SPARQling Pig – Processing Linked Data with Pig Latin

Stefan Hagedorn[1], Katja Hose[2], Kai-Uwe Sattler[1]

[1] Technische Universität Ilmenau, Ilmenau, Germany
{first.last}@tu-ilmenau.de
[2] Aalborg University, Aalborg, Denmark
khose@cs.aau.dk

Abstract: In recent years, dataflow languages such as Pig Latin have emerged as flexible and powerful tools for handling complex analysis tasks on big data. These languages support schema flexibility as well as common programming patterns such as iteration. They offer extensibility through user-defined functions while running on top of scalable distributed platforms. In doing so, these languages enable analytical tasks while avoiding the limitations of classical query languages such as SQL and SPARQL. However, the tuple-oriented view of general-purpose languages like Pig does not match very well the specifics of modern datasets available on the Web, which often use the RDF data model. Graph patterns, for instance, are one of the core concepts of SPARQL but have to be formulated as explicit joins, which burdens the user with the details of efficient query processing strategies. In this paper, we address this problem by proposing extensions to Pig that deal with linked data in RDF to bridge the gap between Pig and SPARQL for analytics. These extensions are realized by a set of user-defined functions and rewriting rules, still allowing to compile the enhanced Pig scripts to plain MapReduce programs. For all proposed extensions, we discuss possible rewriting strategies and present results from an experimental evaluation.

1 Introduction

Processing and analyzing RDF [W3C04] data and particularly Linked (Open) Data [BL06] often requires operations going beyond the capabilities of classic query languages such as SPARQL [W3C08]. Although the most recent W3C recommendations for SPARQL support federated queries and aggregate functions, typical data preparation and cleaning steps as well as more advanced analytical tasks cannot easily be implemented directly in SPARQL or any similar query language.

Consider for example a user who wants to obtain information about events in all available categories (festival, jazz, orchestra, rock, etc.) that take place in the five biggest cities in the world. Even if the user has full access to the event dataset, determining which 5 cities are the biggest in the world requires access to another remote dataset. Answering such a query further requires expensive grouping and can therefore easily become expensive to evaluate in a triple store. Furthermore, integrating non-RDF sources, e.g., CSV files or complex tasks such as clustering the events on their geographic location or data cleaning tasks are not possible in SPARQL engines.

On the other hand, with the wide acceptance of the MapReduce platform [DG08] declarative dataflow languages such as Pig [ORS$^+$08] or Jaql [BEG$^+$11] have gained much attention. They provide a rich set of data processing operations, transparent parallelization to exploit data parallelism, can deal with flexible schemas, and are easy to learn even for non-specialists. Furthermore, these languages can easily be extended by user-defined functions enabling custom processing. Finally, by targeting MapReduce as execution environment, dataflow scripts are deployable to large-scale clusters for analyzing big datasets.

Although Pig supports a nested data model which is based on bags of tuples that in turn can contain bags, a tuple-oriented view results in several limitations and shortcomings:

- While SPARQL provides basic graph patterns (BGP) as a core concept of query formulation, such query patterns have to be expressed as self joins in a tuple-oriented language. Unfortunately, this is a quite expensive operation; first, because it is implemented in Pig by a so-called `COGROUP` operator representing a generalized form of join and grouping that is mapped to a combination of map and reduce tasks. The second reason is that self joins in Pig require to load the data twice, i.e., once for each branch of a join. Furthermore, the user is responsible for formulating efficient scripts, e.g., in terms of join order or join implementation.
- Pig is targeting the MapReduce platform and therefore works best if the data can be loaded from HDFS [Fou09]. However, processing Linked Data often requires to access and retrieve data from remote (SPARQL) endpoints – which is usually addressed by federated SPARQL engines. Depending on the size of the remote dataset, the capabilities of the endpoint, and the frequency of script execution, it might be useful to download and materialize the dataset before running the Pig script or to fetch (and cache) the remote data on demand. In both cases, this has to be explicitly implemented in the scripts.
- Pig provides a flexible nested data model that offers several alternatives for representing groups of triples or RDF statements. This includes not only the representation while processing the data but also the storage format in HDFS files. However, the choice of the most efficient format also depends on the workload and is hence again the responsibility of the user.

To overcome these limitations, this paper presents Pig language extensions adding SPARQL-like features that are particularly useful for implementing analytical processing tasks over Linked Data. The main goal of our work is to provide operators on a level of abstraction that lifts the burden of choosing the most efficient strategy and operator implementations from the user. These extensions include efficient support of BGPs and `FILTER`s as well as transparent access to remote Linked Open Data. Our contribution is twofold:

- We present a set of language extensions for making Pig a language suitable for analytical Linked Data processing. We discuss possible strategies for implementing these extensions in Pig.
- We report results from an experimental evaluation of the different strategies. In lack of a cost model and cost-based Pig compilers, these results can be used as a foundation to implement rewriting heuristics in Pig.

The work described in this paper is embedded in a larger vision of a platform for (Linked) Open Data analytics (LODHub [HS14]) that allows to publish, share, and analyze Linked Datasets. In this context, Pig is used – under the hood of a visual dataflow designer – as the data processing language.

The remainder of the paper is structured as follows. After a discussion of related work in Sect. 2, we briefly describe the context of our work in Sect. 3 and derive requirements on Pig suitable extensions for Linked Data processing. The proposed extensions are then presented in Sect. 4, followed by a discussion of the necessary dataflow planning and compiling steps in Sect. 5. Results of the experimental evaluation of the implementation and rewriting strategies are reported in Sect. 6. Finally, Sect. 7 concludes this paper and outlines future work.

2 Related Work

With the continuous growth of RDF datasets, we experience scalability issues when using a single server to process SPARQL queries. Despite efficient centralized SPARQL engines for a single server installation, such as RDF-3X [NW08], literature has proposed several systems [KM14] that use parallel processing in a cluster of machines. The architectures used for this purpose differ between orchestrating multiple centralized RDF stores [GHS14], using key-value stores [ZYW+13], or building upon distributed file systems [SPZL11, ZCTW13].

Especially systems using a distributed file system make use of MapReduce [DG08] to process joins [HMM+11, PKT+13]. One of the key considerations is to minimize the number of MapReduce cycles while still computing the complete answer to the query. Instead of directly mapping SPARQL queries to MapReduce programs, Pig Latin [ORS+08] can be used as an intermediary step, i.e., SPARQL queries are mapped to Pig Latin, which in turn is compiled into MapReduce. One of the first approaches of this class was presented in [SPZL11] where the translation of SPARQL queries into Pig Latin scripts is discussed. In this work, RDF data is represented as plain tuples of three fields and the operators of the SPARQL algebra are mapped to sequences of Pig Latin commands. In contrast to this approach, our work aims at integrating SPARQL elements such as BGPs into Pig, keeping the dataflow language as the primary language for specifying data analytics pipelines. Furthermore, we support multiple RDF datasets and SPARQL endpoints within a single script. In addition, we discuss and exploit different (intermediate) data representation formats for RDF data together with possible optimization strategies depending on these data representations.

