PigSPARQL: Mapping SPARQL to Pig Latin[*]

Alexander Schätzle
University of Freiburg, Germany
schaetzl@informatik.uni-freiburg.de

Martin Przyjaciel-Zablocki
University of Freiburg, Germany
zablocki@informatik.uni-freiburg.de

Georg Lausen
University of Freiburg, Germany
lausen@informatik.uni-freiburg.de

ABSTRACT

In this paper we investigate the scalable processing of complex SPARQL queries on very large RDF datasets. As underlying platform we use Apache Hadoop, an open source implementation of Google's MapReduce for massively parallelized computations on a computer cluster. We introduce PigSPARQL, a system which gives us the opportunity to process complex SPARQL queries on a MapReduce cluster. To this end, SPARQL queries are translated into Pig Latin, a data analysis language developed by Yahoo! Research. Pig Latin programs are executed by a series of MapReduce jobs on a Hadoop cluster. We evaluate the processing of SPARQL queries by means of PigSPARQL using the SP^{2}Bench, a SPARQL specific performance benchmark and demonstrate that PigSPARQL enables a scalable execution of SPARQL queries based on Hadoop without any additional programming efforts.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; H.2.4 [**Database Management**]: Systems—*Query processing*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications, MapReduce*

1. INTRODUCTION

The amount of available information in the world wide web is increasing rapidly. Unfortunately, most of this information cannot be gathered and processed automatically because its presentation is designed for the processing by humans. To enable the processing by machines, the Resource Description Framework (RDF) [15] has been developed, a standard for representing data in a machine-readable format. SPARQL [24] is the standard query language for RDF recommended by the W3C.

[*]The paper is an extended version of a paper written in German and presented at the German database conference *Datenbanken in Business, Technologie und Web*, 2011 [26].

Because of the increasing size of RDF datasets up to several billions of RDF statements, scalability of query processing becomes an issue. In 2004 Google introduced their so-called *MapReduce* paradigm [5] which allows parallel processing of very large datasets distributed over a computer cluster. Hadoop is the most popular open source implementation of MapReduce. However, developing on the MapReduce level still is technically challenging and sophisticated. Therefore, Yahoo! developed Pig Latin [20] for the analysis of large datasets based on Hadoop that gives the user a simple level of abstraction by providing high-level primitives like Filters and Joins. In 2010 the implementation of Pig Latin for Hadoop, Pig, became an Apache top-level project.

In this paper we shall present PigSPARQL, a translation framework from full SPARQL 1.0 to Pig Latin, which allows a scalable processing of SPARQL queries on a MapReduce cluster without any additional programming efforts. It has been discussed extensively, whether a MapReduce cluster or a parallel database system approach is more promising to achieve efficiency and scalability (e.g. [14, 22]). From this discussion we conclude that in scenarios, which can be characterized by first extracting information from a huge data set, second by transforming and loading the extracted data into a different format, e.g. a relational database, such that further application specific processing is possible, cluster-based parallelism seems to outperform parallel databases. For such ETL-based applications, which we believe are typical for information processing of web data, PigSPARQL offers not only a declarative way of specifying the transformation part, but also a scalable implementation of the whole ETL-process on a MapReduce cluster. The major contributions of this paper are as follows.

- We describe a translation of SPARQL into an equivalent Pig Latin program that supports all SPARQL 1.0 operators. To the best of our knowledge, this is the first comprehensive presentation of a full translation from SPARQL to Pig Latin. In comparison to a direct mapping into MapReduce, our approach is much easier to achieve and handle. In particular, our approach is able to profit from all optimizations of Pig provided by the developer community and is independent of any changes to the Hadoop framework.
- We provide an implementation of our translation that applies several optimization strategies that have been confirmed to be effective.
- We evaluate PigSPARQL using a SPARQL specific performance benchmark and demonstrate scalability of the processing of SPARQL queries using PigSPARQL.

The rest of the paper is organized as follows: In Chapter 2 we briefly introduce the necessary notions of RDF, SPARQL, MapReduce and Pig Latin. In Chapter 3 we describe the translation from SPARQL to Pig Latin. In Chapter 4 we present the results of the evaluation, Chapter 5 gives an overview of related work and Chapter 6 summarizes our work.

2. FRAMEWORK

Due to space limitations we can only give a short introduction to RDF, SPARQL, MapReduce and Pig Latin.

2.1 RDF

RDF [15] is a standard format for modeling knowledge about arbitrary resources, e.g. persons or documents. An RDF dataset consists of a set of so-called RDF triples in the form (*subject, predicate, object*) that can be interpreted as "subject has property predicate with value object". URIs (Uniform Resource Identifier) are globally unique identifiers used to represent resources in RDF (e.g. URLs are a subset of URIs). For clarity of presentation, we use a simplified RDF notation without URI prefixes in the following. It is possible to represent an RDF dataset as directed, labeled graph where every triple corresponds to an edge (predicate) from the subject to the object. Figure 1 shows an RDF graph consisting of ten RDF triples.

