Map-Side Merge Joins for Scalable SPARQL BGP Processing

Martin Przyjaciel-Zablocki*, Alexander Schätzle*, Eduard Skaley, Thomas Hornung, Georg Lausen

Department of Computer Science
University of Freiburg
Georges-Köhler-Allee 051, 79110 Freiburg, Germany
zablocki|schaetzle|skaley|hornungt|lausen@informatik.uni-freiburg.de

***Abstract*—In recent times, it has been widely recognized that, due to their inherent scalability, frameworks based on MapReduce are indispensable for so-called "Big Data" applications. However, for Semantic Web applications using SPARQL, there is still a demand for sophisticated MapReduce join techniques for processing basic graph patterns, which are at the core of SPARQL. Renowned for their stable and efficient performance, sort-merge joins have become widely used in DBMSs. In this paper, we demonstrate the adaptation of merge joins for SPARQL BGP processing with MapReduce. Our technique supports both n-way joins and sequences of join operations by applying merge joins within the map phase of MapReduce while the reduce phase is only used to fulfill the preconditions of a subsequent join iteration. Our experiments with the LUBM benchmark show an average performance benefit between 15% and 48% compared to other MapReduce based approaches while at the same time scaling linearly with the RDF dataset size.**

I. Introduction

System architectures for processing "Big Data" typically follow a layered approach: the front tier is responsible for answering simple queries in real-time as low latencies are essential. More complex analyses are performed offline in batches and results are pushed to the front tier in intervals (cf. e.g. [1]). Typical representatives of such long-running queries are e.g. "who knows whom" queries that require many costly joins. Due to its inherent high degree of parallelism and good scalability properties, MapReduce [2] is one of the predominant frameworks used in many large companies for dealing with "Big Data". Thus, it is a natural candidate for processing long-running queries in the background. Although, it might not be the most efficient solution wrt. node utilization, it gracefully handles load-balancing on top of commodity hardware, especially when it comes to rapidly growing datasets where its built-in fault tolerance becomes another advantage.

Given the wide adoption of Semantic Web technologies, the amount of available RDF data [3] has also grown into dimensions where it is crucial that solutions scale out [4] as witnessed by the annual Semantic Web Challenge[1]. Consequently, MapReduce has been applied for query processing on top of large RDF graphs with (a subset of) SPARQL [4], [5], [6], the official query language for RDF [7], [8]. Following this line of research, we investigate the efficient computation of SPARQL basic graph patterns (BGPs) – which are at the core of SPARQL – on top of MapReduce. The computation of SPARQL BGPs translates to the evaluation of joins on the operator level, an area which has been extensively studied by the database community in the past [9]. Here, due to their stable and efficient performance, merge joins have emerged as a widely adopted solution used in many databases.

In this paper, we present an implementation of a distributed (n-way) sort-merge join on top of MapReduce, where the join is computed completely in the map phase. It addresses the problem of cascaded executions by using the reduce phase of MapReduce to assure that the "left-hand" side of the join is sorted wrt. the join attribute(s). Our data model assures that the "right-hand" side of the join is always pre-sorted on the required attributes. For the reduction of intermediate results, bloom filters [10], [11] are used to remove dangling tuples. A comparison of our approach to other MapReduce based join techniques showed that our system exhibits a performance benefit of 15% to 48% on average over all LUBM [12] queries. Our proof-of-concept implementation is also available for download[2].

The remainder of the paper is structured as follows: Section II gives an introduction to RDF, SPARQL, and MapReduce. It is followed by a conceptual overview of our approach in Section III. Section IV introduces the data store layout for our merge join implementation that is presented in Section V. Section VI reports the results of our experimental comparison of different SPARQL BGP implementations. Section VII discusses related work and Section VIII summarizes our results.

II. Preliminaries

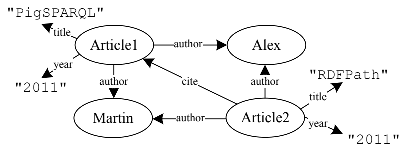
A. RDF & SPARQL

RDF [3] is the W3C recommended standard model for representing knowledge about arbitrary resources, e.g. articles and authors. An RDF dataset consists of a set of RDF triples in the form (*subject predicate object*) that can be interpreted as "*subject* has property *predicate* with value *object*". For clarity of presentation, we use a simplified RDF notation in the following. It is possible to visualize an RDF dataset as directed labeled graph where every triple corresponds to an edge (predicate) from subject to object. Figure 1 shows an RDF graph about articles and corresponding authors.

* The work of these authors was partially funded by Deutsche Forschungsgemeinschaft (DFG), grant LA 598/7-1.

[1]http://challenge.semanticweb.org/

[2]http://dbis.informatik.uni-freiburg.de/forschung/projekte/DiPoS/



SPARQL BGP query

```
SELECT *
WHERE {
  ?article1 title ?title .
  ?article1 year  "2011" .
  ?article2 cite  ?article1
}
```

Fig. 1. Example RDF graph and SPARQL BGP query

SPARQL is the W3C recommended declarative query language for RDF. A SPARQL query defines a graph pattern P that is matched against an RDF graph G. This is done by replacing the variables in P with elements of G such that the resulting graph is contained in G (pattern matching). The most basic construct in a SPARQL query is a *triple pattern*, i.e. an RDF triple where subject, predicate and object can be variables, e.g. $(?s\ p\ ?o)$. That is, a triple pattern selects a subset of an RDF graph that matches the bound values in the pattern. A set of triple patterns concatenated by AND (.) is then called a *basic graph pattern* (BGP) as illustrated in Figure 1. The query asks for all articles published in 2011 that are cited by at least one article. The result of a BGP is defined to be the intersection of all subsets defined by the corresponding triple patterns and can be computed by joining the results of all triple patterns on their shared variables, in this case $?article1$. For a detailed definition of the SPARQL syntax we refer the interested reader to the official W3C Recommendation [7]. A formal definition of the SPARQL semantics can also be found in [8]. In this paper we focus on efficient join processing with MapReduce and therefore only consider SPARQL BGPs.

