Cascading Map-Side Joins over HBase for Scalable Join Processing

Alexander Schätzle, Martin Przyjaciel-Zablocki,
Christopher Dorner, Thomas Hornung, and Georg Lausen

Department of Computer Science, University of Freiburg, Germany
{schaetzle,zablocki,dornerc,hornungt,lausen}
@informatik.uni-freiburg.de

Abstract. One of the major challenges in large-scale data processing with MapReduce is the smart computation of joins. Since Semantic Web datasets published in RDF have increased rapidly over the last few years, scalable join techniques become an important issue for SPARQL query processing as well. In this paper, we introduce the *Map-Side Index Nested Loop Join* (MAPSIN join) which combines scalable indexing capabilities of NoSQL data stores like HBase, that suffer from an insufficient distributed processing layer, with MapReduce, which in turn does not provide appropriate storage structures for efficient large-scale join processing. While retaining the flexibility of commonly used reduce-side joins, we leverage the effectiveness of map-side joins without any changes to the underlying framework. We demonstrate the significant benefits of MAPSIN joins for the processing of SPARQL basic graph patterns on large RDF datasets by an evaluation with the LUBM and SP^{2}Bench benchmarks. For selective queries, MAPSIN join based query execution outperforms reduce-side join based execution by an order of magnitude.

1 Introduction

Most of the information in the classical *"Web of Documents"* is designed for human readers, whereas the idea behind the Semantic Web is to build a *"Web of Data"* that enables computers to understand and use the information in the web. The advent of this Web of Data gives rise to new challenges with regard to query evaluation on the Semantic Web. The core technologies of the Semantic Web are RDF (Resource Description Framework) [1] for representing data in a machine-readable format and SPARQL [2] for querying RDF data. However, querying RDF datasets at web-scale is challenging, especially because the computation of SPARQL queries usually requires several joins between subsets of the data. On the other side, classical single-place machine approaches have reached a point where they cannot scale with respect to the ever increasing amount of available RDF data (cf. [16]). Renowned for its excellent scaling properties, the MapReduce paradigm [8] is an attractive candidate for distributed SPARQL processing. The *Apache Hadoop* platform is the most prominent and widely used open-source MapReduce implementation. In the last few years many companies have built-up their own Hadoop infrastructure but there are also ready-to-use cloud services like Amazon's Elastic Compute Cloud (EC2) offering the Hadoop platform as a service (PaaS). Thus, in contrast to specialized distributed RDF systems like YARS2 [15] or 4store [14], the use of existing Hadoop MapReduce infrastructures enables scalable, distributed and fault-tolerant SPARQL processing out-of-the-box without any additional installation or management overhead. Following this avenue, we introduced the *PigSPARQL* project in [26] that offers full support for SPARQL 1.0 and is implemented on top of Hadoop. However, while the performance and scaling properties of PigSPARQL for complex analytical queries are competitive, the performance for selective queries is not satisfying due to the lack of built-in index structures and unnecessary data shuffling as join computation is done in the reduce phase.

In this paper we present a new MapReduce join technique, the *Map-Side Index Nested Loop Join* (MAPSIN join), that uses the indexing capabilities of a distributed NoSQL data store to improve query performance of selective queries. MAPSIN joins are completely processed in the map phase to avoid costly data shuffling by using HBase as underlying storage layer. Our evaluation shows an improvement of up to one order of magnitude over the common reduce-side join for selective queries. Overall, the major contributions of this paper are as follows:

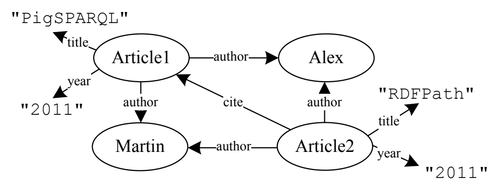
- We describe a space-efficient storage schema for large RDF graphs in HBase while retaining favourable access characteristics. By using HBase instead of HDFS, we can avoid shuffling join partitions across the network and instead only access the relevant join partners in each iteration.
- We present the MAPSIN join algorithm, which can be evaluated cascadingly in subsequent MapReduce iterations. In contrast to other approaches, we do not require an additional shuffle and reduce phase in order to preprocess the data for consecutive joins. Moreover, we do not require any changes to the underlying frameworks.
- We demonstrate an optimization of the basic MAPSIN join algorithm for the efficient processing of multiway joins. This way, we can save n MapReduce iterations for star join queries with $n + 2$ triple patterns.

The paper is structured as follows: Section 2 provides a brief introduction to the technical foundations for this paper. Section 3 describes our RDF storage schema for HBase, while Section 4 presents the MAPSIN join algorithm. We continue with a presentation of the evaluation of our approach in Section 5, followed by a discussion of related work in Section 6. We conclude in Section 7 and give an outlook on future work.

2 Background

2.1 RDF & SPARQL

RDF [1] 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 with information about articles and corresponding authors.