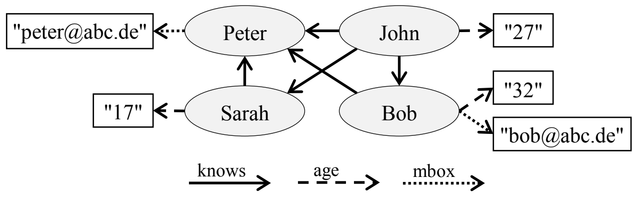

Figure 1: RDF graph

2.2 SPARQL

SPARQL is the W3C recommended query language for RDF [24]. A formal definition of the SPARQL semantics can also be found in [23]. 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 basis of all graph patterns are the so-called *Triple Patterns*. A Triple Pattern is an RDF triple where subject, predicate and object can be variables ($?var$), e.g. ($?s$, p, $?o$). A set of Triple Patterns concatenated by AND (.) is called a *Basic Graph Pattern* (BGP). A SPARQL graph pattern can be defined recursively as follows:

- A BGP is a graph pattern.
- If P, P' are graph pattern, then $\{P\}.\{P'\}$, $\{P\}$ UNION $\{P'\}$ and $\{P\}$ OPTIONAL $\{P'\}$ are also graph pattern.
- If P is a graph pattern and R is a filter condition, then P FILTER (R) is also a graph pattern.
- If P is a graph pattern, u an URI and $?v$ a variable, then GRAPH u $\{P\}$ and GRAPH $?v$ $\{P\}$ are also graph pattern.

FILTER can be used to restrict the values of variables and OPTIONAL allows to add additional information to the result of a query. If the desired information does not exist, the optional variables remain *unbound* in the query result. UNION can be used to define two alternative graph patterns where the query results must match at least one of the patterns. A SPARQL query can also address several RDF graphs by using the GRAPH operator. The following example shows a simple SPARQL query that gives all persons who know 'Peter' and are at least 18 years old together with their mailboxes, if they exist. Executed on the RDF graph of Figure 1 the query would give two results for 'John' and 'Bob' where only 'Bob' has a known email address.

Example SPARQL Query

```
SELECT *
WHERE {
  { ?person knows Peter . ?person age ?age }
  OPTIONAL { ?person mbox ?mb }
  FILTER (?age >= 18)
}
```

2.3 MapReduce

The MapReduce programming model was originally introduced by Google in 2004 [5] and 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 [8] 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 jobs each consisting of a *Map* and a *Reduce* phase separated by a so-called *Shuffle & Sort* phase. A user has to implement the *map* and *reduce* functions which are automatically executed in parallel on a portion of the data. The Mappers invoke the map function for every record of their input dataset represented as a key-value pair. The map function outputs a list of new intermediate key-value pairs which are then sorted according to their key and distributed to the Reducers such that all values with the same key are sent to the same Reducer. The reduce function is invoked for every distinct key together with a list of all according values and outputs a list of values which can be used as input for the next MapReduce job. The signatures of the map and reduce functions are therefore as follows.

```
map:    (inKey, inValue) -> list(outKey, tmpValue)
reduce: (outKey, list(tmpValue)) -> list(outValue)
```

2.4 Pig Latin

Pig Latin [20] is a language for the analysis of very large datasets developed by Yahoo! Research. It is based on the well-known Apache Hadoop Framework, an open source implementation of Google's MapReduce. The implementation of Pig Latin for Hadoop, *Pig*, is an Apache top-level project that automatically translates a Pig Latin program into a series of MapReduce jobs.

Data model.

Pig Latin has a fully nested data model which allows more flexibility than the flat tables required by the first normal form in relational databases. The data model of Pig Latin provides four different types:

- **Atom**: An atom contains a simple atomic value like a string or number, e.g. `'Sarah'` or 24.
- **Tuple**: A tuple is a sequence of fields of any type. Every field can have a name (alias) that can be used to reference the field, e.g. (`'John'`, `'Doe'`) with alias (*firstname*, *lastname*).
- **Bag**: A bag is a collection of tuples with possible duplicates. The schemas of the tuples do not have to match, i.e. the number and types of fields can differ.

$$\left\{ \begin{array}{c} (\texttt{'Bob'}, \texttt{'Sarah'}) \\ (\texttt{'Peter'}, (\texttt{'likes'}, \texttt{'football'})) \end{array} \right\}$$

- **Map**: A map is a collection of data items where each item can be looked up by an associated key.

$$\left[\begin{array}{c} \texttt{'name'} \rightarrow \texttt{'John'} \\ \texttt{'knows'} \rightarrow \left\{ \begin{array}{c} (\texttt{'Sarah'}) \\ (\texttt{'Bob'}) \end{array} \right\} \end{array} \right]$$

Operators.

A Pig Latin program consists of a sequence of instructions where each instruction performs a single data transformation. We shortly introduce those Pig Latin operators that we used for our translation. The interested reader can find a more detailed description of Pig Latin in [2].