B. MapReduce

The MapReduce programming model [2] enables scalable, fault tolerant and massively parallel computations using a cluster of machines. MapReduce is built on top of a distributed filesystem where large files are split into equal sized blocks, spread across the cluster and fault tolerance is achieved by replication. *Hadoop* is the most prominent open source implementation of MapReduce. The workflow of a MapReduce program is a sequence of MapReduce iterations each consisting of a *map* and a *reduce* phase. A user has to implement the map and reduce functions which are automatically executed in parallel on a portion of the data. The map function gets invoked for every input record represented as a key-value pair and outputs a list of new intermediate key-value pairs. In the reduce phase these intermediate pairs are first sorted and grouped by their key. The reduce function gets then invoked for every distinct key together with the list of all according values and outputs a list of values which can be used as input for the next MapReduce iteration. We omit a more detailed introduction to MapReduce due to space limitations.

III. Our Approach in a Nutshell

Processing joins on large datasets is even with MapReduce a challenging task [13]. If we want to join two datasets with MapReduce, $L \bowtie R$, we have to ensure that the subsets of L and R with the same join key values can be processed on the same machine. With respect to RDF, the join key is the shared variable between triple patterns (e.g. $?article1$ in Figure 1) and datasets can be different RDF graphs but also subsets of the same RDF graph. This is typical for most SPARQL queries as BGPs essentially correspond to self-joins between subsets of the same graph.

For joining arbitrary datasets on arbitrary keys we generally have to shuffle data over the network or choose appropriate pre-partitioning and replication strategies. The most prominent and flexible join technique in MapReduce is the so-called *Reduce-Side* or *Repartition Join* [13] where the idea is based on reading both datasets (map phase) and repartition them according to the join key. The actual join computation is done in the reduce phase. The main drawback of this approach is that both datasets are completely transferred over the network regardless of the join output. This is especially inefficient for selective joins and consumes a lot of network bandwidth.

In this paper, we present an adaptation of the classical sort-merge join, a well known technique in database systems, where we use the map phase for join computation to reduce network I/O. The key idea is to first sort datasets L and R by the join key such that identifying equal values in both datasets can be done using interleaved linear scans. Consequently, the first thing we have to guarantee is that L and R are always sorted by join key and also the join output has to be sorted according to the join key of the next join iteration in a sequence of joins. Furthermore, for an efficient parallel execution in a cluster of N machines we have to divide the join task into N independent subtasks where each subtask can be processed by exactly one machine. Therefore, we split both (sorted) datasets in N non-overlapping subsets of continuous key ranges such that $L = \bigcup_{1 \leq i \leq N} L_i$ and $R = \bigcup_{1 \leq i \leq N} R_i$. If we use the same key ranges for both datasets, it holds that $L \bowtie R = \bigcup_{1 \leq i \leq N} L_i \bowtie R_i$ (cf. Figure 2). Our data preprocessing and store layout is described in detail in Section IV. Driven by this data partitioning, the map phase can process an efficient parallel merge join between pre-sorted dataset splits. We use the subsequent reduce phase to guarantee that the join output fullfills the preconditions for the next iteration, i.e. it must be sorted according to the next join key and split into N subsets such that key ranges match with the next join partition. Furthermore, we use *dynamic bloom filters* to discard *dangling tuples* in intermediate join results, i.e. tuples where the bloom filter guarantees that they will not find a join partner in the next iteration and hence do not contribute to the final query result. The join processing is described in detail in Section V.

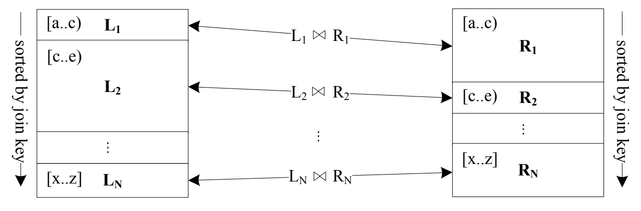


Fig. 2. Distributed merge join as a union of N independent subtasks

IV. RDF Data Store Layout

Since the input datasets have to be sorted by join key to apply a merge join and basic graph patterns operate on a single input RDF graph, it is reasonable to perform a data preprocessing that reduces the sorting effort during query execution. Furthermore, it is a common practice to partition the RDF graph into smaller subsets such that triple pattern matching can be done more efficiently [14]. In this section we describe our data store layout for RDF that (1) partitions the data to efficiently support the most common triple pattern types and (2) ensures that we only have to sort the output of the previous join while the second input is always pre-sorted.

Based on the ideas in [14] we split the data using a vertical partitioning schema where all triples with the same predicate are stored in the same first level partition. We call such a partition a *P-partition*. Similar to [5], we complement these partitioning schema by also looking at the objects such that all triples with the same predicate and object are also stored in the same second level partition, denoted as *PO-partition* (cf. Figure 3). That is, for every triple $(s\ p\ o)$, there exists a tuple $(s\ o)$ in P-partition p and an entry (s) in PO-partition $p|o$. The original RDF graph and all partitions (P and PO) are stored in the distributed filesystem (HDFS). Technically, they are stored using the `SequenceFile` format of Hadoop that allows comparisons on byte level. We do not consider the combination of predicate and subject since this would result in many small partitions which is undesired in a MapReduce framework.