SPARQL BGP query

```
SELECT *
WHERE {
  ?article title  ?title .
  ?article author ?author .
  ?article year   ?year
}
```

Fig. 1. RDF graph and SPARQL 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 constructs in a SPARQL query are *triple patterns*, i.e. RDF triples where subject, predicate and object can be variables, e.g. ($?s$, p, $?o$). A set of triple patterns concatenated by AND (.) is called a *basic graph pattern* (BGP) as illustrated in Figure 1. The query asks for all articles with known title, author and year of publication. The result of a BGP is computed by joining the variable mappings of all triple patterns on their shared variables, in this case $?article$. For a detailed definition of the SPARQL syntax we refer the interested reader to the official W3C Recommendation [2]. A formal definition of the SPARQL semantics can also be found in [23]. In this paper we focus on efficient join processing with MapReduce and NoSQL (i.e. HBase) and therefore only consider SPARQL BGPs.

2.2 MapReduce

The MapReduce programming model [8] enables scalable, fault tolerant and massively parallel computations using a cluster of machines. The basis of Google's MapReduce is the distributed file system GFS [12] where large files are split into equal sized blocks, spread across the cluster and fault tolerance is achieved by replication. The workflow of a MapReduce program is a sequence of MapReduce iterations each consisting of a *Map* and a *Reduce* phase separated by a so-called *Shuffle* & *Sort* phase. A user has to implement *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. It outputs a list of new intermediate key-value pairs which are then sorted and grouped by their key. The reduce function gets invoked for every distinct intermediate 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 use *Apache Hadoop* as it is the most popular open-source implementation of Google's GFS and MapReduce framework that is used by many companies like Yahoo!, IBM or Facebook.

Map-Side vs. Reduce-Side Join. Processing joins with MapReduce is a challenging task as datasets are typically very large [5,20]. 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. 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 called *Reduce-Side Join* [5,20]. Some literature also refer to it as *Repartition Join* [5] as the idea is based on reading both datasets (map phase) and repartition them according to the join key (shuffle phase). 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. Another group of joins is based on getting rid of the shuffle and reduce phase to avoid transferring both datasets over the network. This kind of join technique is called *Map-Side Join* since the actual join processing is done in the map phase. The most common one is the *Map-Side Merge Join* [20]. However, this join cannot be applied on arbitrary datasets. A preprocessing step is necessary to fulfill several requirements: datasets have to be sorted and equally partitioned according to the join key. If the preconditions are fulfilled, the map phase can process an efficient parallel merge join between pre-sorted partitions and data shuffling is not necessary. However, if we want to compute a sequence of such joins, the shuffle and reduce phases are needed to guarantee that the preconditions for the next join iteration are fulfilled. Therefore, map-side joins are generally hard to cascade and the advantage of avoiding a shuffle and reduce phase is lost. Our MAPSIN join approach is designed to overcome this drawback by using the distributed index of a NoSQL system like HBase.

2.3 HBase

HBase is a distributed, scalable and strictly consistent column-oriented NoSQL data store, inspired by Google's Bigtable [7] and well integrated into Hadoop. Hadoop's distributed file system, HDFS, is designed for sequential reads and writes of very large files in a batch processing manner but lacks the ability to access data randomly in close to real-time. HBase can be seen as an additional storage layer on top of HDFS that supports efficient random access. The data model of HBase corresponds to a sparse multi-dimensional sorted map with the following access pattern:

$$(Table, RowKey, Family, Column, Timestamp) \rightarrow Value$$

The rows of a table are sorted and indexed according to their *row key* and every row can have an arbitrary number of *columns*. Columns are grouped into *column families* and column values (denoted as cell) are timestamped and thus support multiple versions. HBase tables are dynamically split into *regions* of contiguous row ranges with a configured maximum size. When a region becomes too large, it is automatically split into two regions at the middle key (auto-sharding).

However, HBase has neither a declarative query language nor built-in support for native join processing, leaving higher-level data transformations to the overlying application layer. In our approach we propose a map-side join strategy that leverages the implicit index capabilities of HBase to overcome the usual restrictions of map-side joins as outlined in Section 2.2.

3 RDF Storage Schema for HBase

In contrast to relational databases, NoSQL data stores do neither have a common data model nor a common query language like SQL. Hence, the implementation of our join approach strongly relies on the actual NoSQL store used as backend. In our initial experiments we considered HBase and Cassandra, two popular NoSQL stores with support for MapReduce. We decided to use HBase for our implementation as it proved to be more stable and also easier to handle in our cluster since HBase was developed to work with Hadoop from the beginning.

In [28] the authors adopted the idea of Hexastore [30] to index all possible orderings of an RDF triple for storing RDF data in HBase. This results in six tables in HBase allowing to retrieve results for any possible SPARQL triple pattern with a single lookup on one of the tables (except for a triple pattern with three variables). However, as HDFS has a default replication factor of three and data in HBase is stored in files on HDFS, an RDF dataset is actually stored 18 times using this schema. But it's not only about storage space, also loading a web-scale RDF dataset into HBase becomes very costly and consumes many resources. Our storage schema for RDF data in HBase is inspired by [10] and uses only two tables, T_{s_po} and T_{o_ps}. We extend the schema with a triple pattern mapping that leverages the power of predicate push-down filters in HBase to overcome possible performance shortcomings of a two table schema. Furthermore, we improve the scalibility of the schema by introducing a modified row key design for class assignments in RDF which would otherwise lead to overloaded regions constraining both scalability and performance.