LOAD deserializes the input data and maps it to the data model of Pig Latin. The user can implement an *User Defined Function* (UDF) that defines how to map an input tuple to a Pig Latin tuple as shown in the following example. The result of LOAD is a bag of tuples. For example,

```
people = LOAD 'input' USING myLoad() AS (name,age);
```

FOREACH can be used to apply some processing on every tuple of a bag. It can also be used for projection or adding new fields to a tuple. For example,

```
A = FOREACH people GENERATE name,
    age>=18? 'adult':'minor' AS type;
```

FILTER allows to remove unwanted tuples of a bag, e.g.

```
B = FILTER people BY age >= 18;
```

[OUTER] JOIN performs an equi or outer join between bags. It can also be applied to more than two bags at once (multi join). For example,

```
C = JOIN A BY name [LEFT OUTER], B BY name;
```

UNION can be used to combine two or more bags. Unlike relational databases, the schemas of the tuples do not have to match although this is not recommended in general since the schema information, especially the alias names of the fields, is lost in such cases. For example,

```
D = UNION B,C;
```

SPLIT partitions a bag into two or more bags that do not have to be distinct or complete, i.e. tuples can end up in more than one partition or no partition at all, e.g.

```
SPLIT people INTO E IF age<18, F IF age>=21;
```

3. TRANSLATION

For translating SPARQL to Pig Latin we follow a standard approach which centers on an algebraic representation of SPARQL expressions (cf. Figure 2). First, a SPARQL query is parsed to generate an abstract syntax tree which is then translated into a SPARQL algebra tree as described by the W3C documentation [24]. For the syntax and algebra tree generation we used the well-known ARQ[1] engine of the Jena framework. Before translating the resulting algebra expressions into Pig Latin, certain optimizations are applied which will be explained later.

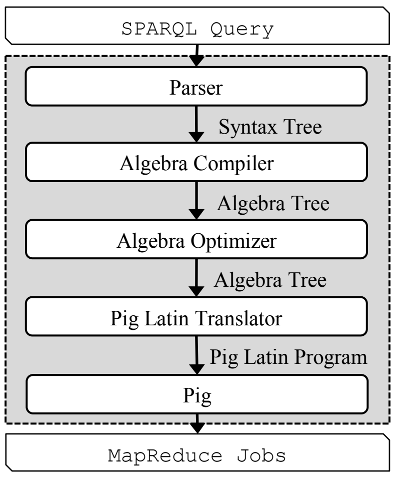

Figure 2: Modular Translation Process

The semantics of a SPARQL query is defined on the algebra level and an expression of the SPARQL algebra can be represented as a tree that is evaluated *bottom-up*. Table 1 shows the operators of the SPARQL algebra and the corresponding syntax expressions.

Table 1: SPARQL Algebra & Syntax

Algebra	Syntax
BGP	Set of Triple patterns concatened via .
Join	Concatenation of two groups $\{\cdots\}.\{\cdots\}$
Filter	FILTER
LeftJoin	OPTIONAL
Union	UNION
Graph	GRAPH

3.1 RDF data mapping

In order to process RDF datasets with Pig Latin, we first have to define how to represent an RDF triple in Pig Latin. An RDF triple is a tuple with three fields that can consist of *URIs* (Uniform Resource Identifier), *RDF literals* or *blank nodes*. Since URIs are strings in a special format, we represent them as atoms in angle brackets (`<URI>`). Simple and typed RDF literals can also be represented as atoms by using a compound value for literals with a language or datatype tag (`"literal"@lang`,`"literal"^^type`). If needed in arithmetic expressions, numeric literals (e.g. literals of type `xsd:integer`) are parsed into the appropriate numeric datatype of Pig Latin at runtime. The RDF syntax does not define an internal structure of blank nodes, they just have to be distinguishable from URIs and literals. Thus, we can also represent them as atoms with a leading underscore (`_:nodeID`). Hence, an RDF triple can be represented as a tuple with three fields of atomic type (`chararray`) with schema (`s:chararray`, `p:chararray`, `o:chararray`).

[1]http://jena.sourceforge.net/ARQ

3.2 Algebra translation

For each operator of the SPARQL algebra we give a translation into a sequence of Pig Latin commands illustrated by a representative example (P1-P6). First, we introduce the needed terminology analogous to [24]: Let V be the infinite set of query variables and T the set of valid RDF terms (URIs, RDF literals, blank nodes).

Definition 1. A solution mapping μ is a partial function $\mu : V \rightarrow T$. We call $\mu(?v)$ the variable binding of μ for $?v$. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. The result of a SPARQL query is a multiset of solution mappings Ω.

Definition 2. Two solution 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 $\mu_1 \cup \mu_2$ is also a solution mapping and solution mappings with disjoint domains are always compatible.