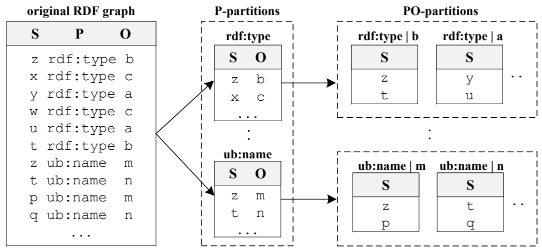


Fig. 3. General store layout with P and PO-partitions

To reduce the sorting effort during query execution we perform a pre-sorting of the input dataset and the partitions by all possible attributes, i.e. the overall input dataset (RDF graph) is sorted three times by subject, predicate and object, P-partitions are sorted twice by subject and object and PO-partitions are sorted by subject.

As already mentioned in Section III, we have to split every sorted partition into N non-overlapping subsets of continuous key ranges. This is achieved during the initial sorting of the partition by assigning continuous and non-overlapping key ranges to the N reducers. The key ranges are derived by a *sampling* of the partition values, i.e. a representative sample of the values is extracted and key ranges are assigned such that there is a uniform and ordered distribution of partition values to the resulting N subsets. To get an uniform distribution, the sample is taken from the subject values if the partition is sorted by subject or from the object values if it is sorted by object.

This layout works fine for joins between triple patterns where the join variable is on the same position, e.g. subject-subject joins (cf. first two triple patterns in Figure 1). For these joins, both sides must be sorted by the same attribute (e.g. by subject) and therefore they also have the same key ranges. However, when it comes to mixed join variable positions, e.g. the rather common subject-object join (cf. last two triple patterns in Figure 1), both sides must be sorted by different attributes (one side by subject, the other side by object) and hence the key ranges of both sides will not match in general. But if the key ranges do not match, the join result is not guaranteed to be complete. To overcome this problem, we use two different samplings when sorting a partition. For example, when sorting a P-partition by subject, we do not only pick a sample of the subject values but also a sample of the object values and derive two different key ranges from these samples. We then split the sorted partition in two ways according to subject key ranges and object key ranges, respectively. Hence, a P-partition is actually stored four times in our store layout (cf. Figure 4 for P-partition *rdf:type*). To reduce the storage overhead introduced by this layout, we compress every partition using the snappy compression library[3] that is already shipped with Hadoop such that the final store size is actually smaller than the original uncompressed input (cf. Table I in Section VI).

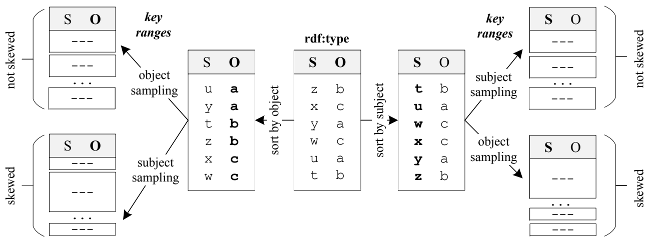


Fig. 4. Detailed view of sorting and key ranges for P-partition *rdf:type*

A drawback of this approach is that data distribution can be skewed in general when key ranges are not derived from the sort attribute (e.g. key ranges derived from object sampling while sorting by subject). For joins on the same variable position, e.g. subject-subject joins, this is not a problem as we can use the key ranges that give a uniform distribution for both sides. But in case of mixed join variable positions, one side will be equally balanced (where the sampling fits to the sorting) while the other side can be more or less unbalanced. This is best illustrated by an example. Consider the BGP $(a\ b\ ?x\ .\ ?x\ c\ d)$ with two triple patterns that translates to an object-subject join on variable $?x$. Consequently, for the first triple pattern we use the P-partition b sorted by object (and filtered by subject a) and for the second pattern we use the PO-partition $c|d$ sorted by subject. But in addition, both sides must have the same key ranges and hence use the same sampling. In this case, we could either use key ranges derived by object or subject sampling for both sides. The former gives an uniform distribution for the partition of the first triple pattern but a potential skew for the second pattern and vice versa.

Data skew handling in parallel joins has already been studied in research (e.g. [15]). Our solution follows a greedy approach, i.e. we always use the sampling that is optimal for the larger of both sides. Though this is not an optimal solution in theory, our experiments confirm that it works fine in practice for most queries. Nonetheless, this is a crucial point for future optimizations of our approach.

A. Dynamic Bloom Filter Integration

A *bloom filter* [10] is a space-efficient probabilistic data structure used to check whether a given element is contained in a set. It consists of a bit vector of size m and k different

[3]https://code.google.com/p/snappy/

uniform hash functions that map an element to one of the m bit vector positions. The filter is constructed by applying the hash functions to each element of the set and setting all corresponding positions to 1. To check whether an element is contained in the set, all k hash functions are applied.

We use bloom filters to remove dangling intermediate results, i.e. results that do not contribute to the final query result. To this end, we build up a bloom filter for every of the N subsets of a partition during the initial sorting (cf. Figure 5) and store them on every machine in the cluster by setting the number of replications to N. We can then access these filters locally during join execution to discard those intermediate results where the filter guarantees that they will not find a join partner in the partition of the next join iteration. This is done in the map function for every intermediate join output. The efficiency of this approach strongly relies on the false positive probability but for static bloom filters this can only be estimated if the number of elements to be inserted is known a-priori such that the bloom filter size can be determined in advance. For that reason, we use dynamic bloom filters [11] which are essentially a collection of standard bloom filters that increases dynamically with the number of inserted elements while guaranteeing a pre-defined false positive probability.