In T_{s_po} table an RDF triple is stored using the subject as row key, the predicate as column name and the object as column value. If a subject has more than one object for a given predicate (e.g. an article having more than one author), these objects are stored as different versions in the same column. The notation T_{s_po} indicates that the table is indexed by subject. Table T_{o_ps} follows the same design. In both tables there is only one single column family that contains all columns. Table 1 illustrates the corresponding T_{s_po} table for the RDF graph in Section 2.1.

Table 1. T_{s_po} table for RDF graph in Section 2.1

rowkey	**family:column→value**
Article1	p:title→{"PigSPARQL"}, p:year→{"2011"}, p:author→{Alex, Martin}
Article2	p:title→{"RDFPath"}, p:year→{"2011"}, p:author→{Martin, Alex}, p:cite→{Article1}

At first glance, this storage schema seems to have performance drawbacks when compared to the six table schema in [28] since there are only indexes for subjects and objects. However, we can use the HBase Filter API to specify additional column filters for table index lookups. These filters are applied directly on server side such that no unnecessary data must be transferred over the network (*predicate push-down*). As already mentioned in [10], a table with predicates as row keys causes scalability problems since the number of predicates in an ontology is usually fixed and relatively small which results in a table with just a few very fat rows. Considering that all data in a row is stored on the same machine, the resources of a single machine in the cluster become a bottleneck. Indeed, if only the predicate in a triple pattern is given, we can use the HBase Filter API to answer this request with a table scan on T_{s_po} or T_{o_ps} using the predicate as column filter. Table 2 shows the mapping of every possible triple pattern to the corresponding HBase table. Overall, experiments on our cluster showed that the two table schema with server side filters has similar performance characteristics compared to the six table schema but uses only one third of storage space.

Table 2. SPARQL triple pattern mapping using HBase predicate push-down filters

pattern	**table**	**filter**
(s, p, o)	T_{s_po} or T_{o_ps}	column & value
(?s, p, o)	T_{o_ps}	column
(s, ?p, o)	T_{s_po} or T_{o_ps}	value
(s, p, ?o)	T_{s_po}	column
(?s, ?p, o)	T_{o_ps}	
(?s, p, ?o)	T_{s_po} or T_{o_ps} (table scan)	column
(s, ?p, ?o)	T_{s_po}	
(?s, ?p, ?o)	T_{s_po} or T_{o_ps} (table scan)	

Our experiments also revealed some fundamental scaling limitations of the storage schema caused by the T_{o_ps} table. In general, an RDF dataset uses a relatively small number of classes but contains many triples that link resources to classes, e.g. (Alex, rdf:type, foaf:Person). Thus, using the object of a triple as row key means that all resources of the same class will be stored in the same row. With increasing dataset size these rows become very large and exceed the configured maximum region size resulting in overloaded regions that contain only a single row. Since HBase cannot split these regions the resources of a single machine become a bottleneck for scalability. To circumvent this problem we use a modified T_{o_ps} row key design for triples with predicate rdf:type. Instead of using the object as row key we use a compound row key of object and subject, e.g. (foaf:Person|Alex). As a result, we can not access all resources of a class with a single table lookup but as the corresponding rows will be consecutive in T_{o_ps} we can use an efficient range scan starting at the first entry of the class.

4 MAPSIN Join

The major task in SPARQL query evaluation is the computation of joins between triple patterns, i.e. basic graph patterns. However, join processing on large RDF datasets, especially if it involves more than two triple patterns, is challenging [20]. Our approach combines the scalable storage capabilities of NoSQL data stores (i.e. HBase), that suffer from a suitable distributed processing layer, with MapReduce, a highly scalable and distributed computation framework, which in turn does not support appropriate storage structures for large scale join processing. This allows us to catch up with the flexibility of reduce-side joins while utilizing the effectiveness of a map-side join without any changes to the underlying frameworks.

First, we introduce the needed SPARQL terminology analogous to [23]: Let V be the infinite set of query variables and T be the set of valid RDF terms.

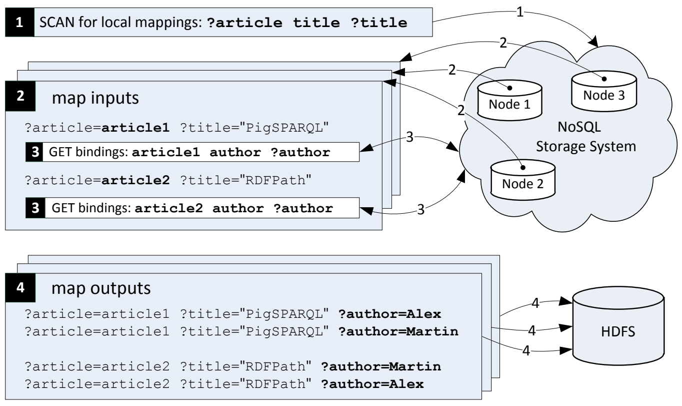
Definition 1. *A (solution) mapping μ is a partial function $\mu : V \rightarrow T$. We call $\mu(?v)$ the variable binding of μ for $?v$. Abusing notation, for a triple pattern p we call $\mu(p)$ the triple pattern that is obtained by substituting the variables in p according to μ. The domain of μ,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

