.w3.org/TR/sparql11-query, 2013.

[53] ——, "SPARQL 1.1 Query Language: Translation to the SPARQL Algebra," http://www.w3.org/TR/sparql11-query/#sparqlQuery, 2013.

[54] ——, "W3C SEMANTIC WEB ACTIVITY," http://www.w3.org/2001/sw, 2013.

[55] G. Zeev Tarantov, Steinar H. Gunderson, "SNAPPY Project Site," https://code.google.com/p/snappy, accessed on 4.9.2014.

[56] X. Zhang, L. Chen, and M. Wang, "Towards efficient join processing over large RDF graph using mapreduce," in *Scientific and Statistical Database Management*. Springer, 2012, pp. 250–259.

APPENDIX A - TRANSFORMATION TO PIG LATIN

We describe the transformation from relational expressions (to which SPARQL queries can be transformed) into Pig Latin syntax in this section. The following functions are used within the transformation rules:

- $bag(R)$**:** This function returns for a given relational expression R a unique name (also called *alias* in Pig), which is used as the name for the Pig Latin bag for the result of R.

- $pos(b, \{v_1, ..., v_n\})$**:** This function returns the position of one or more variables v_1 to v_n in the bag b. For the bag $(var1 = a, var2 = b)$ we retrieve \$1 when applying $pos(bag, \{var2\})$, and (\$1, \$2) for $pos(bag, \{var1, var2\})$.

We use also some placeholders which are given in Table 21.

Placeholder	Placeholder represents...
R, R_x	Relational expression
c, c_x	RDF term (blank node, IRI or literal)
v, v_x	variable
i, i_x	variable or RDF term
(i_1, i_2, i_3)	triple pattern

Table 21: Placeholders in the transformation rules

A transformation rule consists of a left side and a right side separated by a right arrow ($\Rightarrow$). The transformation rules are applied in the following way. We assume that the relational expression R represents the query. Then we apply the function $map(R)$ in order to retrieve a Pig Latin program for calculating the result of R. Whenever the left side of a transformation rule matches the relational expression to be transformed, the relational expression is replaced with the right side of the transformation rule. The right side might contain again functions like map, which are recursively applied.

We collect several transformation rules for one purpose in a tranformation rule set.

Transforming Triple Patterns

The most inner relational expression is always a triple pattern. In order to retrieve the result of a triple pattern, we have to load the data by choosing the right index. Transformation Rule Set 1 contains the transformation rules for handling triple patterns.

Transforming Joins and Cross Products

The join as well as the cross product have several relational expressions as input. The join can be joined over several variables v_1 to v_n. They can be transformed to Pig Latin syntax in a straightforward way (see Transformation Rule Sets 2 and 3).

Transformation Rule Set 1: Triple Pattern

$map((i_1, i_2, i_3)) \Rightarrow$
  $bag((i_1, i_2, i_3))$ = $load((i_1, i_2, i_3))$
  $bag((i_1, i_2, i_3))$ = $mapToBag((i_1, i_2, i_3))$

$load((?s, c_1, c_2)) \Rightarrow$
  load 'hbase://PO_S' using HBaseLoad(
    'Hexa','$c1$,$c2$','bf_$?s$') as (...);
$mapToBag((?s, c_1, c_2)) \Rightarrow$
  FOREACH $bag((?s, c_1, c_2))$ GENERATE
    flatten(MapToBag(\$0,\$1)) as (...);

  $\vdots$

$load((?s, ?p, c_1)) \Rightarrow$
  load 'hbase://O_SP' using HBaseLoad(
    'Hexa','$c1$','bf_$?s$','bf_$?p$') as (...);
$mapToBag((?s, ?p, c_1)) \Rightarrow$
  FOREACH $bag((?s, ?p, c_1))$ GENERATE
    flatten(MapToBag(\$0,\$1,\$2)) as (...);