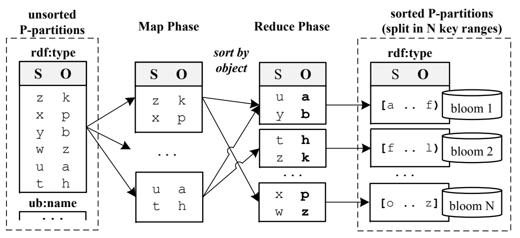


Fig. 5. Initial sorting (object) and bloom filter creation for P-parition *rdf:type*

V. Map-Side Merge Join with MapReduce

After the initial data store generation, the actual BGP query processing can be devided into three subtasks: (1) First, we have to select the input partitions of the data store that match the triple patterns of the BGP. (2) The results for every triple pattern are iteratively joined in the map phase. (3) If there is more than one join iteration the join output has to be post-processed in the reduce phase, i.e. it must be sorted and split into N subsets.

First, we introduce the SPARQL terminology that is used in the following analogous to [8]: Let V be the infinite set of query variables and T be the set of valid RDF terms.

Definition 1: A (solution) mapping μ is a partial function $\mu : V \to T$. We call $\mu(?v)$ the variable binding of μ for $?v$. Abusing notation, for a triple pattern p we call $\mu(p)$ the triple that is obtained by substituting the variables in p according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined.

Definition 2: Two mappings μ_1, μ_2 are compatible, $\mu_1 \sim \mu_2$, iff for every variable $?v \in dom(\mu_1) \cap dom(\mu_2)$ it holds that $\mu_1(?v) = \mu_2(?v)$. It follows that mappings with disjoint domains are always compatible and the set-union (merge) of μ_1 and μ_2, $\mu_1 \cup \mu_2$, is also a mapping.

Definition 3: The answer to a triple pattern p for an RDF graph G is a list of mappings $\Omega = \{\mu \mid \mu(p) \in G\}$ without a given order. The join of two lists of mappings, $\Omega \bowtie \Omega'$, is defined as the merge of the compatible mappings in Ω and Ω', $\Omega \bowtie \Omega' = \{(\mu_1 \cup \mu_2) \mid \mu_1 \in \Omega, \mu_2 \in \Omega', \mu_1 \sim \mu_2\}$.

For the following discussion, we consider the example SPARQL BGP from Figure 1 in Section II-A which consists of three triple patterns p_1, p_2, p_3:
$(?art1\ title\ ?title\ .\ ?art1\ year\ 2011\ .\ ?art2\ cite\ ?art1)$
Let $\Omega^1, \Omega^2, \Omega^3$ denote the mappings for p_1, p_2, p_3, respectively. The query result is then defined as $\Omega^1 \bowtie \Omega^2 \bowtie \Omega^3$. Furthermore, we use the notation Ω^1_i to refer to the i-th subset of Ω^1 defined by key ranges, i.e. Ω^1_i and Ω^2_i have the same key range for the join key. It follows that $\Omega^1 \bowtie \Omega^2 = \bigcup_{1 \le i \le N}(\Omega^1_i \bowtie \Omega^2_i)$. In our example, the join key is $?article1$ for all triple patterns.

A. Input Selection

For every triple pattern in a BGP we have to (1) identify the corresponding partition, (2) select the appropriate sorting and (3) choose the matching key ranges. The partition selection is derived from the bounded values in a triple pattern. Regarding the example, we would choose P-partitions *title* and *cite* for p_1, p_3, respectively, and PO-partition *year*|*2011* for p_2. The entries of these partitions directly correspond to $\Omega^1, \Omega^2, \Omega^3$. If there is no partition that directly matches the given pattern, e.g. if the subject is bound, we choose the most appropriate partition and apply a filter in the map phase before feeding the data to the map function. The sorting of the selected partitions is defined by the position of the join variable, i.e. partitions for p_1 and p_2 must be sorted by subject whereas the partition for p_3 must be sorted by object. The selection of the matching key ranges has already been outlined in Section IV. For the first join between Ω^1 and Ω^2 the choice is clear as it is a subject-subject join, hence we can use key ranges derived by subject sampling which means that both sides are equally balanced. However, the second join between the result of $(\Omega^1 \bowtie \Omega^2)$ and Ω^3 is a subject-object join where we have to decide whether we use key ranges derived by subject or object sampling. Without loss of generality, we assume $|(\Omega^1 \bowtie \Omega^2)| < |\Omega^3|$. Thus, we choose key ranges derived by object sampling such that the partition splits for p_3 are equally balanced.

B. 2-Way Merge Join

After the input selection, the query result is computed by a sequence of cascaded 2-way merge joins as illustrated in Figure 6. The input partitions (recall that the entries correspond to the solution mappings for the triple patterns) are pre-sorted by the join key and split into N subsets with matching key ranges, e.g. $\Omega^1 = \bigcup_{1 \le i \le N} \Omega^1_i$. Within the map phase, every machine in the cluster computes the partial join between two subsets with matching key ranges, i.e. $(\Omega^1_i \bowtie \Omega^2_i)$.

Due to the locality principle of MapReduce, one subset is always read locally. However, we cannot guarantee that both subsets with the same key range are stored on the same machine as data placement is done by the distributed filesystem where we store the partitions (for Hadoop this is HDFS). In general, the larger subset is chosen to be processed locally whereas the smaller subset has to be transferred over the network at the beginning of the map phase. This is automatically handled by Hadoop. Thus, in every join iteration only the smaller subset (which is typically the output of the previous join iteration) is transferred, in contrast to reduce-side joins where typically both sides must be transferred. However, it is a topic of our future developments to improve the co-locality of matching key ranges such that both sides can be read locally.