Definition 2. *Two mappings μ_1, μ_2 are compatible if, 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.*

4.1 Base Case

To compute the join between two triple patterns, $p_1 \bowtie p_2$, we have to merge the compatible mappings for p_1 and p_2. Therefore, it is necessary that subsets of both multisets of mappings are brought together such that all compatible mappings can be processed on the same machine.

Our MAPSIN join technique computes the join between p_1 and p_2 in a single map phase. At the beginning, the map phase is initialized with a parallel distributed HBase table scan for the first triple pattern p_1 where each machine retrieves only those mappings that are locally available. This is achieved by utilizing a mechanism for allocating local records to map functions, which is supported by the MapReduce input format for HBase. The map function is invoked for each retrieved mapping μ_1 for p_1. To compute the partial join between p_1 and p_2 for the given mapping μ_1, the map function needs to retrieve those mappings for p_2 that are compatible to μ_1 based on the shared variables between p_1 and p_2. At this point, the map function utilizes the input mapping μ_1 to substitute the shared variables in p_2, i.e. the join variables. The substituted triple pattern p_2^{sub} is then used to retrieve the compatible mappings with a table lookup in HBase following the triple pattern mapping outlined in Table 2. Since there is no guarantee that the corresponding HBase entries reside on the same machine, the results of the request have to be transferred over the network in general. However, in contrast to a reduce-side join approach where a lot of data is transferred over the network, we only transfer the data that is really needed. Finally, the computed multiset of mappings is stored in HDFS.



1 SCAN for local mappings: ?article title ?title
2 map inputs
?article=article1 ?title="PigSPARQL"
3 GET bindings: article1 author ?author
?article=article2 ?title="RDFPath"
3 GET bindings: article2 author ?author
Node 1
Node 2
Node 3
NoSQL Storage System
4 map outputs
?article=article1 ?title="PigSPARQL" ?author=Alex
?article=article1 ?title="PigSPARQL" ?author=Martin
?article=article2 ?title="RDFPath" ?author=Martin
?article=article2 ?title="RDFPath" ?author=Alex
HDFS

Fig. 2. MAPSIN join base case for the first two triple patterns of query in Figure 1

Figure 2 is an example for the base case of our MAPSIN join that illustrates the join between the first two triple patterns of the SPARQL query in Figure 1. While the mappings for the first triple pattern ($?article$, title, $?title$) are retrieved locally using a distributed table scan (step 1+2), the compatible mappings for ($?article$, author, $?author$) are requested within the map function (step 3) and the resulting set of mappings is stored in HDFS (step 4).

4.2 Cascading Joins

Chains of concatenated triple patterns require some slight modifications to the previously described base case. To compute a query of at least three triple patterns we have to process several joins successively, e.g. $p_1 \bowtie p_2 \bowtie p_3$. The processing of the first two patterns $p_1 \bowtie p_2$ correspond to the base case and the results are stored in HDFS. The additional triple pattern p_3 is then joined with the mappings for $p_1 \bowtie p_2$. To this end, an additional map-phase (without any intermediate shuffle or reduce phase) is initialized with the previously computed mappings as input. Since these mappings reside in HDFS, they are retrieved locally in parallel such that the map function gets invoked for each mapping μ_2 for $p_1 \bowtie p_2$. The compatible mappings for p_3 are retrieved using the same strategy as for the base case, i.e. μ_2 is used to substitute the shared variables in p_3 and compatible mappings are retrieved following the triple pattern mapping outlined in Table 2. Algorithm 1 outlines one iteration of the MAPSIN join. The input for the map function contains either a mapping for the first triple pattern (via distributed table scan) or a mapping for previously joined triple patterns (loaded from HDFS).

Algorithm 1: MAPSIN join: **map**(inKey, inValue)

```
input : inKey, inValue: value contains input mapping, key can be ignored
output: multiset of mappings
$p_{n+1} \leftarrow$ Config.getNextPattern()
$\mu_n \leftarrow inValue$.getInputMapping()
$\Omega_{n+1} \leftarrow \emptyset$
if $dom(\mu_n) \cap dom(p_{n+1}) \neq \emptyset$ then
    // substitute shared vars in $p_{n+1}$
    $p^{sub}_{n+1} \leftarrow \mu_n(p_{n+1})$
    $results \leftarrow$ HBase.GET($p^{sub}_{n+1}$) // table index lookup using substituted pattern
else
    $results \leftarrow$ HBase.GET($p_{n+1}$) // table index lookup using unsubstituted pattern
end
if $results \neq \emptyset$ then
    // merge $\mu_n$ with compatible mappings for $p_{n+1}$
    foreach mapping $\mu$ in results do
        $\mu_{n+1} \leftarrow \mu_n \cup \mu$
        $\Omega_{n+1} \leftarrow \Omega_{n+1} \cup \mu_{n+1}$
    end
    emit($null$, $\Omega_{n+1}$) // key is not used since there is no reduce phase
end
```