An alternative line of optimization techniques make use of algebraic optimization [RKA11] to minimize the number of MapReduce cycles on a system based on Pig in combination with an algebra [KRA11] that considers triple groups, e.g., all triples with the same subject are considered a group. In our work we follow a similar idea of representing RDF triples but generalize the approach and consider it as only one alternative strategy for deriving efficient MapReduce programs.

In comparison to processing of standard SPARQL queries, analytical queries pose additional challenges due to their special characteristics such as complexity, evaluation on typically very large datasets, and long runtime [ADE+13, CGMR14]. Data cubes with dimensions and measures over RDF data can be defined using RDF compliant standards [EV12, W3C13]. On this basis, analytical queries can be formulated, evaluated, and optimized. Kotoulas et al. [KUBM12] propose an approach for query optimization that interleaves query execution and optimization and SPARQL queries are translated to Pig Latin.

In summary, existing systems have focused on RDF data organized in a single dataset that the user has complete control over. In contrast, this paper proposes a system that targets a broader perspective by considering analytical queries and Linked Data organized in multiple datasets and even remote sources that the user who issues the query does not have any control over.

3 Use Case: The LODHub Platform

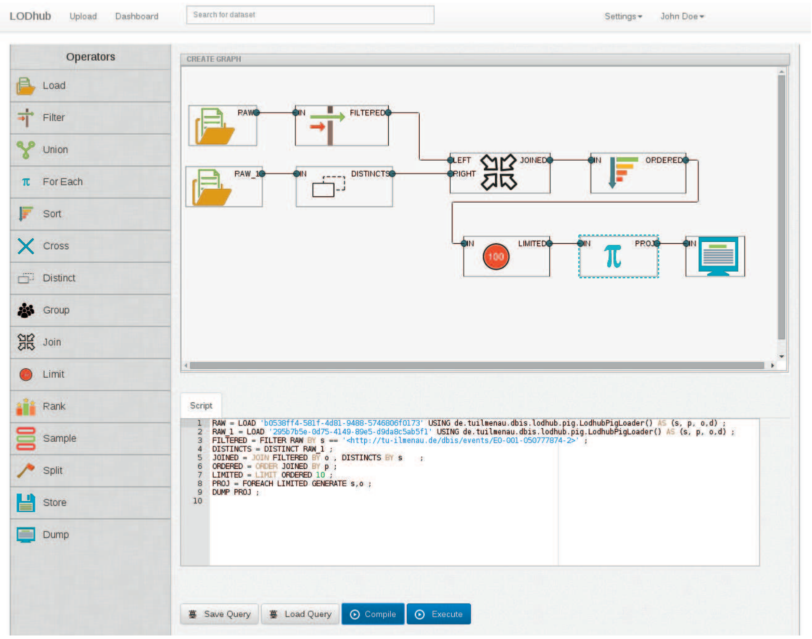
The goal of LODHub [HS14] is to combine the functionalities of frameworks for Open Data management with infrastructure for storing and processing large sets of Linked (Open) Data. An elastic server infrastructure, e.g., hosted by IaaS (Infrastructure as a Service) providers such as Amazon EC2, will allow to start machines automatically as workload peaks occur. Queries and datasets can then be distributed among these machines transparently to the user.

LODHub is designed to encourage users to share their data with other users and thus contribute to the idea of Open Data. Still, users can also choose to keep their datasets private. Additionally, as users often tend to work in teams, the platform will allow to associate user accounts with organizations and to organize users in teams. When sharing datasets with other users or creating a new version of an existing dataset, provenance information will be kept to track the original author.

Data exploration features will help users to discover new knowledge by automatically extending their query results with information found not only in other available datasets within LODHub, but also from remote sources on the Web.

The integration of Pig [ORS+08] enables users to perform analytical tasks that are not possible with a pure SPARQL approach. Such tasks include data transformation, combining different sources with different formats, and other complex tasks for which no equivalent SPARQL operations exist.

To let users work with their data, LODHub provides an easy-to-use and intuitive graphical editor to create Pig scripts. With this editor, a user can can drag and drop operators and edit their parameters. Operators have ports for input and output, and by connecting the output port of one operator to the input port of another operator, a dataflow is created. Connecting all needed operators results in a dataflow graph, which must contain one or more `LOAD` operators and at least one `DUMP` or `STORE` operator to show the results on screen or to save them as a new dataset, respectively. Between those two types of operators, users can add as many operators as needed. The screenshot in Fig. 1 shows the graphical Pig editor as well as the Pig script compiled from the operator graph.



```
RAW = LOAD 'b0538ff4-581f-4d81-9488-5746806f0173' USING de.tuilmenau.dbis.lodhub.pig.LodhubPigLoader() AS (s, p, o,d) ;
RAW_1 = LOAD '295b7b5e-0d75-4149-89e5-d9da8c5ab5f1' USING de.tuilmenau.dbis.lodhub.pig.LodhubPigLoader() AS (s, p, o,d) ;
FILTERED = FILTER RAW BY s == '<http://tu-ilmenau.de/dbis/events/E0-001-050777874-2>' ;
DISTINCTS = DISTINCT RAW_1 ;
JOINED = JOIN FILTERED BY o , DISTINCTS BY s    ;
ORDERED = ORDER JOINED BY p ;
LIMITED = LIMIT ORDERED 10 ;
PROJ = FOREACH LIMITED GENERATE s,o ;
DUMP PROJ ;
```

Figure 1: LODHub's visual dataflow designer for Pig scripts

In its current implementation, LODHub features almost all operators that Pig provides. Some of these operators, however, were modified to meet the challenges that arise when working with RDF data. Note, that LODHub is able to work not only with RDF data, but with any format for which a load function, e.g. as a UDF, exists.

The following scenario illustrates an example scenario and an analytical task that is supported by LODHub. Assume there exists an RDF dataset with information about events. These events may be concerts, sport events, etc., or even some sensor readings that have been logged into a file. Among other information each event is described by a title, a description of the event or sensor reading, a category (i.e., ”sport”, ”music”, or ”temperature”), and a position of the event in the form of lat/long coordinates.

Consider the following excerpt from a dataset (in total ca. 6.5 million events and more than 53 million statements) that were extracted from the website `http://eventful.com` and transformed into RDF (for simplicity, we abbreviate namespaces and event IDs):

```
<ev:event1> <ev:#title> "Metallica" .
<ev:event1> <ev:#start_time> "2012-08-17T18:00:00+02" .
<ev:event1> <ev:#venue_name> "Rexall Place" .
<ev:event1> <geo:#long> "-1.135E2" .
<ev:event1> <geo:#lat> "5.353E1" .
<ev:event1> <ev:#category> "music" .
```

Assume a user wants to find the number of events in each category for each of the five biggest cities in the world. To answer this question, we first need to find five biggest cities.