The merge-join algorithm for the map function is illustrated in Algorithm 1. On every machine the map function is invoked with a composite key consisting of the current join key and the join key of the next iteration, if any. The value is also a composite value consisting of the corresponding subsets of solution mappings (these are essentially the entries of the input partitions) and the bloom filter of the next join partition. Regarding our example, the map invocation for the i-th mapper in the first iteration would be

$$\texttt{map}(\{?article1, ?article1\}, \{\Omega_i^1, \Omega_i^2, bloom(\Omega_i^3)\}).$$

The current join key is $article1$ and this is also the join key of the next iteration. The map function computes $\Omega_i^1 \bowtie \Omega_i^2$ and discards those mappings where the bloom filter membership test fails, i.e. for every merge of compatible mappings it is checked whether the value of the next join key is contained in the bloom filter of the next join partition. If this test fails, it is guaranteed that there is no join partner in the next iteration and the mapping can be discarded.

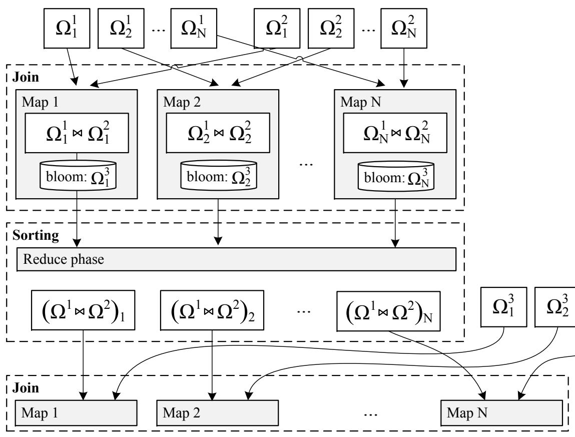


Fig. 6. Cascaded 2-way map-side merge join

For a cascaded join sequence we use the subsequent reduce phase to sort the join output, $\Omega^1 \bowtie \Omega^2$, according to the join key of the next iteration and to split it into N subsets such that the key ranges match with the key ranges of the next join partition, $\Omega^1 \bowtie \Omega^2 = \bigcup_{1 \leq i \leq N} (\Omega^1 \bowtie \Omega^2)_i$ (cf. lower half of Figure 6). In our example, we would use the same key ranges that are used for the input partition matching p_3. Therefore, we also store the key ranges of a partition such that we can reuse them for intermediate join results. As the sorting and assignment of values to reducers (partitioning) is done when shuffling data from mappers to reducers, the reduce function is only an identity function that just stores its input to HDFS. In the following join iteration, the i-th mapper then computes $(\Omega^1 \bowtie \Omega^2)_i \bowtie \Omega_i^3$ and so on.

Algorithm 1: 2-way merge join - **map**(key, value)

```
input : key = $\{k', k''\}$, value = $\{\Omega_i, \Omega_i', bloom(\Omega_i'')\}$
        // $k'$ is the current join key, $k''$ is the join key of the next join
        // $\Omega_i$ and $\Omega_i'$ are sorted by join key and have the same key range
        // $bloom(\Omega_i'')$ is the bloom filter of the next join subset $\Omega_i''$
output: $\Omega_i \bowtie \Omega_i'$
$l \leftarrow 1, r \leftarrow 1$
while $l \leq |\Omega_i|$ and $r \leq |\Omega_i'|$ do
    $\mu_1 \leftarrow \Omega_i[l], \mu_2 \leftarrow \Omega_i'[r]$
    if $\mu_1(k') = \mu_2(k')$ then // $\mu_1 \sim \mu_2$
        $r' \leftarrow r$ // temporary pointer to iterate over all compatible $\mu_2 \in \Omega_i'$
        while $\mu_1 \sim \mu_2$ do
            // emit merge of $\mu_1$, $\mu_2$ if it passes the bloom filter
            // membership test for the next join key $k''$
            if $(\mu_1 \cup \mu_2)(k'') \in bloom(\Omega_i'')$ then emit$\{(\mu_1 \cup \mu_2)(k''), (\mu_1 \cup \mu_2)\}$
            $r' \leftarrow r' + 1, \mu_2 \leftarrow \Omega_i'[r']$
        end
        $l \leftarrow l + 1$
    else $\mu_1 \nsim \mu_2$
        if $\mu_1(k') < \mu_2(k')$ then $l \leftarrow l + 1$ else $r \leftarrow r + 1$
    end
end
```

C. N-Way Merge Join

If the join key is the same in a sequence of n 2-way merge joins, we can also compute the result with a single n-way merge join, thus saving $n - 1$ MapReduce iterations. In our example, the join key for both 2-way join iterations is $?article1$, so we could also use a single 3-way join instead. The basic principle is the same as for the 2-way join but instead of two subsets each machine joins n subsets in a single map phase, i.e. $(\Omega_i^1 \bowtie \cdots \bowtie \Omega_i^n)$. Just like for the 2-way join, it cannot be guaranteed that all n subsets with the same key range are stored on the same machine, hence the missing subsets must be transferred to the corresponding machine at the beginning of the map phase. Input partition selection and also post-processing in the reduce phase is the same as for the 2-way join. The 2-way merge join algorithm in Algorithm 1 can be easily extended for n-way merge joins using n interleaved linear scans instead of two. However, a disadvantage of the n-way join compared to a sequence of 2-way joins is that we do not benefit from bloom filters for intermediate results. Hence, we can only apply bloom filters on the results of the n-way join (if another join iteration follows). Yet, our experiments demonstrate that, in general, the saving of MapReduce iterations has a greater impact on query performance than discarding dangling intermediate results.