4.3 Multiway Join Optimization

Instead of processing concatenated triple patterns successively as a sequence of two-way joins, some basic graph patterns allow to apply a multiway join approach to process joins between several concatenated triple patterns at once in a single map phase. This is typically the case for star pattern queries where triple patterns share the same join variable. The SPARQL query introduced in Section 2.1 is an example for such a query as all triple patterns share the same join variable ?*article*. This query can be processed by a three-way join in a single map-phase instead of two consecutive two-way joins.

We extended our approach to support this multiway join optimization. Again, the first triple pattern p_1 is processed using a distributed table scan as input for the map phase. But instead of using a sequence of n map phases to compute $p_1 \bowtie p_2 \bowtie ... \bowtie p_{n+1}$ we use a single map phase thus saving $n-1$ MapReduce iterations. Hence, the map function needs to retrieve all mappings for $p_2, p_3, ..., p_{n+1}$ that are compatible to the input mapping μ_1 for p_1. Therefore, the join variable $?v_s$ in $p_2, p_3, ..., p_{n+1}$ (e.g. ?*article*) is substituted with the corresponding variable binding $\mu_1(?v_s)$. The substituted triple patterns $p_2^{sub}, p_3^{sub}, ..., p_{n+1}^{sub}$ are then used to retrieve the compatible mappings using HBase table lookups. This general case of the MAPSIN multiway join is outlined in Algorithm 2.

Algorithm 2: MAPSIN multiway join: **map**(inKey, inValue)

```
input  : inKey, inValue: value contains input mapping, key can be ignored
output: multiset of mappings
#p ← Config.getNumberOfMultiwayPatterns()
$\mu_n$ ← inValue.getInputMapping()
$\Omega_n$ ← {$\mu_n$}
// iterate over all subsequent multiway patterns
for i ← 1 to #p do
    $\Omega_{n+i}$ ← ∅
    $p_{n+i}$ ← Config.getNextPattern()
    // substitute shared vars in $p_{n+i}$
    $p_{n+i}^{sub}$ ← $\mu_n(p_{n+i})$
    results ← HBase.GET($p_{n+i}^{sub}$) // table index lookup using substituted pattern
    if results ≠ ∅ then
        // merge previous mappings with compatible mappings for $p_{n+i}$
        foreach mapping μ in results do
            foreach mapping μ' in $\Omega_{n+i-1}$ do
                $\Omega_{n+i} \leftarrow \Omega_{n+i} \cup (\mu \cup \mu')$
            end
        end
    else
        // no compatible mappings for $p_{n+i}$ hence join result for $\mu_n$ is empty
        return
    end
end
emit(null, $\Omega_{n+\#p}$) // key is not used since there is no reduce phase
```

The performance of MAPSIN joins strongly correlates with the number of index lookups in HBase. Hence, minimizing the number of lookups is a crucial point for optimization. In many situations, it is possible to reduce the number of requests by leveraging the RDF schema design for HBase outlined in Section 3. If the join variable for all triple patterns is always on subject or always on object position, then all mappings for $p_2, p_3, ..., p_{n+1}$ that are compatible to the input mapping μ_1 for p_1 are stored in the same HBase table row of T_{s_po} or T_{o_ps}, respectively, making it possible to use a single instead of n subsequent table lookups. Hence, all compatible mappings can be retrieved at once thus saving $n-1$ lookups for each invocation of the map function. Due to space limitations the corresponding algorithm for this optimized case can be found in the technical report version of this paper [24].

5 Evaluation

The evaluation was performed on a cluster of 10 Dell PowerEdge R200 servers equipped with a Dual Core 3.16 GHz CPU, 8 GB RAM, 3 TB disk space and connected via gigabit network. The software installation includes Hadoop 0.20.2, HBase 0.90.4 and Java 1.6.0 update 26.

Table 3. SP^{2}Bench & LUBM loading times for tables T_{s_po} and T_{o_ps} (hh:mm:ss)

SP^{2}Bench	**200M**	**400M**	**600M**	**800M**	**1000M**
# RDF triples	∼ 200 million	∼ 400 million	∼ 600 million	∼ 800 million	∼ 1000 million
T_{s_po}	00:28:39	00:45:33	01:01:19	01:16:09	01:33:47
T_{o_ps}	00:27:24	01:04:30	01:28:23	01:43:36	02:19:05
total	00:56:03	01:50:03	02:29:42	02:59:45	03:52:52
LUBM	**1000**	**1500**	**2000**	**2500**	**3000**
# RDF triples	∼ 210 million	∼ 315 million	∼ 420 million	∼ 525 million	∼ 630 million
T_{s_po}	00:28:50	00:42:10	00:52:03	00:56:00	01:05:25
T_{o_ps}	00:48:57	01:14:59	01:21:53	01:38:52	01:34:22
total	01:17:47	01:57:09	02:13:56	02:34:52	02:39:47