This information could for instance be extracted from the *remote* DBpedia [ABK$^+$07] dataset, which contains information about cities including name, location, and population.

The next step is to find out, which event took place in which city. As the event dataset contains the coordinates in two separate RDF statements (longitude and latitude), a *self-join* is required to obtain the full coordinate. Now, using these coordinates, we can use an *external webservice* such as GeoNames [Wic] to obtain the name of the city that an event is located in. Once we have identified the corresponding city for each event, we can join the results obtained from the DBpedia dataset with our events dataset on the cities. This produces the events located in the 5 biggest cities.

This intermediate result now has to be joined again with our original event dataset to obtain the event category. We can group this join result on city and event category and then count the frequencies to produce the answer to the user's query.

Based on this example scenario, we identify several challenges to Pig operators:

Loading datasets. First, the dataset loader must be able to efficiently load datasets from the file system (HDFS [Fou09] in our framework). It should hide URIs and file names from the user as users should not be bothered with details such as the internal file locations. Furthermore, the dataset loader must handle the special data format that is optimized for Pig on RDF data (see Sect. 4 for details).

Basic Graph Pattern. SPARQL uses BGPs that are matched against the underlying dataset. Pig, however, does not support these BGPs natively. To allow users to easily work with RDF data, the Pig operators must be able to efficiently interpret BGPs, which may range from single statements to complex graph patterns involving multiple joins.

Self joins. RDF data is based on the concept of triples defining a relationship (property) between a subject and an object. Therefore, processing SPARQL queries requires to evaluate many self joins to process the BGPs defined in SPARQL queries. In contrast to triples stores, Pig (and Hadoop) was not designed for RDF data and assumes only very few joins within a query. This makes plain Pig and Hadoop very inefficient on RDF data, because performing self joins requires to load the same dataset twice. As an example, consider the previous scenario: to construct the coordinate pair for the events, the dataset has to be loaded twice in order to perform the self join. And later to add the category information, we again have to perform a join with the original event dataset.

Remote joins. One of the benefits of RDF is its distributed nature. Information can be distributed over many servers and through the links within the datasets, these datasets are inter-connected. This means that processing a query might include performing a join with a dataset on a remote server. As datasets can be large, a lot of information might have to be downloaded. To assure reasonable response times, query strategies for such remote joins have to implemented. In our scenario, we need to compute a join with the DBpedia dataset. Such a join can be performed in different ways depending on the remote dataset's size and the data needed to answer the query: the datasets can either be downloaded completely (if possible) or if a SPARQL endpoint is available, a SPARQL query can be sent to retrieve only relevant parts of the dataset, which probably results in only a small amount of data to download.

4 Pig Extensions for Linked Data Processing

In order to address the requirements discussed in Sect. 3, we have developed several extensions to the Pig Latin language which we introduce in the following. These extensions particularly deal with features for processing RDF data as well as defining analytical pipelines and can be classified into the following groups:

- conversion between RDF triples and the Pig data model for efficient processing,
- accessing linked (RDF) datasets,
- supporting basic graph patterns for users more familiar with the SPARQL style of query formulation.

`TUPLIFY` – Tuple construction. Although the basic triple structure of RDF data in the form of subject s, predicate p, and object o is very flexible and allows to represent arbitrary structured data including graphs, it is not the optimal choice for efficient analytics. The main reason is that reconstructing complex objects from multiple triples requires (self) joins, which are very expensive particularly in MapReduce-based execution models such as the model underlying Pig. On the other hand, a rigid relational structure with a predefined schema, where a set of triples with the same subject is represented by $(s, o_1, o_2, \ldots, o_n)$ for the schema $(s, p_1, p_2, \ldots p_n)$, is often not flexible enough for RDF data. Fortunately, Pig provides a flexible nested data model with tuples, bags, and maps. Thus, tuple construction is an important operation necessary to process RDF data in Pig efficiently.

A first possible representation is to use maps for the predicate-object pairs, i.e., $(s, \{p_1 \rightarrow o_1, p2 \rightarrow o_2, \ldots\})$. However, this prohibits the use of the same predicate multiple times for a given tuple. Therefore, an alternative approach uses bags instead of maps for the predicate-object pairs, i.e., a schema {`subject:bytearray, stmts:`{`(predicate:bytearray, object:bytearray)`}}. This can further be generalized by allowing to group triples on any of the three components, e.g., on the predicate so that we obtain {`predicate:bytearray, stmts:`{`(subject:bytearray, object:bytearray)`}}. Which of these representations is the best choice depends on the operations that shall be applied on the data.

The details of tuple reconstruction are hidden by a `TUPLIFY` operator which is mapped to different implementations. If the input is a plain bag of triples, it simply groups them as follows:

```
triple_groups = FOREACH (GROUP triples BY subject)
   GENERATE group as subject,
            triples.(predicate, object) AS stmts;
```

`RDFLoad` – Accessing datasets. For accessing linked datasets we have to distinguish between local and remote access. The most straightforward way of local access is to load textual files in the Notation 3 (N3) format from HDFS, which is natively supported by Pig's `LOAD` operator. A more efficient storage scheme both in terms of space consumption and read costs is to leverage triple groups by using adjacency lists, i.e., a set of triples

$\{(s_1, p_1, o_1), (s_1, p_2, o_2), \ldots, (s_1, p_n, o_n)\}$ is represented by $(s_1,\ \{(p_1, o_1), \ldots, (p_n, o_n)\}$. Because the Pig data model supports bags, maps, and nested tuples, textual files representing this structure can be loaded directly into Pig scripts. In addition, by using a user-defined function (UDF) for LOAD this principle can also be extended to block-oriented binary files using dictionary and/or run-length encoding of the s, p, and o values.

Currently, we support the following strategies for accessing RDF data from HDFS:

- using Pig's native LOAD operator for N3 files. Because RDF literals can contain whitespaces, a UDF (RDFize) for tokenizing text lines containing triples into RDF components is used as part of a macro:

```
REGISTER sparklingpig.jar;
DEFINE RDFFileLoad(file) RETURNS T {
    lines = LOAD '$file' AS (txt: chararray);
    $T = FOREACH lines
         GENERATE FLATTEN(pig.RDFize(txt))
         AS (subject, predicate, object);
}
triples = RDFFileLoad('rdf-data.nt'):
```

- using Pig's LOAD operator with the BinStorage function for triple groups:

```
rdf_tuples = LOAD 'rdf-data.dat' USING BinStorage() AS
   (subject: bytearray,
    stmts: (predicate: bytearray, object:bytearray));
```

In addition, RDF datasets may also be retrieved from remote SPARQL endpoints. For this purpose, we provide a UDF, called SPARQLLoader, that sends a given SPARQL query to the endpoint and transforms the result data (usually encoded in XML or JSON) into Pig tuples:

```
REGISTER sparklingpig.jar;
raw = LOAD 'http://endpoint.org:8080/sparql'
      USING SPARQLLoader('SELECT * WHERE { ?s ?p ?o }')
      AS (subject, predicate, object);
```

Note that the latter opens up new options to deal with the retrieved data: depending on the size of the dataset, it could be useful to download and materialize the data in HDFS for reuse before starting the actual processing. Te keep the load at public SPARQL endpoints low, it is of course advisable to download database dumps when available and manually store them in HDFS.