VI. Experiments

The experiments were performed on a cluster of ten machines equipped with a 6 core Xeon E5-2420 CPU (1.9 GHz) and 32 GB RAM connected via gigabit network using the Hadoop distribution of Cloudera in version 4.2.1. We used the well-known Lehigh University Benchmark (LUBM) [12] and generated datasets from 500 up to 3000 universities where we pre-computed the transitive closure using the WebPIE inference engine for Hadoop [16]. The store generation runtimes and dataset sizes are listed in Table I. We can observe that the actual store size is even smaller than the size of the original RDF graph. This is achieved by replacing prefixes and applying snappy compression which reduced the original dataset size by up to 92%.

TABLE I. GENERATING RDF DATA STORE

LUBM	**500**	**1000**	**1500**	**2000**	**2500**	**3000**
triples (million)	105	210	315	420	525	630
input size (GB)	17.0	34.1	51.3	68.5	85.7	102.9
overall store size (GB)	13.6	27.3	41.0	54.8	68.6	82.3
store generation (minutes)	68	111	154	193	241	279

We compared our merge join approach with three systems based on MapReduce. (1) *HadoopRDF* [5] is an advanced SPARQL engine that splits the original RDF graph according to predicates and objects and utilizes a cost-based query execution plan for reduce-side joins. (2) *MAPSIN* [17] is a map-side index nested loop join implementation based on HBase. It processes joins within the map phase and exploits n-way joins by a sophisticated storage schema that significantly reduces the amount of HBase lookups. (3) *PigSPARQL* [6] is a reduce-side join based SPARQL 1.0 query engine built on top of *Apache Pig*. The crucial point for this choice was the sophisticated reduce-side join implementation of Pig [18] that incorporates sampling and hash join techniques.

Figure 7 illustrates the scaling properties of our merge join approach. We can observe a linear scaling of query runtimes where tripling the dataset size does not even take twice the time for most queries. A comparison of execution times (wall time) to other approaches is summarized in Table II. MAPSIN lacks the support for LUBM queries 2, 9 and 10 whereas HadoopRDF runs out of storage space while creating its storage schema for datasets larger than LUBM 2000[4]. We considered a time-out of one hour, denoted by *T*, if a query fails to complete in time.

Queries 6 and 14 are simple and require no join at all. Nevertheless, our approach outperforms the other systems significantly as the whole processing is done locally on all cluster machines without any reduce phase. One may expect MAPSIN to be the fastest for such queries, since a single table lookup could provide the final result. However, such a request would push the result to only one machine violating the scaling properties. Therefore, MAPSIN processes such single pattern queries by a distributed table scan which is executed on each machine preserving scalability.

Queries 1, 3, 5 and 13 contain only one join. Our approach processes these queries within a single map phase as no additional processing for a subsequent join is required. Overall, the merge join performs best for these kind of queries.

Query 2 is rather complex (compared to other LUBM queries) as it exhibits a triangular pattern structure. Moreover, it contains a costly and unselective subject-object and object-subject join that points out a weakness of our approach. As both join partitions must use the same key ranges, one of the input partitions is fairly unbalanced (cf. Section IV) which increases the costs for join processing. Thus, it is a point of future work to develop more sophisticated sampling strategies for key ranges to improve data distribution for such cases.

Queries 7, 8 and 9 demonstrate the base case where several joins have to be processed sequentially. For these queries, the reduce phase is required to sort the join output according to the join key of the next iteration whereas bloom filters are used to remove dangling intermediate results. For queries 7 and 8 we can observe a quite competitive performance of our approach compared to both reduce-side join based systems. Indeed, such selective queries are the core of MAPSIN, where it benefits from its index structure accessing only those triples that are relevant for the query answer. Nonetheless, query runtimes of MAPSIN are close to the runtimes of the merge join approach.

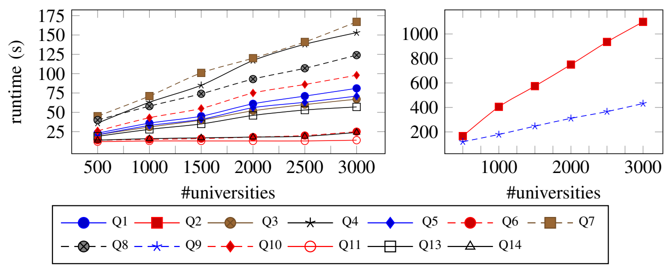


Fig. 7. Runtimes for all LUBM queries (Merge Join).

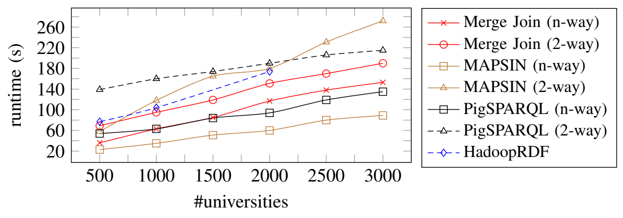
Query 4 is a star pattern query which makes it a good candidate for n-way joins. Figure 8.a illustrates a comparison between 2-way and n-way execution for Merge Join, MAPSIN and PigSPARQL as these systems support n-way joins. In all cases, n-way joins clearly outperform a sequential execution of 2-way joins, while the benefit for our merge join approach is less than for the others. Reducing the amount of MapReduce cycles comes at the cost of more data that has to be processed at once. As we cannot guarantee that all n matching subsets reside on the same machine, more data has to be accessed remotely. This is underpinned by Figure 8.b that shows the amount of data accessed locally and retrieved remotely within a map-side merge join. However, the difference is much less than expected. Processing query 4 with one 5-way merge join compared with several 2-way merge joins increases the amount of data retrieved remotely by only 32% while decreasing the amount of data accessed locally by 34%. This can be explained due to the fact that data is stored with an replication factor of three which increases the chances that matching subsets reside on the same machine. Except for n-way joins, the amount of data accessed locally within a merge join is always higher than the amount of data retrieved remotely, which is an expected behavior since we choose the larger dataset to be processed locally. Nevertheless, since data locality is a crucial point for distributed systems, improving these values, e.g. by a refinement of the data placement strategy wrt. matching subsets, is a worthwhile point for future optimizations.