We used the well-known Lehigh University Benchmark (LUBM) [13] as the queries can easily be formulated as SPARQL basic graph patterns. Furthermore, we also considered the SPARQL-specific SP^{2}Bench Performance Benchmark [27]. However, because most of the SP^{2}Bench queries are rather complex queries that use all different kinds of SPARQL 1.0 operators, we only evaluated some of the queries as the focus of our work is the efficient computation of joins, i.e. basic graph patterns. Both benchmarks offer synthetic data generators that can be used to generate arbitrary large datasets. For SP^{2}Bench we generated datasets from 200 million up to 1000 million triples. For LUBM we generated datasets from 1000 up to 3000 universities and used the WebPIE inference engine for Hadoop [29] to pre-compute the transitive closure. The loading times for both tables T_{s_po} and T_{o_ps} as well as all datasets are listed in Table 3.

The goal of our approach was to optimize MapReduce based join computation for selective queries. Therefore, we compared our MAPSIN join approach with the reduce-side join based query execution in PigSPARQL [26], a SPARQL 1.0 engine built on top of *Pig*. Pig is an Apache top-level project developed by Yahoo! Research that offers a high-level language for the analysis of very large datasets with Hadoop MapReduce. The crucial point for this choice was the sophisticated and efficient reduce-side join implementation of Pig [11] that incorporates sampling and hash join techniques which makes it a challenging candidate for comparison. We illustrate the performance comparison of PigSPARQL and MAPSIN for some selected LUBM queries that represent the different query types in Figure 3. Our proof-of-concept implementation is currently limited to a maximum number of two join variables as the goal was to demonstrate the feasibility of the approach for selective queries rather than supporting all possible BGP constellations. For detailed comparison, the runtimes of all executed queries are listed in Table 4.

LUBM queries Q1, Q3, Q5, Q11, Q13 as well as SP^{2}Bench query Q3a demonstrate the base case with a single join between two triple patterns (cf. Figure 3a). For the LUBM queries, MAPSIN joins performed 8 to 13 times faster compared to the reduce-side joins of PigSPARQL. Even for the less selective SP^{2}Bench query, our MAPSIN join required only one third of the PigSPARQL execution time. Furthermore, the performance gain increases with the size of the dataset for both LUBM and SP^{2}Bench.

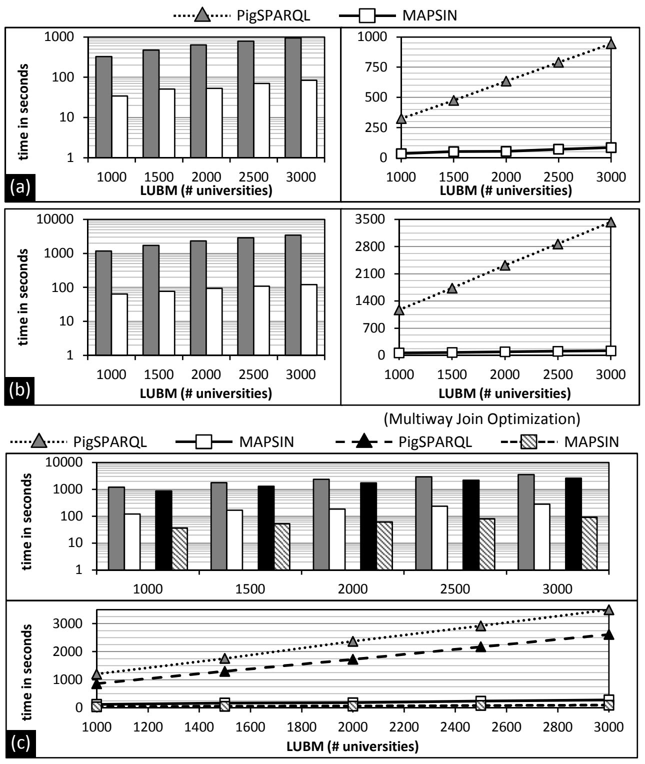


Fig. 3. Performance comparison for LUBM Q1 (a), Q8 (b), Q4 (c)

LUBM queries Q4 (5 triple patterns), Q7 (4 triple patterns), Q8 (5 triple patterns) and SP^{2}Bench queries Q1 (3 triple patterns), Q2 (9 triple patterns) demonstrate the more general case with a sequence of cascaded joins (cf. Figure 3b). In these cases, MAPSIN joins perform even up to 28 times faster than PigSPARQL for LUBM queries and up to 12 times faster for SP^{2}Bench queries.

Of particular interest are queries Q4 of LUBM and Q1, Q2 of SP^{2}Bench since these queries support the multiway join optimization outlined in Section 4.3 as all triple patterns share the same join variable. This kind of optimization is also supported by PigSPARQL such that both approaches can compute the query results with a single multiway join (cf. Figure 3c). The MAPSIN multiway join optimization improves the basic MAPSIN join execution time by a factor of 1.4 (SP^{2}Bench Q1) to 3.3 (LUBM Q4), independently of the data size. For the LUBM queries, the MAPSIN multiway join optimization performs 19 to 28 times faster than the reduce-side based multiway join implementation of PigSPARQL. For the more complex SP^{2}Bench queries, the performance improvements degrade to a factor of approximately 8.5.