Basic Graph Pattern (BGP). BGPs are the basis of all SPARQL queries as it is the only operator that is evaluated directly on the underlying RDF data. The result of a BGP is a multiset of solution mappings that serves as input for other operators. Solution Mappings can be represented in Pig Latin as a (flat) bag where each tuple is a single solution mapping and the fields of a tuple correspond to the variable bindings of that tuple. This bag can be seen as a table where the rows are the solution mappings and the columns are the corresponding variable bindings.

The corresponding Pig Latin program for P1 consists of a LOAD (1), followed by several FILTER/FOREACH (2) and several JOIN/FOREACH (3) statements.

P1. *Persons who know 'Bob' with age and mailbox.*

```
SP  BGP(?a knows Bob . ?a age ?b . ?a mbox ?c)
```
```
PL  A = LOAD 'rdf' USING rdf() AS (s,p,o);  (1)
    t1= FILTER A BY p=='knows' AND o=='Bob';(2)
    t1= FOREACH t1 GENERATE s AS a;
    t2= FILTER A BY p=='age';
    t2= FOREACH t2 GENERATE s AS a, o AS b;
    t3= FILTER A BY p=='mbox';
    t3= FOREACH t3 GENERATE s AS a, o AS c;
    j1= JOIN t1 BY a, t2 BY a;              (3)
    j1= FOREACH j1 GENERATE t1::a AS a, b;
    j2= JOIN j1 BY a, t3 BY a;
    P1= FOREACH j2 GENERATE j1::a AS a, b, c;
```

(1) We implemented a loader UDF for RDF data that maps RDF triples to the data model of Pig Latin as described in section 3.1.
(2) For every Triple Pattern we need a FILTER to select those RDF triples of the input that match the pattern. FOREACH is used to remove unnecessary columns (columns that do not correspond to a variable binding) and update the schema information with the names of the variables.
(3) The results of the Triple Patterns are successively joined to compute the final result. If a BGP consists of n Triple Patterns we need $n - 1$ JOINs in general. The predicate of the join is given by the shared variables of both sides, i.e. the join combines the compatible solution mappings. If there are no shared variables we have to compute the cross product.

Filter. A Filter removes those solution mappings from a multiset of solution mappings that do not satisfy the filter expression. A Filter can be directly expressed as FILTER in Pig Latin (cf. P2). To support the SPARQL built-in functions one could implemented them as UDF in Pig Latin.

P2. *Filter P1 for persons with age between 30 and 40.*

```
SP  Filter(?b >= 30 && ?b <= 40, P1)
```
```
PL  P2 = FILTER P1 BY (b >= 30 AND b <= 40);
```

Join. The Join merges the compatible mappings of two multisets of solution mappings. A Join can be expressed as a JOIN in Pig Latin on the shared variables (cf. P3, assuming the results of the BGPs are stored in `BGP1` and `BGP2`). Again, FOREACH is used to remove unnecessary columns and update the schema information as illustrated in the following.

P3. *Persons who know somebody with the same age.*

```
SP  Join(BGP(?a knows ?b),
         BGP(?a age ?c . ?b age ?c))
```
```
PL  j1 = JOIN BGP1 BY (a,b), BGP2 BY (a,b);
    P3 = FOREACH j1 GENERATE
         BGP1::a AS a, BGP1::b AS b, c;
```

LeftJoin. The LeftJoin operator adds additional information to the result, if it exists. This additional information can be restricted by a filter expression. In Pig Latin we can first use a FILTER to restrict the values of the additional information before performing an OUTER JOIN on the shared variables as illustrated in the following.

P4. *Persons with mailbox and optional age (if >=18).*

```
SP  LeftJoin(BGP(?a mbox ?b),
             BGP(?a age ?c), ?c>=18)
```
```
PL  f1= FILTER BGP2 BY c >= 18;
    lj= JOIN BGP1 BY a LEFT OUTER, BGP2 BY a;
    P4= FOREACH lj GENERATE BGP1::a AS a, b, c;
```

Union. The Union operator combines two multisets of solution mappings (Ω_1, Ω_2) to a single multiset without any further changes, i.e. it unifies the results of two graph patterns. The problem of Union is that for two mappings $\mu_1 \in \Omega_1$ and $\mu_2 \in \Omega_2$ it can be that $dom(\mu_1) \neq dom(\mu_2)$ as it is the case for P5 where `?b` is not defined in the second BGP. To have a common schema in Pig Latin we add a new column to the result of the second BGP and use null values to indicate that the variable binding for `?b` is not defined.

P5. *Persons who know 'Bob' and have a mailbox or persons who know 'John'.*

```
SP  Union(BGP(?a knows Bob . ?a mbox ?b),
          BGP(?a knows John))
```
```
PL  BGP2 = FOREACH BGP2 GENERATE a, null as b;
    P5   = UNION BGP1, BGP2;
```

Graph. A SPARQL query dataset is a collection of RDF graphs with one *default graph* and zero or more additional *named graphs*. In general, a graph pattern is applied to the default graph. The Graph operator can be used to apply a pattern to one or all of the named graphs. A named graph is referenced by an unique URI and for each graph that is used in the query we need a pair $(URI, graph)$ that specifies where to find the corresponding RDF graph. If a variable is used in the Graph operator instead of a specific graph URI, the pattern must be applied to all named graphs.