The average performance benefit of our Merge Join compared to the other systems is between 15% and 48%. In order to compare the performance of the different systems we computed for each query the relative difference of execution time to the respective best case. Then, for each system the average of these relative differences over all queries is computed, whereas missing measure points are replaced with a weak penalty value. The penalty is based on the average of all systems that perform worse than the best execution time. Finally, we computed the relative performance distance of other systems to our approach (c.f. “relative perf.” in Table II). For example, if we refer to LUBM 1000, we can derive that PigSPARQL (72%) is in comparison with Merge Join (100%) on average 28% slower.

[4]We contacted the authors since neither a documentation nor an “out-of-the-box” running system was available, unfortunately we didn’t get any support.

TABLE II. QUERY EXECUTION TIMES FOR MERGE JOIN, MAPSIN, PIGSPARQL AND HADOOPRDF IN SECONDS. T: TIME-OUT

	LUBM query	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**13**	**14**	**relative perf.**
1000	Merge Join	36	405	31	63	33	15	71	58	179	43	13	28	16	100%
	MAPSIN	32	—	30	35	33	45	60	60	—	—	32	42	42	85%
	PigSPARQL	42	199	42	63	43	28	135	141	233	42	33	43	27	72%
	HadoopRDF	48	136	58	104	59	51	148	298	208	55	1585	186	54	52%
1500	Merge Join	45	574	40	85	41	16	101	74	247	55	13	35	17	100%
	MAPSIN	49	—	41	51	43	58	66	75	—	—	44	70	70	82%
	PigSPARQL	53	233	53	84	48	32	176	166	279	53	38	48	32	75%
	HadoopRDF	61	174	71	149	74	54	189	389	233	63	T	193	59	58%
2000	Merge Join	61	750	52	117	56	18	120	93	311	75	13	46	18	100%
	MAPSIN	52	—	47	60	52	67	93	92	—	—	51	81	78	85%
	PigSPARQL	64	283	63	94	58	38	216	212	329	63	49	53	42	78%
	HadoopRDF	77	196	80	174	86	58	215	457	257	68	T	206	62	63%
2500	Merge Join	71	935	60	138	63	20	141	107	366	86	13	53	19	100%
	MAPSIN	69	—	58	80	65	82	110	105	—	—	65	102	87	81%
	PigSPARQL	73	335	68	119	64	48	252	247	399	74	58	64	47	79%
3000	Merge Join	81	1099	67	153	71	25	167	124	432	98	14	57	24	100%
	MAPSIN	81	—	70	89	78	98	120	119	—	—	74	125	105	80%
	PigSPARQL	83	385	78	135	74	53	281	287	460	83	69	68	53	80%



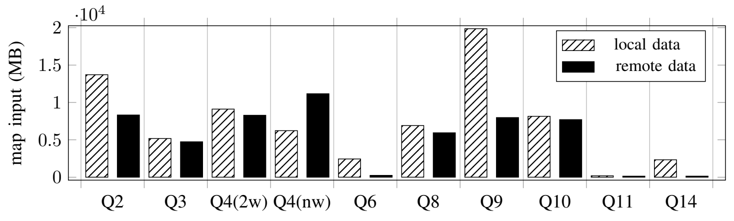


Fig. 8. (a) Comparison of n-way optimizations for LUBM Query 4. (b) Comparison of local and remote data access within a map-side merge join.

Overall, the experiments showed that our map-side merge join approach exhibits in most cases competitive runtimes with a performance benefit of 15% to 48% on average over all queries. It works best for single join queries but performs also good for sequences of joins. However, unselective subject-object or object-subject joins turned out to be a weak point, especially if one join input side is fairly unbalanced. But even for those queries, the differences to the fastest query execution times are still acceptable while showing an excellent scaling behavior at all time. Moreover, our store layout enables retrieving one pattern queries even faster than the index-based query execution of MAPSIN. Taken into account that improving data locality by adopting more suitable data placement strategies for Hadoop and preventing unbalanced partitions by more sophisticated partitioning strategies will further improve query execution times, we can conclude that map-side merge joins are well suited for processing SPARQL BGPs with MapReduce.

VII. RELATED WORK

In terms of mere query performance, *RDF-3X* [19] has established itself as the state-of-the-art "benchmark" engine for single place machines. However, its performance has been shown to degrade for queries with unbound objects and low selectivity factors [5]. Furthermore, with the ever increasing amount of available RDF data, single machine solutions for query processing become more and more challenging [4]. Thus, a number of systems that focus on distributed execution of SPARQL queries have been proposed in recent years (e.g. [20], [21], [22]). Since each of these implementations require some dedicated infrastructure and management, there are no synergy effects by reusing already deployed frameworks. Our research is driven by the idea to reuse existing infrastructures for "Big Data" scenarios. Consequently, we have ensured that no changes to the underlying Hadoop framework are required to run our SPARQL BGP engine. This way, existing Hadoop clusters or cloud services (e.g. Amazon EC2) can used without any changes.