Furthermore, depending on the user query, we can choose not to download all triples of the remote endpoint but only those triples that are necessary to answer the user's query. For instance, given a FILTER expression on the population in a user query, then the following script retrieves only triples encoding the population of cities:

```
REGISTER sparklingpig.jar;
raw = LOAD 'http://endpoint.org:8080/sparql'
      USING SPARQLLoader(
           'CONSTRUCT {?s dbpedia:populationTotal ?pop }
           WHERE {?s rdf:type dbpedia:City .
                  ?s dbpedia:populationTotal ?pop}')
      AS (subject, predicate, object);
```

If metadata is available that describes the storage format of the data and indicates whether data from remote endpoints have already been downloaded, then the details of these different strategies can be encapsulated by a generic RDFLoad operator.

FILTER – Basic Graph Pattern. BGPs are sets of triple patterns that are used not only for simple filtering but also for joins. Translating BGPs in the style of relational query processing results in a number of expensive joins. Furthermore, the exact formulation depends on the triple/tuple layout of the data; for example, if triples are used then we need a JOIN, whereas for bag-based tuple structures, nested FOREACH/FILTER operations can be used. Consider the following simple BGP as an example:

```
{ ?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat>  ?o1 .
  ?s <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?o2 }
```

Based on a triple schema in Pig, i.e., {subject:bytearray, predicate:bytearray, object:bytearray }, the BGP could be implemented by:

```
triples1 = RDFLoader('rdf-data.nt');
triples2 = RDFLoader('rdf-data.nt');
result = JOIN triples1 BY (subject), triples2 BY (subject);
```

Note that for self joins Pig requires to load the data twice as in the example above. Of course, this would not be necessary if the second pattern stemmed from another dataset.

In contrast, if we have constructed RDF tuples using the schema {subject:bytearray, stmts:{(predicate: bytearray,object: bytearray)}} before, i.e., creating a bag of pairs (predicate, object) belonging to a subject, then we can implement the BGP by:

```
tmp = FOREACH rdf_tuples {
   r1 = FILTER stmts
        BY (predicate == '<http://www.w3.org/2003/01/geo/wgs84_pos#lat>');
   r2 = FILTER stmts
        BY (predicate == '<http://www.w3.org/2003/01/geo/wgs84_pos#long>');
   GENERATE *, COUNT(r1) AS cnt1, COUNT(r2) AS cnt2;
};
result = FILTER tmp BY cnt1 > 0 AND cnt2 > 0;
```

For convenience, we hide the details of implementing BGP processing by an extended FILTER operator which accepts a BGP as part of the BY clause, but have to translate this expression accordingly:

```
result = FILTER triples BY
         { ?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat>  ?o1 .
           ?s <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?o2 };
```

5 Pig Planning and Compiling

In the previous section we described several extensions to Pig Latin to process Linked Data. To execute scripts involving these extensions, we need to transform them into plain Pig scripts. This transformation is implemented as a three aspects: planning, rewriting, and UDFs. The remainder of this section discusses these aspects in more detail.

Planning. Given a user query, we can often use metadata to rewrite the query into a more efficient version. Since our extended Pig, for instance, allows access to local as well as remote datasets, a query might contain `LOAD` operations on a remote SPARQL endpoint. Given the information that the dataset was already downloaded and materialized while executing another query, we can rewrite the original query into one that uses a local dataset instead.

To identify these cases, we need to collect metadata and statistics, e.g., a mapping `(URL, query)` → `HDFS path` that we can use to decide whether a remote dataset is already available locally. As an example, consider the `LOAD` operation from Sect. 4:

```
raw = LOAD 'http://endpoint.org:8080/sparql'
      USING SPARQLLoader('SELECT * WHERE { ?s ?p ?o }')
      AS (subject, predicate, object);
```

If a mapping `(http://endpoint.org:8080/sparql, SELECT * WHERE {?s ?p ?o})` → `hdfs:///rdf-data.nt` exists, then we can rewrite the expression as

```
raw = RDFFileLoad('hdfs:///rdf-data.nt');
```

which can obviously be executed more efficiently by avoiding expensive data exchange over the Web.

Note that this cannot be achieved in a pure Pig environment because of the missing option to generate and store meta information. A framework like LODHub, however, could easily store such information and reuse data across user queries.

In addition to the above mentioned mappings, LODHub could also store information about the file format, i.e., if the content of a file is structured as plain triples or as triple groups, as well as additional statistics (e.g., VoID [W3C10] statistics) about the downloaded datasets, which will allow for further optimization.

Rewriting. Though UDFs provide a flexible mechanism to extend Pig with additional functionality, we decided not to implement all extensions as UDFs because transforming the extended Pig expressions to plain Pig allows us to use all optimizations that Pig already comes with. Instead, we decided to implement BGP support via rewriting rules, i.e., before execution the Pig script is passed to a rewriter that analyzes the code and uses rewriting rules to transform extended Pig statements into plain Pig statements. These rules cover various input formats, BGPs, and operations. An example for such a rewriting was already given in Sect. 4 for the extended filter operation. The input format defines the structure of the data stream, i.e., if they are plain tuples from an N3 file or an external SPARQL endpoint, or if the stream is already in a triple group representation as introduced in Sect. 4. In the following, we present and discuss these rewriting rules in detail.

We use the notation $alias_{format}^{(schema)}$ to describe the characteristics of the input and output data of an operator. The *alias* is the Pig alias of the input or output data, *format* describes the format of the data, i.e., if it is plain triples or triple groups. In the former case we write *plain* and in the latter case we write *group*. The *schema* denotes the schema of the data, e.g., `(subject, predicate, object)` for plain triples or `(subject, {(predicate, object)})` for triple groups. For ease of presentation, the following rules use the subject (s) as a default to denote grouping – nevertheless, the rules hold for other groupings as well.