As we want to execute SPARQL queries on large RDF graphs in a MapReduce cluster, all graphs must be stored in the distributed file system. Applying a pattern to one of the named graphs with Pig Latin simply means loading the corresponding data.

P6. *Persons in graph* ***graphURI*** *who know somebody.*

```
SP  Graph(graphURI, BGP(?a knows ?b))
```

```
PL  graph1 = LOAD 'pathToGraphURI'
             USING rdfLoad() AS (s,p,o);
    t1 = FILTER graph1 BY p == 'knows';
    P6 = FOREACH t1 GENERATE s AS a, o AS b;
```

Joins and Null values. As we use flat bags to represent solution mappings in Pig Latin and all tuples of a bag have the same schema we use null values to indicate that a variable is unbound in a solution mapping. This typically occurs when using OPTIONAL to add additional information to a solution mapping. The result of OPTIONAL is a set of solution mappings (i.e. a bag in Pig Latin) where the optional variables can be unbound for some solution mappings (i.e. some tuples of the bag contain null values). However, this is problematic if the further processing of the query requires a join over these possibly unbound variables. In SPARQL an unbound variable is compatible to any other binding of that variable but since Pig Latin follows the relational algebra, a JOIN in Pig Latin is null rejecting.

Assume we have two bags of solution mappings R,S with schemas (A,B) and (B,C) where R can contain null values for variable B as illustrated in the following example.

R

A	B
a_1	b_1
a_2	*null*

$\bowtie_{SPARQL}$

S

B	C
b_1	c_1
b_2	c_2

=

A	B	C
a_1	b_1	c_1
a_2	b_1	c_1
a_2	b_2	c_2

The second tuple of R is compatible to any tuple of S since variable B is unbound. In Pig Latin we would only get one tuple as join result since the second tuple of R will not match with any tuple of S. To get the same result in Pig Latin we split R into two bags (with and without null values) and process them separately, i.e. we perform a normal join for all tuples without null values and a cross product for the tuples with null values.

```
PL SPLIT R INTO R1 IF B is not null,
                R2 IF B is null;
   j1 = JOIN R1 BY B, S BY B;
   j1 = FOREACH j1 GENERATE A, R1::B AS B, C;
   j2 = CROSS R2, S;
   j2 = FOREACH j2 GENERATE A, S::B AS B, C;
   J  = UNION j1, j2;
```

The complexity increases with the number of join variables that can be unbound, e.g. for two possibly unbound join variables we already have to split the bag into four bags (one for every possible combination). Our translator recognizes if a join contains possibly unbound variables and performs the necessary changes to the translation automatically. Fortunately, this situation does not occur in most SPARQL queries. In fact, if a SPARQL query is *well designed* according to [23], there are no joins over unbound variables at all.

3.3 Optimizations

The optimization of SPARQL queries is a subject of current research [11, 28, 29]. As we will demonstrate in the evaluation, optimizing the SPARQL query execution based on Pig Latin means reducing the I/O required to transfer data between Mappers and Reducers as well as the data that is read or stored in the distributed file system.

1. **SPARQL algebra.** We investigated some well-known optimization strategies for the SPARQL algebra to reduce the amount of intemediate results, especially the early execution of Filters and the rearrangement of Triple Patterns by selectivity [29]. We used a fixed scheme without statistical information on the RDF dataset (called *variable counting*) where Triple Patterns with one variable are considered to be more selective than Triple Patterns with two variables and bounded subjects are considered to be more selective than bounded predicates or objects.

2. **Translation.** The early projection of redundant data (*"project early and often"*, e.g. duplicate columns after joins or bounded values that should not occur in the result) as well as the application of multi joins to reduce the number of joins in Pig Latin has proven to be very effective. We can use a multi join if several consecutive joins refer to the same variables. Assume we have three tables (bags) to join (A,B,C) by the common variable ?v. Instead of using two joins we can use a single multi join.
   `JOIN A BY v, B BY v, C BY v;`

3. **Data model.** In a typical SPARQL query the predicate of a Triple Pattern is mostly bounded, i.e. variables are typically used in the subject and object position. Therefore, a *vertical partitioning* [1] of the RDF data by predicates reduces the amount of RDF triples that must be loaded for query execution. This vertical partitioning can be done once in advance using a single MapReduce job and does not cost more disk space. All RDF triples with the same predicate are stored in the same folder and every predicate has its own folder. For further improvements this storage scheme could be extended as the partitioning turned out to be very effective in our evaluation.