$load((?s, ?p, ?o)) \Rightarrow$
  load 'hbase://S_PO' using HBaseLoad(
    'Hexa','-loadKey true','bf_$?s$','bf_$?p$',
    'bf_$?o$') as (...);
$mapToBag((?s, ?p, ?o)) \Rightarrow$
  FOREACH $bag((?s, ?p, ?o))$ GENERATE n
    flatten(MapToBag(\$0,\$1,\$2,\$3,\$4)) as (...);

Transformation Rule Set 2: Join

$map(R_1 \bowtie_{v_1,...,v_n} ... \bowtie_{v_1,...,v_n} R_n) \Rightarrow$
  $map(R_1)$

  $\vdots$

  $map(R_n)$
  $bag(R_1 \bowtie_{v_1,...,v_n} ... \bowtie_{v_1,...,v_n} R_n)$ = JOIN
    $bag(R_1)$ BY $pos(bag(R_1), \{v_1, ..., v_n\})$,

    $\vdots$

    $bag(R_n)$ BY $pos(bag(R_n), \{v_1, ..., v_n\})$;

Transformation Rule Set 3: Cross Product

$map(R_1 \times ... \times R_n) \Rightarrow$
  $map(R_1)$

  $\vdots$

  $map(R_n)$
  $bag(R_1 \times ... \times R_n)$=CROSS $bag(R_1)$, ..., $bag(R_n)$;

Transforming Projection

Projections can be computed in Pig Latin in a *Foreach*-loop with keyword *Generate* (see Transformation Rule

Set 4).

Transformation Rule Set 4: Projection

$map(\pi_{v_1,...,v_n}(R)) \Rightarrow$
  $map(R)$
  $bag(\pi_{v_1,...,v_n}(R))$ = FOREACH $bag(R)$
    GENERATE $pos(bag(R), \{v_1, ..., v_n\})$;

Transforming Selection

The FILTER clause in SPARQL 1.1 is very expressive. Besides simple comparisons between variables or a variable and an RDF term, various built-in functions are supported. The most important constructs in the FILTER clause are also supported by P-LUPOSDATE. The support of missing ones can be easily added. The following language constructs are supported (see Transformation Rule Set 5):

- **Comparisons:** All simple comparisons by $<$, $>$, $<=$, $>=$, != and = are supported. The Pig Latin syntax also covers a FILTER operator, which supports these types of comparison, and which we use in the mapped Pig Latin program.
- **Built-In Functions:** We currently support only the $bound(?v)$ built-in function, which checks whether a variable $?v$ is bound with a value. We have implemented a user-defined function in Pig Latin for this purpose. The support of other built-in functions can be added similarly.
- **AND:** Combining boolean formulas in the FILTER clause with the boolean *and* operator is transformed to Pig Latin by a subsequent application of the FILTER operator with the boolean formulas.
- **OR:** The FILTER clause supports to combine its boolean formulas by *or*, which is also supported by the Pig Latin FILTER operator.

Transforming Sorting

The same as in SPARQL, there is also an *Order-By* operator in Pig Latin, which can be directly used for sorting purposes (see Transformation Rule Set 6).

Transforming Duplicate Elimination

Similar to the SPARQL *DISTINCT* construct, Pig Latin offers a *DISTINCT* operator, which we use for duplicate elimination (see Transformation Rule Set 7).

Transformation Rule Set 5: Filter

$map(\sigma_{f_1 \wedge f_2}(R)) \Rightarrow map(\sigma_{f_1}(\sigma_{f_2}(R)))$
$map(\sigma_f(R)) \Rightarrow map(R)$
  $bag(\sigma(R))$ = FILTER $bag(R)$
    BY $map(bag(R), f)$;
$map(b, f_1 \vee f_2) \Rightarrow map(b, f_1)$ OR $map(b, f_2)$
$map(b, bound(v)) \Rightarrow BoundFilterUDF(pos(b,v))$
$map(b, i_1\ op\ i_2) \Rightarrow map(b, i_1)\ op\ map(b, i_2)$,
  where $op \in \{<, >, >=, <=, != \}$
$map(b, c) \Rightarrow c$
$map(b, v) \Rightarrow pos(b, \{v\})$