(T1) If the data is available as plain triples, it can be tuplified using the TUPLIFY operator. It is the task of the optimizer to decide whether or not to apply this rule in order to create triple groups for a given input.

```
$alias^{(s,p,o)}_{plain}$
$\Rightarrow$
$out^{(s,\{(p,o)\})}_{group}$ = TUPLIFY(alias) = FOREACH (GROUP alias BY s)
                                  GENERATE group as s, alias.(p, o) AS stmts;
```

(R1) If the RDFLoad operator is used with a reference to a remote endpoint ($uri.protocol = 'http'$), replace the operator with the UDF SPARQLLoader as introduced in Sect. 4.

```
out = RDFLoad('uri') AS (schema);
$\Rightarrow$
$out^{(schema)}_{plain}$ = LOAD 'uri'
             USING pig.SPARQLLoader(
                'SELECT * WHERE { ?s ?p ?o }')
             AS (schema);
```

(R2) As already outlined in Sect. 4, we can use a restrictive query to reduce the amount of triples fetched from the SPARQL endpoints. The necessary knowledge can be drawn from FILTER operations that use BGPs.

```
out = RDFLoad('uri') AS (schema);
out2 = FILTER out BY {$BGP_1$ . $BGP_2$};
$\Rightarrow$
$out^{(schema)}_{plain}$  = LOAD 'uri'
               USING pig.SPARQLLoader(
                 'CONSTRUCT * WHERE { $BGP_1$ }')
               AS (schema);
$out2^{(schema)}_{plain}$ = FILTER $out^{(schema)}_{plain}$ BY { $BGP_2$ };
```

BGP_1 and BGP_2 represent basic graph patterns, e.g., star joins. The (sub)set of the BGP that restricts the remote data can be used to define the query that is sent to the SPARQL endpoint. Often, the FILTER's BGP cannot completely be used in the query. In this case, the remaining part of the BGP still has to be applied in a FILTER. Note that the LOAD does not have to be directly followed by the FILTER to match this rule. Rather, the output of the LOAD has to be passed as input to the FILTER transitively.

(L1) To load a local dataset, the URI uses the *hdfs* protocol (*uri.protocol = 'hdfs'*). In this case, the RDFLoad operation can be replaced by the RDFFileLoad macro. If the dataset has been saved as plain triples, the output of the loader are plain triples, too.

```
out = RDFLoad('uri');
$\Rightarrow$
$out^{(s,p,o)}_{plain}$ = RDFFileLoad('uri');
```

(L2) Loading local datasets that have been saved in the triple group format produces an output with the triple group schema. Consider the output was grouped by subject (s), the RDFLoad operation will be rewritten as:

```
out = RDFLoad('uri');
⇒
out_{group}^{(s,{(p,o)})} = LOAD 'uri' USING BinStorage()
                  AS (s:chararray, stmts:bag{
                    t:(p:chararray,o:chararray)
                  });
```

The resulting schema depends on which component (subject, predicate, or object) was used for grouping. This information can be stored in the meta data for each dataset.

(F1) A singe triple pattern may use only unbound variables. If such a triple pattern is used in a `FILTER`, it can be omitted since it has no effect:

```
out^{(schema_1)} = FILTER in^{(schema_1)} BY { ?s ?p ?o };
out2^{(schema_2)} = any_operator out^{(schema_1)};
⇒
out2^{(schema_2)} = any_operator in^{(schema_1)};
```

Note, that in this case, the format of the aliases does not matter.

(F2) Filter operators that use a single triple pattern with only one bound variable can be rewritten as a simple `FILTER` with one predicate.

```
out_{plain}^{(s,p,o)} = FILTER in_{plain}^{(s,p,o)} BY { ?s ?p 'value' };
⇒
out_{plain}^{(s,p,o)} = FILTER in_{plain}^{(s,p,o)} BY o == 'value';
```

Rewriting rules for one bound variable in the other two components (*subject* or *predicate*) are obviously analogously defined. Hence, we omit them here.

(F3) If the triple pattern of a filter contains two (ore more) bound variables, it is rewritten as a filter operation that connects both predicates by `AND`.

```
out_{plain}^{(s,p,o)} = FILTER in_{plain}^{(s,p,o)} BY { ?s 'value1' 'value2' };
⇒
out_{plain}^{(s,p,o)} = FILTER in_{plain}^{(s,p,o)}
              BY p == 'value1' AND o == 'value2';
```

As in rule (F2) above, there are analogous rules for the bound variables in other (or all three) components.

(F4) If the input alias of the FILTER operator is in the triple group format and if the one bound variable is the grouping column, e.g., `s`, the following rule applies:

```
out_{group}^{(s,{(p,o)})} = FILTER in_{group}^{(s,{(p,o)})} BY { 'value' ?p ?o };
⇒
out_{group}^{(s,{(p,o)})} = FILTER in_{group}^{(s,{(p,o)})} BY s == 'value';
```

(F5) If the bound variable of the filter triple pattern is not the grouping column, e.g., `o`, then the filter operation becomes more complex:

```
out$_{group}^{(s,\{(p,o)\})}$ = FILTER in$_{group}^{(s,\{(p,o)\})}$ BY { ?s ?p 'value' };
$\Rightarrow$
tmp$_{group}^{(s,\{(p,o)\},cnt)}$ = FOREACH in$_{group}^{(s,\{(p,o)\})}$ {
  t = FILTER stms BY o == 'value';
  GENERATE *, COUNT(t) AS cnt;
};
out$_{group}^{(s,\{(p,o)\},cnt)}$ = FILTER tmp$_{group}^{(s,\{(p,o)\},cnt)}$ BY cnt > 0;
```

(F6) If the triple pattern contains two bound variables and neither of them is the grouping column, then both predicates can be connected by AND.

```
out$_{group}^{(s,\{(p,o)\})}$ = FILTER in$_{group}^{(s,\{(p,o)\})}$ BY { ?s 'v1' 'v2' };
$\Rightarrow$
tmp$_{group}^{(s,\{(p,o)\},cnt)}$ = FOREACH in$_{group}^{(s,\{(p,o)\})}$ {
  t = FILTER stms BY p == 'v1' AND o == 'v2';
  GENERATE *, COUNT(t) AS cnt;
};
out$_{group}^{(s,\{(p,o)\},cnt)}$ = FILTER tmp$_{group}^{(s,\{(p,o)\},cnt)}$ BY cnt > 0;
```

(F7) If the triple pattern contains two bound variables and one is the grouping column, it will be split into two filter operations. These can then further be processed by rules (F4) and (F5).

```
out$_{group}^{(s,\{(p,o)\})}$ = FILTER in$_{group}^{(s,\{(p,o)\})}$ BY { 'v1' ?p 'v2' };
$\Rightarrow$
tmp$_{group}^{(s,\{(p,o)\})}$     = FILTER in$_{group}^{(s,\{(p,o)\})}$  BY { 'v1' ?p ?o };
out$_{group}^{(s,\{(p,o)\},cnt)}$ = FILTER tmp$_{group}^{(s,\{(p,o)\})}$ BY { ?s ?p 'v2' };
```

(F8) Analogously to rule (F7), if the input is in the triple group format and a filter triple pattern contains only bound variables, it will be replaced by two filter operations, which in turn can further be processed by rules (F4) and (F6).

```
out$_{group}^{(s,\{(p,o)\})}$ = FILTER in$_{group}^{(s,\{(p,o)\})}$ BY { 'v1' 'v2' 'v3' };
$\Rightarrow$
tmp$_{group}^{(s,\{(p,o)\})}$    = FILTER in$_{group}^{(s,\{(p,o)\})}$  BY { 'v1' ?p ?o };
out$_{group}^{(s,\{(p,o)\},cnt)}$ = FILTER tmp$_{group}^{(s,\{(p,o)\})}$ BY { ?s 'v2' 'v3' };
```