The remaining queries (LUBM Q6, Q14 and SP^{2}Bench Q10) consist of only one single triple pattern. Consequently they do not contain a join processing

Table 4. Query execution times for PigSPARQL (P) and MAPSIN (M) in seconds

LUBM	**1000**		**1500**		**2000**		**2500**		**3000**	
	P	M	P	M	P	M	P	M	P	M
Q1	324	34	475	51	634	53	790	70	944	84
Q3	324	33	480	42	642	49	805	59	961	72
Q4	1202	121	1758	167	2368	182	2919	235	3496	279
Q4 MJ	861	37	1297	53	1728	62	2173	81	2613	92
Q5	329	33	484	44	640	53	800	66	955	80
Q6	149	48	214	60	284	69	355	84	424	104
Q7	1013	62	1480	68	1985	93	2472	114	2928	123
Q8	1172	64	1731	77	2318	33	2870	108	3431	121
Q11	319	33	469	46	620	53	780	69	931	79
Q13	325	44	482	72	645	84	800	108	957	128
Q14	149	43	214	70	288	79	364	89	434	107

SP^{2}Bench	**200M**		**400M**		**600M**		**800M**		**1000M**	
	P	M	P	M	P	M	P	M	P	M
Q1	545	58	1026	118	1527	153	2018	177	2519	214
Q1 MJ	310	42	600	87	896	118	1187	154	1476	174
Q2 MJ	1168	241	2341	444	3514	671	4745	834	6005	999
Q3a	227	70	435	139	641	178	845	235	1050	274
Q10	99	40	174	84	254	111	340	151	414	167

step and illustrate primarily the advantages of the distributed HBase table scan compared to the HDFS storage access of PigSPARQL. Improvements are still present but less significant, resulting in an up to 5 times faster query execution.

An open issue of the evaluation remains the actual data flow between HBase and MapReduce as HBase is like a black box where data distribution and partitioning is handled by the system automatically. Since data locality is an important aspect of distributed systems, it is crucial to examine additional measures for future optimizations.

Overall, the MAPSIN join approach clearly outperforms the reduce-side join based query execution for selective queries. Both approaches reveal a linear scaling behavior with the input size but the slope of the MAPSIN join is much smaller. Especially for LUBM queries, MAPSIN joins outperform reduce-side joins by an order of magnitude as these queries are generally rather selective. Moreover, the application of the multiway join optimization results in a further significant improvement of the total query execution times.

6 Related Work

Single machine RDF systems like *Sesame* [6] and *Jena* [31] are widely-used since they are user-friendly and perform well for small and medium sized RDF datasets. *RDF-3X* [21] is considered one of the fastest single machine RDF systems in terms of query performance that vastly outperforms previous single machine systems but performance degrades for queries with unbound objects and low selectivity factor [17]. Furthermore, as the amount of RDF data continues to grow, it will become more and more difficult to store entire datasets on a single machine due to the limited scaling capabilities [16]. One possible approach are specialized clustered RDF systems like OWLIM [19], YARS2 [15] or 4store [14].

However, these systems require a dedicated infrastructure and pose additional installation and management overhead. In contrast, our approach builds upon the idea to use existing infrastructures that are well-known and widely used. As we do not require any changes to Hadoop and HBase at all, it is possible to use any existing Hadoop cluster or cloud service (e.g. Amazon EC2) out of the box.

There is a large body of work dealing with join processing in MapReduce considering various aspects and application fields [4,5,18,20,22,25,32]. In Section 2.2 we briefly outlined the advantages and drawbacks of the general-purpose reduce-side and map-side (merge) join approaches in MapReduce. In addition to these general-purpose approaches there are several proposals focusing on certain join types or optimizations of existing join techniques for particular application fields. In [22] the authors discussed how to process arbitrary joins (theta joins) using MapReduce, whereas [4] focuses on optimizing multiway joins. However, in contrast to our MAPSIN join, both approaches process the join in the reduce phase including a costly data shuffle phase. *Map-Reduce-Merge* [32] describes a modified MapReduce workflow by adding a merge phase after the reduce phase, whereas *Map-Join-Reduce* [18] 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 [9] 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 [3] is a hybrid of MapReduce and DBMS where MapReduce is the communication layer above multiple single node DBMS. The authors in [16] adopt this hybrid approach for the semantic web using RDF-3X. However, the initial graph partitioning is done on a single machine and has to be repeated if the dataset is updated or the number of machines in the cluster change. As we use HBase as underlying storage layer, additional machines can be plugged in seamlessly and updates are possible without having to reload the entire dataset.

HadoopRDF [17] 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. Our MAPSIN join does not use any shuffle or reduce phase at all even in consecutive iterations.