Transformation Rule Set 6: Order By

$map(\tau_v(R)) \Rightarrow map(R)$
  $bag(\tau_v(R))$ = ORDER $bag(R)$
    BY $pos(bag(R), \{v\})$;

Transformation Rule Set 7: Distinct

$map(\delta(R)) \Rightarrow map(R)$
  $bag(\delta(R))$ = DISTINCT $bag(R)$;

Transforming Limit

With the help of the limit operator the query result can be limited to a given number of results. Pig Latin also supports this functionality with the *LIMIT* construct (see Transformation Rule Set 8).

Transformation Rule Set 8: Limit

$map(limit_n(R)) \Rightarrow$
  $map(R)$
  $bag(limit_n(R))$ = LIMIT $bag(R)$ n;

Transforming Union

The Union operator can be implemented in Pig Latin with the *UNION* construct (see Transformtion Rule Set 9).

Transformation Rule Set 9: Union

$map(R_1 \cup ... \cup R_n) \Rightarrow$
  $map(R_1)$
  $\vdots$
  $map(R_n)$
  $bag(R_1 \cup ... \cup R_n)$ = UNION $bag(R_1)$, ..., $bag(R_n)$;

Transforming Optional

The SPARQL *Optional* construct causes a left outer join. Pig Latin also supports outer joins and especially left outer joins. The transformation is straight forward (see Transformation Rule Set 10).

However, SPARQL supports a left outer join over *null*-values, whereas Pig Latin does not, i.e., for very special cases the transformed Pig Latin program does not yield the same results as SPARQL queries do, and is hence incorrect. These special cases can be detected by a static analysis of the queries. As consequence these queries cannot be computed completely in the Cloud, and at least the left outer join must be computed at the master node, which is the default strategy of P-LUPOSDATE for all language constructs not supported by Pig Latin.

Transformation Rule Set 10: Optional

$map(R_1 \bowtie_{v_1,...,v_n} R_2) \Rightarrow$
$map(R_1)$
$map(R_2)$
$bag(R_1 \bowtie R_2)$ = JOIN $bag(R_1)$ BY
$pos(bag(R_1), \{v_1, ..., v_n\})$ LEFT OUTER,
$bag(R_2)$ BY $pos(bag(R_2), \{v_1, ..., v_n\})$;

Appendix B - Queries for Real-World Data

This section contains the queries we used for querying the content of the D2R Server publishing the DBLP Bibliography (see Section 5.2). We describe the queries in natural language, present the SPARQL query itself as well as the Pig Latin script of the query using the Pig Latin built-in bloom filter, with which we compare our approach in Section 5.2.2.2.

Query Q1

The first query determines all coauthors of Le Gruenwald.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?coauthor WHERE {
  ?article dc:creator <http://dblp.l3s.de/d2r/
       resource/authors/Le_Gruenwald>.
  ?article dc:creator ?coauthor.
}
```

Listing 4: Query 1

The Pig Latin script for the traditional bloom filter approach determines the bloom filter of all publications of Le Gruenwald and filters with it the publication-author relationship.