(F9) If a filter contains more than one triple pattern ($TP_1, ..., TP_N$), i.e. a BGP, and they do not form a join, they can be evaluated one after the other as an implicit AND. We leave it to the Pig engine to optimize the sequence of filters. In this case the format of the respective aliases again do not matter.

```
out$^{(schema)}$ = FILTER in$^{(schema)}$ BY { TP$_1$ ... TP$_N$ };
$\Rightarrow$
f1$^{(schema)}$ = FILTER in$^{(schema)}$  BY { TP$_1$ };
...
fN$^{(schema)}$ = FILTER f(N-1)$^{(schema)}$  BY { TP$_N$ };
out$^{(schema)}$ = fN;
```

(J1) If the data is available in plain triple format and if the BGP of a filter expression forms a star join (i.e., the triple pattern $TP_1, ..., TP_N$ share the same variable in the same position, e.g., on subject), the dataset has to be loaded for each BGP participating in the join. Such joins may contain bound variables. Hence, we first filter the input, for these bound variables.

```
$in_{plain}^{(s,p,o)}$  = RDFLoad('uri');
$out_{plain}^{(s,p,o)}$ = FILTER $in_{plain}^{(s,p,o)}$ BY { TP$_1$ ... TP$_N$ };
⇒
$in1_{plain}^{(s,p,o)}$ = RDFLoad('uri');
$f1_{plain}^{(s,p,o)}$  = FILTER $in1_{plain}^{(s,p,o)}$ BY { TP$_1$ };
...
$inN_{plain}^{(s,p,o)}$  = RDFLoad('uri');
$fN_{plain}^{(s,p,o)}$   = FILTER $inN_{plain}^{(s,p,o)}$ BY { TP$_N$ };
$out_{plain}^{(s,p,o)}$ = JOIN f1$_{plain}^{(s,p,o)}$ BY (s), ..., fN$_{plain}^{(s,p,o)}$ BY (s);
```

Each filter operation can then be further processed by rules (F1) - (F3).

(J2) As already stated in previous sections, the star join can easily be realized using `FILTER` operators if the input is already in the triple group format. If the triple pattern contains bound variables, e.g. in the predicate position with values $p_1, ..., p_N$, a filter has to be applied.

```
out$_{group}^{(s,\{(p,o)\})}$  = FILTER $in_{group}^{(s,\{(p,o)\})}$
              BY { TP$_1$ ... TP$_N$ };
⇒
tmp$_{group}^{(s,\{(p,o)\},cnt1,...,cntN)}$  = FOREACH $in_{group}^{(s,\{(p,o)\})}$ {
  t1 = FILTER stms BY p == '$p_1$';
  ...
  tN = FILTER stms BY p == '$p_N$';
  GENERATE *, COUNT(t1) AS cnt1, ..., COUNT(tN) as cntN;
};
out$_{group}^{(s,\{(p,o)\},cnt1,...,cntN)}$  = FILTER tmp$_{group}^{(s,\{(p,o)\},cnt1,...,cntN)}$
                         BY cnt1 > 0 AND ... AND cntN > 0;
```

(J3) A path join is characterized by the fact that the join variables are not on the same positions in all triple pattern, e.g., the object of TP$_i$ is joined with the subject of TP$_{i+1}$. If a triple pattern contains a constant that can be used as a filter, those filters can be pushed down.

```
$in_{plain}^{(s,p,o)}$  = RDFLoad('uri');
out$_{plain}^{(s,p,o)}$ = FILTER $in_{plain}^{(s,p,o)}$ BY { TP$_1$ ... TP$_N$ };
⇒
$in1_{plain}^{(s,p,o)}$      = RDFLoad('uri');
filter1$_{plain}^{(s,p,o)}$  = FILTER $in_{plain}^{(s,p,o)}$ BY { TP$_1$ };
...
$inN_{plain}^{(s,p,o)}$       = RDFLoad('uri');
filterN$_{plain}^{(s,p,o)}$  = FILTER $inN_{plain}^{(s,p,o)}$ BY { TP$_N$ };
out1$_{plain}^{(schema_{filter1},schema_{filter2})}$  = JOIN filter1$_{plain}^{(s,p,o)}$ BY (o),
                                   filter2$_{plain}^{(s,p,o)}$ BY (s);
...
out$i_{plain}^{(schema_{outi-1},schema_{filteri+1})}$  = JOIN out$_{i-1}{}_{plain}^{(schema_{outi-1})}$ BY (o),
                                    filter$i+1_{plain}^{(s,p,o)}$  BY (s);
...
out$_{plain}^{(schema_{outN-1},schema_{filterN})}$    = JOIN out$_{N-1}{}_{plain}^{(schema_{outN-1})}$ BY (o),
                                    filter$N_{plain}^{(s,p,o)}$    BY (s);
```

The filter operations on each triple pattern can be processed by rules (F1)-(F3)

(J4) In case the input is loaded as triple groups and a path join needs to be evaluated, we have to flatten the structure and pass the flat triples to the join as in rule (J3) above.

```
$in_{group}^{(s,\{(p,o)\})}$  = RDFLoad('uri');
$out_{group}^{(s,\{(p,o)\})}$ = FILTER $in_{group}^{(s,\{(p,o)\})}$
                BY { TP$_1$ ... TP$_N$ };
⇒
$in1_{group}^{(s,\{(p,o)\})}$     = RDFLoad('uri');
$filter1_{group}^{(s,\{(p,o)\})}$ = FILTER $in_{group}^{(s,\{(p,o)\})}$ BY { TP$_1$ };
$flat1_{plain}^{(s,p,o)}$       = FOREACH $filter1_{group}^{(s,\{(p,o)\})}$ GENERATE s, FLATTEN(stmts);
...
$inN_{group}^{(s,\{(p,o)\})}$     = RDFLoad('uri');
$filterN_{group}^{(s,\{(p,o)\})}$ = FILTER $inN_{group}^{(s,\{(p,o)\})}$ BY { TP$_N$ };
$flatN_{plain}^{(s,p,o)}$       = FOREACH $filterN_{group}^{(s,\{(p,o)\})}$ GENERATE s, FLATTEN(stmts);
$out1_{plain}^{(schema_{flat1},schema_{flat2})}$ = JOIN $flat1_{plain}^{(s,p,o)}$ BY (o), $flat2_{plain}^{(s,p,o)}$ BY (s);
...
$outi_{plain}^{(schema_{outi-1},schema_{flati+1})}$ = JOIN $out_{i-1}{}_{plain}^{(schema_{outi-1})}$ BY (o),
                                   $flati+1_{plain}^{(s,p,o)}$     BY (s);
...
$out_{plain}^{(schema_{outN-1},schema_{flatN})}$ = JOIN $out_{N-1}{}_{plain}^{(schema_{outN-1})}$ BY (o),
                                   $flatN_{plain}^{(s,p,o)}$       BY (s);
```

UDFs and Compiling. We implemented a few UDFs to load and process linked datasets. We use the `SPARQLLoader` that accepts a URL of a SPARQL endpoint and a query to retrieve data from that endpoint. Furthermore, there is a `RDFize` UDF that is used to read and parse statements in N3 files. For compiling and optimizing the resulting script we completely rely on the Pig compiler to generate the MapReduce programs.