The efficent computation of joins is the main driver for the performance of SPARQL BGP evaluation, and thus we have focused on join processing in MapReduce in this paper. This topic has already been studied considering various aspects and application fields [23], [24], [13], [25], [26]. In [25] the authors discussed how to process arbitrary joins (θ joins) using MapReduce, whereas [23] focuses on optimizing n-way joins. θ joins are not required for the evaluation of SPARQL BGPs, and they are not supported by our solution. The execution of n-way joins is a generalization of our 2-way join, where instead of two all n pre-sorted input partitions are processed in a single map phase. *Map-Reduce-Merge* [26] describes a modified MapReduce workflow by adding a merge phase after the reduce phase, whereas *Map-Join-Reduce* [24] proposes a join phase in between the map and reduce phase. Both techniques attempt to improve the support for joins in MapReduce but require profound modifications to the MapReduce framework. In [27] the authors present non-invasive index and join techniques for SQL processing in MapReduce that also reduce the amount of shuffled data at the cost of an additional co-partitioning and indexing phase at load time. However, the schema and workload is assumed to be known in advance which is typically feasible for relational data but does not hold for RDF in general. *HadoopDB* [28] is a hybrid of MapReduce and DBMS where MapReduce is the communication layer above multiple single node DBMS. The authors in [4] adopt this hybrid approach for the Semantic Web using RDF-3X. Initially, they partition the graph on a single machine in a loading phase. We also initially store the dataset wrt. different sort orders but we employ the MapReduce framework to partition the dataset at loading time. Common to both approaches is that data has to be reloaded in case of updates, while we do not require the installation of additonal engines at each cluster node. *HadoopRDF* [5] is a MapReduce based RDF system that stores data directly in HDFS and does also not require any changes to the Hadoop framework. It is able to rebalance automatically when cluster size changes but join processing is also done in the reduce phase. As already mentioned, our join processing is done in the map phase and additionally we reduce the amount of data sent over the network by proactively filtering dangling tuples using bloom filters.

VIII. Conclusion

In the area of "Big Data" applications, MapReduce has become a state-of-the-art technology for large-scale data processing. On the other side, the advent of the Semantic Web promotes the growing adoption of RDF and SPARQL as its core technologies, raising attention for distributed SPARQL query processing in current research. As basic graph patterns, which are at the core of SPARQL, translate to the computation of joins on the operator level, efficient distributed join techniques for RDF are of particular interest. The fixed ternary structure of RDF makes it a well suited candidate for sort-merge joins as presorting the data is affordable. In this paper we presented an adaptation of sort-merge joins for SPARQL basic graph patterns with MapReduce which supports both 2-way and n-way joins. The actual join computation is completely done in the map phase, complemented by bloom filters to discard dangling intermediate results, while the reduce phase is used to post-process the join output for subsequent join iterations. Our experiments with the LUBM bechmark demonstrated an average performance benefit between 15% and 48% of our approach compared to other MapReduce based systems while scaling smoothly with the dataset size. For future work, we will consider refinements of the data placement strategy to further optimize data locality as well as techniques to handle data skew in parallel joins.

References

[1] R. Sumbaly, J. Kreps, and S. Shah, "The Big Data Ecosystem at LinkedIn," in *SIGMOD Conference*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 1125–1134.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] F. Manola, E. Miller, and B. McBride. (2004) RDF Primer. W3C Recom. W3C. [Online]. Available: http://www.w3.org/TR/rdf-primer/

[4] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, vol. 4, no. 11, pp. 1123–1134, 2011.

[5] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing," *IEEE TKDE*, vol. 23, no. 9, 2011.

[6] 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 (SWIM)*, 2011, pp. 4:1–4:8.

[7] E. Prud'hommeaux and A. Seaborne. (2008) SPARQL Query Language for RDF. W3C Recom. W3C. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[8] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and Complexity of SPARQL," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 3, p. 16, 2009.

[9] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–170, 1993.

[10] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[11] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Applications of Dynamic Bloom Filters," in *INFOCOM*, 2006.

[12] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A Benchmark for OWL Knowledge Base Systems," *Web Semantics*, vol. 3, no. 2, pp. 158 – 182, 2005.

[13] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel Data Processing with MapReduce: A Survey," *SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2011.

[14] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," in *VLDB*, 2007, pp. 411–422.

[15] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling Data Skew in Parallel Joins in Shared-Nothing Systems," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08, 2008, pp. 1043–1052.

[16] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal, "WebPIE: A Web-scale Parallel Inference Engine using MapReduce," *J. Web Sem.*, vol. 10, pp. 59–75, 2012.

[17] A. Schätzle, M. Przyjaciel-Zablocki, C. Dorner, T. Hornung, and G. Lausen, "Cascading Map-Side Joins over HBase for Scalable Join Processing," in *Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+ HPCSW 2012)*, 2012, p. 59.

[18] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience," *PVLDB*, vol. 2, no. 2, 2009.

[19] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *PVLDB*, vol. 1, no. 1, pp. 647–659, 2008.

[20] O. Erling, "Virtuoso, a Hybrid RDBMS/Graph Column Store," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 3–8, 2012.

[21] S. Harris, N. Lamb, and N. Shadbolt, "4store: The Design and Implementation of a Clustered RDF Store," in *SSWS*, 2009, pp. 94–109.

[22] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "YARS2: A Federated Repository for Querying Graph Structured Data from the Web," *The Semantic Web*, 2007.

[23] F. N. Afrati and J. D. Ullman, "Optimizing Multiway Joins in a Map-Reduce Environment," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 9, pp. 1282–1298, 2011.

[24] D. Jiang, A. K. H. Tung, and G. Chen, "Map-Join-Reduce: Toward Scalable and Efficient Data Analysis on Large Clusters," *IEEE TKDE*, vol. 23, no. 9, pp. 1299–1311, 2011.

[25] A. Okcan and M. Riedewald, "Processing Theta-Joins using MapReduce," in *SIGMOD Conference*, 2011, pp. 949–960.

[26] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. P. Jr., "Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters," in *SIGMOD*, 2007.

[27] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)," *PVLDB*, vol. 3, no. 1, pp. 518–529, 2010.

[28] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.