```
define bb BuildBloom('jenkins', '1000', '0.1');
P0 = load 'hbase://po_s' using lupos.cloud.pig.
     udfs.HBaseLoadUDF('Hexa', '', '<http://
     purl.org/dc/elements/1.1/creator>,<http
     ://dblp.l3s.de/d2r/resource/authors/
     Le_Gruenwald>') as (columncontent_0:map
     []);
I0 = foreach P0 generate flatten(lupos.cloud.pig.
     udfs.MapToBagUDF($0)) as (output0:
     chararray);
B = group I0 all;
C = foreach B generate bb(I0.output0);
store C into 'mybloom';
define bloom Bloom('mybloom');
P1 = load 'hbase://p_so' using lupos.cloud.pig.
     udfs.HBaseLoadUDF('Hexa', '', '<http://
     purl.org/dc/elements/1.1/creator>') as (
     columncontent_1:map[]);
I1 = foreach P1 generate flatten(lupos.cloud.pig.
     udfs.MapToBagUDF($0)) as (output1_1:
     chararray, output2_1:chararray);
D = filter I1 by bloom($0);
I2 = JOIN I0 BY $0, D BY $0 PARALLEL 5;
I3 = FOREACH I2 GENERATE $2;
X = DISTINCT I3 PARALLEL 5;
```

Listing 5: Pig Latin script of query 1 for the traditional bloom filter approach

Query Q2

The second query retrieves a lots of information of each of the publications of Le Gruenwald.

```
PREFIX swrc: <http://swrc.ontoware.org/
     ontology#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf
     -syntax-ns#>
SELECT DISTINCT * WHERE {
  ?article dc:creator <http://dblp.l3s.de/d2r/
       resource/authors/Le_Gruenwald>.
  ?article dc:title ?title.
  ?article dc:creator ?creator.
  ?article foaf:homepage ?url.
  ?article dcterms:partOf ?partOf.
  ?article swrc:pages ?pages.
}
```

Listing 6: Query 2

Again the Pig Latin script for the traditional bloom filter approach determines the bloom filter of all publications of Le Gruenwald, but filters now with it all the other publication relationships.

```
define bb BuildBloom('jenkins', '1000', '0.1');
P0 = load 'hbase://po_s' using lupos.cloud.pig.
     udfs.HBaseLoadUDF('Hexa', '', '<http://
     purl.org/dc/elements/1.1/creator>,<http
```

```
      ://dblp.l3s.de/d2r/resource/authors/
      Le_Gruenwald>') as (columncontent_0:map
      []);
3 I0 = foreach P0 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output0:
      chararray);
4 B = group I0 all;
5 C = foreach B generate bb(I0.output0);
6 store C into 'mybloom';
7 define bloom Bloom('mybloom');
8 P1 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/elements/1.1/title>') as (
      columncontent_1:map[]);
9 I1 = foreach P1 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_1:
      chararray, output2_1:chararray);
10 I2 = filter I1 by bloom($0);
11 P2 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/elements/1.1/creator>') as (
      columncontent_2:map[]);
12 I3 = foreach P2 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_2:
      chararray, output2_2:chararray);
13 I4 = filter I3 by bloom($0);
14 P3 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      xmlns.com/foaf/0.1/homepage>') as (
      columncontent_4:map[]);
15 I5 = foreach P3 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_4:
      chararray, output2_4:chararray);
16 I6 = filter I5 by bloom($0);
17 P4 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/terms/partOf>') as (
      columncontent_7:map[]);
18 I7 = foreach P4 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_7:
      chararray, output2_7:chararray);
19 I8 = filter I7 by bloom($0);
20 P5 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      swrc.ontoware.org/ontology#pages>') as (
      columncontent_8:map[]);
21 I9 = foreach P5 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_8:
      chararray, output2_8:chararray);
22 I10 = filter I9 by bloom($0);
23 I11 = JOIN I0 BY $0, I2 BY $0, I4 BY $0, I6
      BY $0, I8 BY $0, I10 BY $0 PARALLEL
      5;
24 X = DISTINCT I11 PARALLEL 5;
```

Listing 7: Pig Latin script of query 2 for the traditional bloom filter approach

Query Q3

The third query asks for the conferences Le Gruenwald participated and for the publications presented there.