7 Conclusion

In this paper we introduced the Map-Side Index Nested Loop join (MAPSIN join) which combines the advantages of NoSQL data stores like HBase with the well-known and approved distributed processing facilities of MapReduce. In general, map-side joins are more efficient than reduce-side joins in MapReduce as there is no expensive data shuffle phase involved. However, current map-side join approaches suffer from strict preconditions what makes them hard to apply in general, especially in a sequence of joins. The combination of HBase and MapReduce allows us to cascade a sequence of MAPSIN joins without having to sort and repartition the intermediate output for the next iteration. Furthermore, with the multiway join optimization we can reduce the number of MapReduce iterations and HBase requests. Using an index to selectively request only those data that is really needed also saves network bandwidth, making parallel query execution more efficient. The evaluation with the LUBM and SP^{2}Bench benchmarks demonstrate the advantages of our approach compared to the commonly used reduce-side join approach in MapReduce. For selective queries, MAPSIN join based SPARQL query execution outperforms reduce-side join based execution by an order of magnitude while scaling very smoothly with the input size. Lastly, our approach does not require any changes to Hadoop and HBase at all. Consequently, MAPSIN joins can be run on any existing Hadoop infrastructure and also on an instance of Amazon's Elastic Compute Cloud (EC2) without additional installation or management overhead.

In our future work, we will investigate alternatives and improvements of the RDF storage schema for HBase and incorporate MAPSIN joins into PigSPARQL in a hybrid fashion such that the actual join method is dynamically selected based on pattern selectivity and statistics gathered at data loading time.

References

1. RDF Primer. W3C Recom. (2004), `http://www.w3.org/TR/rdf-primer/`
2. SPARQL Query Language for RDF. W3C Recom. (2008), `http://www.w3.org/TR/rdf-sparql-query/`
3. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. PVLDB 2(1), 922–933 (2009)
4. Afrati, F.N., Ullman, J.D.: Optimizing Multiway Joins in a Map-Reduce Environment. IEEE Trans. Knowl. Data Eng. 23(9), 1282–1298 (2011)
5. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A Comparison of Join Algorithms for Log Processing in MapReduce. In: SIGMOD (2010)
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: ISWC (2002)
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems (TOCS) 26(2), 4 (2008)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 51(1), 107–113 (2008)
9. Dittrich, J., Quiané-Ruiz, J.A., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). PVLDB 3(1), 518–529 (2010)
10. Franke, C., Morin, S., Chebotko, A., Abraham, J., Brazier, P.: Distributed Semantic Web Data Management in HBase and MySQL Cluster. In: IEEE International Conference on Cloud Computing (CLOUD). pp. 105 –112 (2011)
11. Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. PVLDB 2(2) (2009)

12. Ghemawat, S., Gobioff, H., Leung, S.: The Google File System. In: ACM SIGOPS Operating Systems Review. vol. 37, pp. 29–43. ACM (2003)
13. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics 3(2) (2005)
14. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: SSWS. pp. 94–109 (2009)
15. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. The Semantic Web (2007)
16. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4(11), 1123–1134 (2011)
17. Husain, M.F., McGlothlin, J.P., Masud, M.M., Khan, L.R., Thuraisingham, B.M.: Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. IEEE TKDE 23(9) (2011)
18. Jiang, D., Tung, A.K.H., Chen, G.: Map-Join-Reduce: Toward Scalable and Efficient Data Analysis on Large Clusters. IEEE TKDE 23(9), 1299–1311 (2011)
19. Kiryakov, A., Ognyanov, D., Manov, D.: OWLIM - A Pragmatic Semantic Repository for OWL. In: WISE Workshops. pp. 182–192 (2005)
20. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel Data Processing with MapReduce: A Survey. SIGMOD Record 40(4), 11–20 (2011)
21. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB 1(1), 647–659 (2008)
22. Okcan, A., Riedewald, M.: Processing Theta-Joins using MapReduce. In: SIGMOD Conference. pp. 949–960 (2011)
23. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems (TODS) 34(3), 16 (2009)
24. Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., Dorner, C., Lausen, G.: Cascading Map-Side Joins over HBase for Scalable Join Processing. Technical Report. CoRR abs/1206.6293 (2012)
25. Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., Lausen, G.: RDFPath: Path Query Processing on Large RDF Graphs with MapReduce. In: ESWC Workshops. pp. 50–64 (2011)
26. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: Mapping SPARQL to Pig Latin. In: Proceedings of the International Workshop on Semantic Web Information Management (SWIM). pp. 4:1–4:8 (2011)
27. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark. In: ICDE. pp. 222–233 (2009)
28. Sun, J., Jin, Q.: Scalable RDF Store Based on HBase and MapReduce. In: ICACTE. vol. 1, pp. 633–636 (2010)
29. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In: ESWC. pp. 213–227 (2010)
30. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. PVLDB 1(1), 1008–1019 (2008)
31. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D.: Efficient RDF Storage and Retrieval in Jena2. In: SWDB. pp. 131–150 (2003)
32. Yang, H.C., Dasdan, A., Hsiao, R.L., Jr., D.S.P.: Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In: SIGMOD (2007)