6 Evaluation

In this section, we show that the performance of the scripts depends on which rules are applied to transform the extended Pig script into plain Pig. To do so, we compare the execution times for scripts that were generated by different rules. These experiments show that there is a need for an optimizer that has access to various statistics in order to decide which rule to choose.

In our experiments, we used a Hadoop cluster of eight nodes, consisting of one name node and seven data nodes. Each node has an Intel Core i5-3470S CPU with four cores running at 2.9 GHz and 16 GB RAM. The cluster is managed by Cloudera CDH 5.2 running Hadoop 2.5 and Pig 0.12. We imported an 8 GB dataset, which we introduced in Sect. 3, that contains ∼54 million statements.

In the experiments, we manually transformed the extended Pig statements into plain Pig for typical operations on RDF data such as filtering on BGP and self joins. There are two scenarios for input data: (i) the input data are plain N3 triples (based on which triple groups might be generated on-the-fly) and (ii) the input data is in triple group format. The third case, where data is fetched from a remote SPARQL endpoint, is not considered in this evaluation as the delays caused by remote communication will not lead to additional insights.

For these scenarios, we first compared the execution times of a self join. The task is to compute the latitude and longitude for each event. We started with a Pig script that uses extended Pig features for loading and evaluating a BGP:

```
raw = RDFLoad(’hdfs:///eventful.nt’)
res = FILTER raw BY  {
  ?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat>  ?lat .
  ?s <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long }
```

The script includes one `LOAD` and one `FILTER` operator. For `LOAD` we have the two scenarios mentioned above. Depending on the chosen rewriting for `LOAD`, we have two scenarios for the BGP processing: on plain triples or on triple groups. For the latter case, the triple groups are either read directly from the input file or generated on-the-fly. This results in (theoretically) four execution scenarios: (i) data is loaded as plain triples and the filter is evaluated on plain triples, (ii) data is loaded as plain triples and the filter is evaluated on triple groups (this requires generating the groups on-the-fly), (iii) data is loaded as triple groups and evaluation is on plain triples, and (iv) input is loaded as triple groups and filter is evaluated on these groups. We do not consider case (iii) in this evaluation because converting the compact form to the verbose N3 format will obviously not be beneficial. For scenario (iv) the triple groups were generated beforehand using the code shown in Sect. 4. The result was saved to HDFS using Pig’s `BinStorage`.

For scenario (i) the above extended Pig script can be rewritten using rule (L1) for the load operation and (J1) for the BGPs:

```
raw1 = RDFFileLoad(’hdfs:///eventful.nt’);
f1 = FILTER triples1 BY predicate ==
          ’<http://www.w3.org/2003/01/geo/wgs84_pos#long>’;
raw2 = RDFLoad(’hdfs:///eventful.nt’);
f2 = FILTER triples2 BY predicate ==
         ’<http://www.w3.org/2003/01/geo/wgs84_pos#lat>’;
res = JOIN f1 BY (subject), f2 BY (subject);
```

To rewrite the input scenario (ii), we used rules (L1) for the `LOAD` and (T1) to create the triple groups. To rewrite the BGP in the filter expression, rule (J2) was used. For scenario (iv) the rules (L2) and (J2) were applied.

The execution times of the different rewriting results are shown in Fig. 2. For each script, we used one to eight partitions per input dataset. One partition is then processed by one mapper. For scenario (i) this means that because we need to load the dataset twice to perform the join, we actually have twice as many mappers as partitions. For both scenarios (ii) and (iv) we only need to load the dataset once.

As we can see in the figure, working with triple groups is clearly faster than working with plain triples. Although these gaps decrease when increasing the number of partitions,

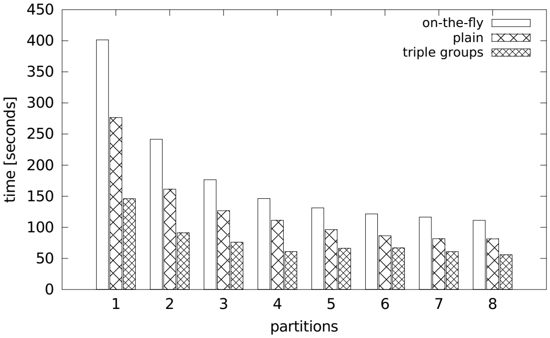


Figure 2: Execution times for self join operation with different numbers of partitions. Left: scenario (ii), middle: scenario (i), and right: scenario (iv).

note that for plain triples there are actually twice as many mappers as for the other two scenarios, i.e., this scenario consumes much more resources. However, if the triple groups have to be generated on-the-fly first, there is a significant performance overhead and the processing takes significantly longer as for the scenario where the groups were loaded from a file. As expected, the results suggest that it may be beneficial to materialize such intermediate results, since this would allow other operators in complex scripts to reuse these results. If the materialized results are stored permanently, the optimizer can make use of it in other scripts, too.

Further, we can see the impact of the number of mappers used for execution. When increasing the number of partitions, Pig can make use of more mappers to process the data in parallel, which reduces the overall processing time.

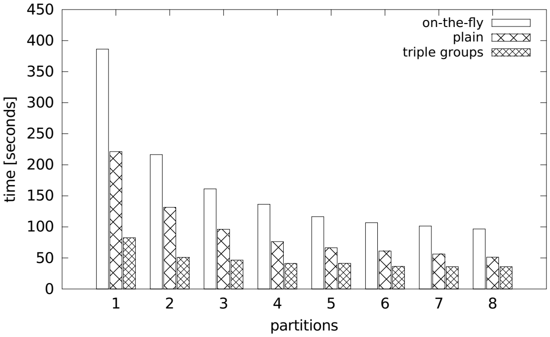


Figure 3: Execution times for a BGP filter on a non-grouping component (object) with different numbers of partitions. Left: scenario (ii), middle: scenario (i), and right: scenario (iv).

Next, we measured the performance for a filter on the object component of a triple. The extended Pig script is similar to the one above, despite the filter contains only one triple pattern:

```
raw = RDFLoad('hdfs:///eventful.nt');
res = FILTER raw BY { ?s ?p 'Metallica'. };
```

Similar to the self join script above, we evaluated three rewriting scenarios. For scenario (i), the load operation is also rewritten using rule (L1) and the filer using rule (F2). In scenario (ii), we applied the rules (L1), (T1), and (F5). And in scenario (iv), the rules (L2) and (F5) were used. As an example, we show the rewriting result for scenario (ii):

```
raw = RDFFileLoad('hdfs:///eventful.nt');
out = TUPLIFY(raw);
tmp = FOREACH out {
        t = FILTER stmts BY (object == 'Metallica');
        GENERATE *, COUNT(t) as cnt;
      };
res = FILTER tmp BY cnt > 0;
```

The execution times for the three input scenarios are shown in Fig. 3.