```
1 PREFIX swrc: <http://swrc.ontoware.org/
      ontology#>
2 PREFIX dcterms: <http://purl.org/dc/terms/>
3 PREFIX dc: <http://purl.org/dc/elements/1.1/>
4 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf
      -syntax-ns#>
5 SELECT DISTINCT ?title WHERE {
6   ?article dc:creator <http://dblp.l3s.de/d2r/
        resource/authors/Le_Gruenwald>.
7   ?article dcterms:partOf ?conf.
8   ?conf rdf:type swrc:Proceedings.
9   ?article2 dcterms:partOf ?conf.
10  ?article2 dc:title ?title.
11 }
```

Listing 8: Query 3

Now 3 bloom filters are generated in the Pig Latin script for the content of the variables ?article, ?conf and ?article2.

```
1 define bbArticle BuildBloom('jenkins', '1000',
      '0.1');
2 define bbConf BuildBloom('jenkins', '1000', '
      0.1');
3 define bbArt2 BuildBloom('jenkins', '1000', '
      0.1');
4 P0 = load 'hbase://po_s' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/elements/1.1/creator>,<http
      ://dblp.l3s.de/d2r/resource/authors/
      Le_Gruenwald>') as (columncontent_0:map
      []);
5 I0 = foreach P0 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output0:
      chararray);
6 BArticle = group I0 all;
7 CArticle = foreach BArticle generate bbArticle(
      I0.output0);
8 store CArticle into 'article';
9 P1 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/terms/partOf>') as (
      columncontent_1:map[]);
10 I1 = foreach P1 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_1:
      chararray, output2_1:chararray);
11 define bloomArticle Bloom('article');
12 I2 = filter I1 by bloomArticle($0);
13 BConf = group I2 all;
14 CConf = foreach BConf generate bbConf(I2.
      output2_1);
15 store CConf into 'conf';
16 P2 = load 'hbase://po_s' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      www.w3.org/1999/02/22-rdf-syntax-ns#type
      >,<http://swrc.ontoware.org/ontology#
      Proceedings>') as (columncontent_2:map[])
      ;
17 I3 = foreach P2 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output2:
      chararray);
18 define bloomConf Bloom('conf');
19 I4 = filter I3 by bloomConf($0);
20 P3 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
```

```
      purl.org/dc/terms/partOf>') as (
      columncontent_3:map[]);
I5 = foreach P3 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_3:
      chararray, output2_3:chararray);
I6 = filter I5 by bloomConf($1);
BArt2 = group I6 all;
CArt2 = foreach BArt2 generate bbArt2(I6.
      output1_3);
store CArt2 into 'art2';
P4 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/elements/1.1/title>') as (
      columncontent_4:map[]);
I7 = foreach P4 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_4:
      chararray, output2_4:chararray);
define bloomArt2 Bloom('art2');
I8 = filter I7 by bloomArt2($0);
I9 = JOIN I4 BY $0, I2 BY $1, I6 BY $1
      PARALLEL 5;
I10 = JOIN I0 BY $0, I9 BY $1 PARALLEL 5;
I11 = JOIN I8 BY $0, I10 BY $4 PARALLEL 5;
I12 = FOREACH I11 GENERATE $1;
X = DISTINCT I12 PARALLEL 5;
```

Listing 9: Pig Latin script of query 3 for the traditional bloom filter approach

Query Q4

The fourth query's result is the set of authors Le Gruenwald could have met at the conferences in which she participated.

```
PREFIX swrc: <http://swrc.ontoware.org/
      ontology#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf
      -syntax-ns#>
SELECT DISTINCT ?name WHERE {
  ?article dc:creator <http://dblp.l3s.de/d2r/
       resource/authors/Le_Gruenwald>.
  ?article dcterms:partOf ?conf.
  ?article2 dcterms:partOf ?conf.
  ?conf rdf:type swrc:Proceedings.
  ?article2 dc:creator ?name.
}
```

Listing 10: Query 4

Again we compute 3 bloom filters for the content of the variables ?article, ?conf and ?article2 in the Pig Latin script.