3.4 Example

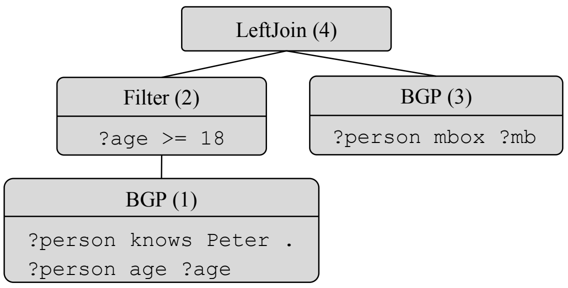

Figure 3: SPARQL algebra tree

Figure 3 shows the algebra tree after optimization (pushing Filter execution before LeftJoin) for the SPARQL query of section 2.2. The tree is traversed bottom-up and translated into the following sequence of Pig Latin commands, assuming a vertical partitioning of the RDF data.

```
PL knows = LOAD 'rdf/knows'                     (1)
           USING rdf() AS (s,o);
   age   = LOAD 'rdf/age' USING rdf() AS (s,o);
   f1 = FILTER knows BY o == 'Peter';
   t1 = FOREACH f1 GENERATE s AS person;
   t2 = FOREACH age GENERATE
         s AS person,o AS age;
   j1 = JOIN t1 BY person, t2 BY person;
   BGP1= FOREACH j1 GENERATE
         t1::person AS person, t2::age AS age;
   F   = FILTER BGP1 BY age >= 18;              (2)
   mbox= LOAD 'rdf/mbox' USING rdf() AS (s,o);(3)
   BGP2= FOREACH mbox GENERATE
         s AS person,o AS mb;
   lj  = JOIN F BY person LEFT OUTER,           (4)
         BGP2 BY person;
   LJ  = FOREACH lj GENERATE F::person AS person,
         F::age AS age, BGP2::mb AS mb;
   STORE LJ INTO 'output' USING resultWriter();
```

4. EVALUATION

We evaluated our implementation on 10 Dell PowerEdge R200 servers connected via a gigabit network. Each server was equipped with a Dual Core 3.16 GHz processor, 4 GB RAM, 1 TB hard disk and Hadoop 0.20.2 as well as Pig 0.5.0 installed. Due to the replication of the distributed file system (HDFS), the actual available payload was 2.5 TB.

We investigated the execution times, the amount of data read from HDFS *(HDFS Bytes Read)*, the amount of data written to HDFS *(HDFS Bytes Written)* and the amount of data that was transferred from Mappers to Reducers *(Reduce Shuffle Bytes)*. We used the SP^{2}Bench [27], a SPARQL specific performance benchmark, which is in our opinion more suited for the evaluation of SPARQL engines than the general LUBM Benchmark [9] which does not consider SPARQL specific operators like OPTIONAL. The SP^{2}Bench data generator was used to produce RDF datasets of up to 1.6 Billion triples based on the DBLP library [13].

4.1 Example Queries

In the following we present the evaluation of three representative SP^{2}Bench queries.

Q3a. *Select all articles with property `swrc:pages`.*

```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property = swrc:pages)
}
```

Q3a. The execution of this query requires only one join but generates a huge amount of intermediate results since the second triple pattern matches all RDF triples. However, we can observe that the output does not contain the filter variable `?property` hence the query can be optimized on algebra level by a filter substitution where the variable is replaced by its value. This optimization reduces the execution time of this query by 70% (a) due to a significant reduction of the Reduce Shuffle Bytes (b). A positive side effect of this optimization is the elimination of the unbounded predicate in the second triple pattern. Thus, using a vertical partitioned dataset, only the two predicates `rdf:type` and `swrc:pages` must be considered which results in a significant reduction of data read from HDFS (Q3a opt+part). The appliance of the filter optimization and the vertical partitioning reduces the execution time of this query by 97%.

Q2. *Extract all inproceedings with the given properties and optional abstract, sorted by the year of publication.*

```
SELECT *
WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
  OPTIONAL { ?inproc bench:abstract ?abstract }
} ORDER BY ?yr
```

Q2. The left side of the OPTIONAL contains a BGP with nine triple patterns that requires (without any optimization) eight joins. In addition, the results should be emitted in a sorted order. Since all eight joins apply to the same variable `?inproc` they can be implemented by a single multi join (Q2 opt). As a result, the number of MapReduce jobs that are necessary for executing Q2 is reduced from twelve to five. The query also benefits from the vertical partitioning (Q2 opt+part) as all predicates are bounded which leads to an overall query execution time reduction of nearly 90% (c).