Here, the differences between the scenarios are even greater. Since filtering is not as expensive as a join operation, the overhead of creating the triple groups on-the-fly causes a greater gap in comparison to the other two scenarios. However, we can clearly see that the triple groups (when loaded directly from file) are still faster than filtering plain triples. Since the filter was on the object, the filter operation had to process each tuple in the bag for each subject, which leads to inspecting just as many triples as when evaluation plain triples. In this case, the advantage comes from the fewer data that has to be read from disk, since the subject is stored only once. Hence, when the dataset becomes larger, the differences will increase to the advantage of the triple groups. Also, if the filter has to be evaluated on the component on which the triple groups are grouped, the difference between filtering triple groups and plain triples becomes even greater because the filter predicate has to be evaluated only once per grouped entity (e.g., subject), see Fig. 4. In contrast, for plain triples the filter is evaluated for each statement that belongs to an entity, leading to significantly more filter operations.

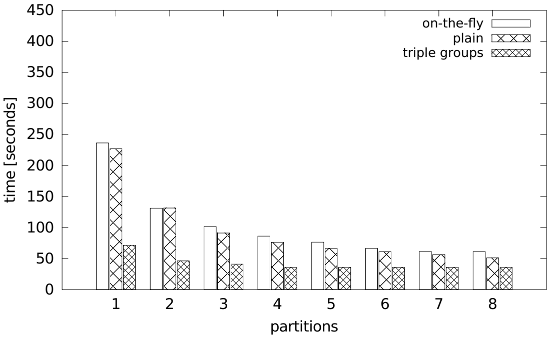


Figure 4: Execution times for a BGP filter on a grouping component (subject) with different numbers of partitions. Left: scenario (ii), middle: scenario (i), and right: scenario (iv).

One can see that the number of mappers also has an impact on the execution time. In our small setup, these differences do not seem to be big enough to be significant. However, if there is a difference in such a small setup, the difference will be even greater when the number of nodes in the cluster that can participate in the computation or the size of the dataset to process increases.

In this evaluation, we forgo experiments with a path join, because our rule set transforms triple groups to plain triples and then relies on Pig join. Hence, an evaluation of this operation will not reveal new insights.

7 Conclusion

In this paper, we extended the Pig language with support of SPARQL-like features, such as filtering and joins on BGPs, which are particularly useful to perform analytical tasks on linked datasets. The extensions were introduced at three levels of implementation: as strategies for query planning and data access, as rewriting rules to transform the Pig extensions to plain Pig, and as UDFs that implement new functionality. In doing so, we are able to use the Pig compiler to compile and optimize the scripts for execution. The current implementation of our system contains the UDFs and macros to load RDF data either from HDFS or from remote sources via SPARQL endpoints. The implementation is also capable of generating, loading, processing, and storing data in the triple group format.

The results of our experiments show that performance depends on which rules are used for in the rewriting step. To show this effect, the scripts were generated manually using a set of rewriting rules. To support arbitrary queries, we plan to implement a rewriter in our future work that combines the planning and rewriting steps into one optimization step. The optimizer will use heuristics to automatically decide which rules to use for rewriting extended Pig and to decide whether intermediate results should be materialized. To facilitate these decisions, statistics and metadata about the datasets will be obtained from the LODHub platform, which provides the context for our work.

Acknowledgements

This work was partially funded by the German Research Foundation (DFG) under grant no. SA782/22

References

[ABK$^+$07] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC*, 2007.

[ADE$^+$13] A. Abelló, J. Darmont, L. Etcheverry, M. Golfarelli, J. Mazón, F. Naumann, T. B. Pedersen, S. Rizzi, J. Trujillo, P. Vassiliadis, and G. Vossen. Fusion Cubes: Towards Self-Service Business Intelligence. *IJDWM*, 9(2):66–88, 2013.

[BEG$^+$11] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *PVLDB'11*, 2011.

[BL06] T. Berners-Lee. Linked Data. W3C Design Issues. `http://www.w3.org/DesignIssues/LinkedData.html`, 2006.

[CGMR14] D. Colazzo, F. Goasdoué, I. Manolescu, and A. Roatis. RDF analytics: lenses over semantic graphs. In *WWW'14*, pages 467–478, 2014.

[DG08] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[EV12] L. Etcheverry and A. A. Vaisman. QB4OLAP: A Vocabulary for OLAP Cubes on the Semantic Web. In *COLD'12*, 2012.

[Fou09] The Apache Software Foundation. Hadoop. `http://hadoop.apache.org/core/`, 2009.

[GHS14] L. Galárraga, K. Hose, and R. Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. In *WWW'14*, pages 267–268, 2014.

[HMM+11] M. Husain, J. McGlothlin, M. M Masud, L. Khan, and B. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1312–1327, 2011.

[HS14] S. Hagedorn and K.-U. Sattler. LODHub - A Platform for Sharing and Analyzing large-scale Linked Open Data. In *Proc. of ISWC 2014 demo track*, volume 1272, pages 181–184, October 2014.

[KM14] Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *The VLDB Journal*, pages 1–25, 2014.

[KRA11] H. Kim, P. Ravindra, and K. Anyanwu. From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra. *PVLDB*, 4(12):1426–1429, 2011.

[KUBM12] S. Kotoulas, J. Urbani, P. Boncz, and P. Mika. Robust Runtime Optimization and Skew-Resistant Execution of Analytical SPARQL Queries on Pig. In *ISWC'12*, pages 247–262. 2012.

[NW08] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[ORS+08] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD'08*, pages 1099–1110, 2008.

[PKT+13] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *IEEE International Conference on Big Data*, pages 255–263, 2013.

[RKA11] P. Ravindra, H. Kim, and K. Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *ESWC'11*, pages 46–61, 2011.

[SPZL11] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *SWIM'11*, pages 4:1–4:8, 2011.

[W3C04] W3C. RDF Primer (W3C Recommendation 2004-02-10). `http://www.w3.org/TR/rdf-primer/`, 2004.

[W3C08] W3C. SPARQL Query Language for RDF (W3C Recommendation 2008-01-15). `http://www.w3.org/TR/rdf-sparql-query/`, 2008.

[W3C10] W3C. Describing Linked Datasets with the VoID Vocabulary. `http://www.w3.org/TR/void/`, 2010.

[W3C13] W3C. The RDF Data Cube Vocabulary. `http://www.w3.org/TR/2013/CR-vocab-data-cube-20130625/`, 2013.

[Wic] Marc Wick. GeoNames. `http://www.geonames.org/`.

[ZCTW13] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *ICDE'13*, pages 565–576, 2013.

[ZYW+13] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. In *PVLDB'13*, pages 265–276, 2013.