```
define bbArticle BuildBloom('jenkins', '1000',
      '0.1');
define bbConf BuildBloom('jenkins', '1000', '
      0.1');
define bbArt2 BuildBloom('jenkins', '1000', '
      0.1');
P0 = load 'hbase://po_s' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/elements/1.1/creator>,<http
      ://dblp.l3s.de/d2r/resource/authors/
      Le_Gruenwald>') as (columncontent_0:map
      []);
I0 = foreach P0 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output0:
      chararray);
BArticle = group I0 all;
CArticle = foreach BArticle generate bbArticle(
      I0.output0);
store CArticle into 'article';
P1 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/terms/partOf>') as (
      columncontent_1:map[]);
I1 = foreach P1 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_1:
      chararray, output2_1:chararray);
define bloomArticle Bloom('article');
I2 = filter I1 by bloomArticle($0);
BConf = group I2 all;
CConf = foreach BConf generate bbConf(I2.
      output2_1);
store CConf into 'conf';
P2 = load 'hbase://po_s' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      www.w3.org/1999/02/22-rdf-syntax-ns#type
      >,<http://swrc.ontoware.org/ontology#
      Proceedings>') as (columncontent_2:map[])
      ;
I3 = foreach P2 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output2:
      chararray);
define bloomConf Bloom('conf');
I4 = filter I3 by bloomConf($0);
P3 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/terms/partOf>') as (
      columncontent_3:map[]);
I5 = foreach P3 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_3:
      chararray, output2_3:chararray);
I6 = filter I5 by bloomConf($1);
BArt2 = group I6 all;
CArt2 = foreach BArt2 generate bbArt2(I6.
      output1_3);
store CArt2 into 'art2';
P4 = load 'hbase://p_so' using lupos.cloud.pig.
      udfs.HBaseLoadUDF('Hexa', '', '<http://
      purl.org/dc/elements/1.1/creator>') as (
      columncontent_4:map[]);
I7 = foreach P4 generate flatten(lupos.cloud.pig.
      udfs.MapToBagUDF($0)) as (output1_4:
      chararray, output2_4:chararray);
define bloomArt2 Bloom('art2');
I8 = filter I7 by bloomArt2($0);
I9 = JOIN I4 BY $0, I2 BY $1, I6 BY $1
      PARALLEL 5;
I10 = JOIN I0 BY $0, I9 BY $1 PARALLEL 5;
I11 = JOIN I8 BY $0, I10 BY $4 PARALLEL 5;
I12 = FOREACH I11 GENERATE $1;
X = DISTINCT I12 PARALLEL 5;
```

Listing 11: Pig Latin script of query 4 for the traditional bloom filter approach

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.

Thomas Kiencke was born in Ludwigslust, Germany in 1987. He received his M.Sc. in Computer Science in 2013 from the University of Lübeck, Germany. He wrote his master thesis at the Institute of Information Systems (IFIS) about the topic ”Semantic Web Database in the Cloud”.

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 FPGAs.

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.

Marc Stelzner has received his M.Sc. in Computer Science in 2013 from the University of Lübeck, Germany. He is employed as a research scientist and student coordinator at the Institute of Information Systems and the Institute of Telematics at the University of Lübeck. His research interests include wireless sensor networks integration in upcoming nanonetworks.

Dr. **Le Gruenwald** is a Professor, Dr. David W. Franke Professor, and Samuel Roberts Noble Foundation Presidential Professor in the School of Computer Science at The University of Oklahoma, U.S.A. She received her Ph.D. in Computer Science from Southern Methodist University, M.S. in Computer Science from the University of Houston, and B.S. in Physics from the University of Saigon. She also worked for National Science Foundation (NSF) as a Cluster Lead and Program Director of the Information Integration and Informatics program and a Program Director of the Cyber Trust program. Her major research interests include Database Management, Data Mining, and Information Privacy and Security. She has published more than 180 technical articles in journals, books and conference proceedings. She is a member of ACM, SIGMOD, and IEEE Computer Society.