Q6. *Return, for each year, all publications of persons that have not published in years before.*

```
SELECT ?yr ?name ?doc
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?doc rdf:type ?class .
  ?doc dcterms:issued ?yr .
  ?doc dc:creator ?author .
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document .
    ?doc2 rdf:type ?class2 .
    ?doc2 dcterms:issued ?yr2 .
    ?doc2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2 < ?yr)
  } FILTER (!bound(?author2))
}
```

Q6. This query implements a (closed world) negation by the combination of OPTIONAL and a FILTER for unbounded values. Here, no optimization on the algebra level was possible. As a consequence, the computation of the OPTIONAL produces many intermediate results. In fact, 75% of the aggregated I/O values (diagram (f) of Fig. 4) arise

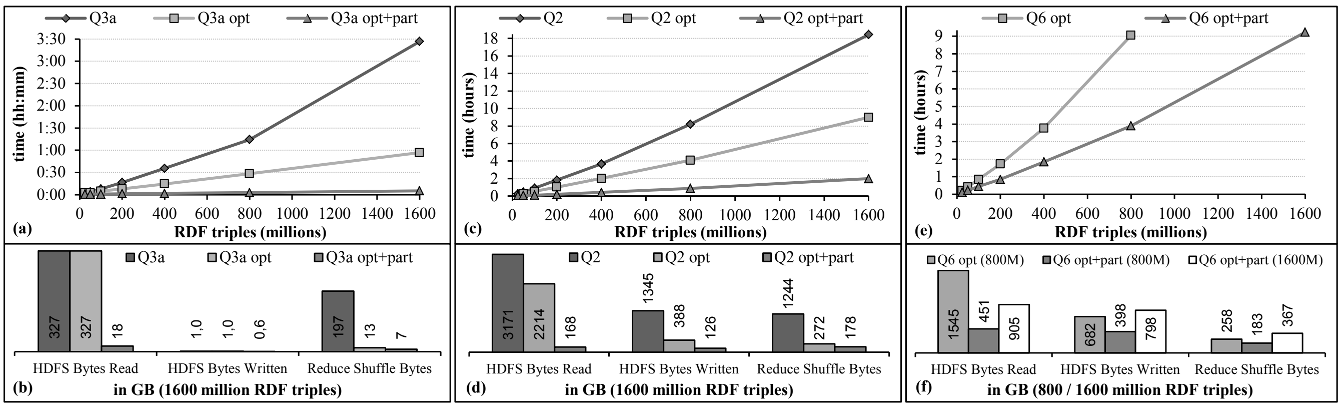

Figure 4: SP^{2}Bench queries Q3a, Q2 and Q6

in a single MapReduce job (computation of OPTIONAL), making the query especially challenging. Only when using a vertical partitioned dataset, the capacity of our cluster was sufficient for executing Q6 with 1600 million triples. Without vertical partitioning, there was not enough local and distributed disk space. To overcome such situation we could make use of the horizontal scalability of MapReduce and simply add more machines to the cluster without any other changes becoming necessary. Note that in Diagram (f) of Fig. 4 we refer to 800 million RDF triples to be able to compare executions with and without vertical partitioning and 1600 million triples for comparison with the other queries.

4.2 Results

As an immediate observation our experiments confirm a nearly linear scalability of the query processing time with respect to the size of the data, a well-known feature of the MapReduce paradigm. This underlines that PigSPARQL indeed is an effective application of the MapReduce paradigm for SPARQL. Our evaluations in particular demonstrate how dramatically PigSPARQL's optimization reduces the amount of data to be handled and the corresponding query processing time. A comparison of PigSPARQL with other technologies, e.g. parallel database approaches and single machine based approaches with respect to efficiency and scalability is a topic of future research. However, we would like to stress that we could observe linear scalability also for query Q6 which might be highly problematic when not executed on a MapReduce cluster. This claim is justified by the observation that in Q6 we first have to compute all publications with respect to all authors before we can find out those authors which have not published in the years before. Therefore, we expect that the sheer amount of intermediate results to be stored and processed will be responsible for a non-linear scalability of competing technologies.

5. RELATED WORK

In [17] a translation from SPARQL to Pig Latin was already mentioned. However, the authors provide no further information or technical details about it. To the best of our knowledge, we present the first detailed and comprehensive translation from SPARQL to Pig Latin that also considers efficient optimizations on different levels and is evaluated with a SPARQL performance benchmark that also contains queries with the SPARQL specific OPTIONAL operator.

The authors in [12] also consider the execution of SPARQL queries based on Hadoop. In contrast to our approach a query is directly mapped into a sequence of MapReduce jobs. They also provide evaluation results for the SP^{2}Bench queries Q1, Q2 and Q3a on a Hadoop cluster of ten nodes similar to our cluster. A comparison of the results confirms that both approaches have a similar performance whereby our implementation is more than 40% faster for Q3a. This demonstrates that our approach based on mapping SPARQL to Pig Latin achieves an execution of SPARQL queries that keeps up with a direct mapping to MapReduce with respect to efficiency if not being more efficient. A direct mapping approach is also proposed in [18]. In contrast to these approaches, our translation supports all SPARQL 1.0 operators and also benefits from further developments of Pig [7] and Hadoop. As we map to Pig Latin, we can expect a greater independence from possible changes inside the underlying MapReduce layer in comparison to a direct mapping.

The execution of SPARQL queries in general plays an important role for the Semantic Web. Sesame [3], Jena [16] and RDF-3X [19] are well-known examples for the execution of SPARQL queries on single machines. Due to the growing amount of available semantic data the evaluation of very large RDF datasets becomes a stronger focus of scientific research. SPIDER [4] uses HBase for storing RDF datasets in Hadoop as flat tables and supports also basic SPARQL queries but the authors do not give detailed information on the supported operators. [25] use UDFs to reduce I/O costs in analytical queries over RDF graphs with Pig Latin. They showed that UDFs can reduce the I/O costs in certain situations which makes this idea also interesting for further improvements of our approach.

Instead of a general MapReduce cluster some RDF stores are built on top of a specialized computer cluster. Virtuoso Cluster Edition [6] and Clustered TDB [21] are cluster extensions of the well-known Virtuoso and Jena RDF stores. 4store [10] is a ready-to-use RDF store which divides the cluster in storage and processing nodes. Nevertheless, the usage of a specialized cluster has the disadvantage that it requires a dedicated infrastructure whereas our approach is based on a general cluster that can be used for different purposes. As our translation does not require any changes to the MapReduce framework or installation, an existing MapReduce cluster can be used out of the box.

6. CONCLUSION

In this paper we proposed PigSPARQL, a new approach for the scalable execution of SPARQL queries geared towards applications which are based on information extraction from very large RDF datasets. For this purpose, we designed and implemented a translation from SPARQL to Pig Latin. The resulting Pig Latin program is translated into a sequence of MapReduce jobs and executed in parallel on a Hadoop cluster. Our evaluation with a SPARQL specific benchmark confirmed that PigSPARQL is well-suited for the scalable execution of SPARQL queries on large RDF datasets with Hadoop. This is also demonstrated by the used dataset size of up to 1.6 billion RDF triples that already exceeds the capabilities of many single machine systems [27]. Taking into account that our Hadoop cluster still is rather small (e.g. Yahoo! maintains Hadoop clusters of several thousand machines) and the relatively early development stage of Pig (the evaluation was performed with Pig 0.5.0) we expect a great future potential of our approach. The proposed translation offers an easy and efficient way to take advantage of the performance and scalability of Hadoop for the distributed and parallelized execution of SPARQL queries on large RDF datasets. As topic for the next future, we plan to compare our MapReduce based system to state-of-the-art single machine SPARQL engines to investigate the advantages and drawbacks of both approaches.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proc. VLDB*, pages 411–422, 2007.

[2] Apache. Pig Latin Reference Manual 1 & 2. http://pig.apache.org/docs/, 2010.

[3] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proc. ISWC*, pages 54–68. Springer, 2002.

[4] H. Choi, J. Son, Y. Cho, M. K. Sung, and Y. D. Chung. SPIDER: A System for Scalable, Parallel/Distributed Evaluation of Large-Scale RDF Data. In *CIKM*, pages 2087–2088, 2009.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] O. Erling and I. Mikhailov. Towards web scale RDF. In *Proc. SSWS*, 2008.

[7] 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. *Proc. VLDB Endow.*, 2:1414–1425, 2009.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. SOSP*, pages 29–43, 2003.

[9] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[10] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered rdf store. In *Proc. SSWS*, page 81, 2009.

[11] O. Hartig and R. Heese. The SPARQL query graph model for query optimization. *The Semantic Web: Research and Applications*, pages 564–578, 2007.

[12] M. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data intensive query processing for large RDF graphs using cloud computing tools. In *Proc. CLOUD*, pages 1–10. IEEE, 2010.

[13] M. Ley. DBLP Bibliography. http://www.informatik.uni-trier.de/ ley/db/, 2010.

[14] J. Lin and C. Dyer. Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[15] F. Manola, E. Miller, and B. McBride. RDF Primer. http://www.w3.org/TR/rdf-primer/, 2004.

[16] B. McBride. Jena: Implementing the RDF Model and Syntax Specification. In *SemWeb*, 2001.

[17] P. Mika and G. Tummarello. Web Semantics in the Clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[18] J. Myung, J. Yeon, and S. Lee. SPARQL basic graph pattern processing with iterative MapReduce. In *Proc. MDAC*, pages 1–6. ACM, 2010.

[19] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. of the VLDB Endowment*, 1(1):647–659, 2008.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. SIGMOD*, pages 1099–1110. ACM, 2008.

[21] A. Owens, A. Seaborne, and N. Gibbins. Clustered TDB: A Clustered Triple Store for Jena. 2008.

[22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. SIGMOD*, pages 165–178. ACM, 2009.

[23] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009.

[24] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2006.

[25] P. Ravindra, V. Deshpande, and K. Anyanwu. Towards scalable RDF graph analytics on MapReduce. In *Proc. MDAC*, pages 1–6. ACM, 2010.

[26] A. Schätzle, M. Przyjaciel-Zablocki, T. Hornung, and G. Lausen. PigSPARQL: Übersetzung von SPARQL nach PigLatin. In *Proc. BTW*, pages 65–84, 2011.

[27] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. In *Proc. ICDE*, pages 222–233, 2009.

[28] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *Proc. ICDT*, pages 4–33, 2010.

[29] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proc. WWW*, pages 595–604. ACM, 2008.