Robust Runtime Optimization and Skew-Resistant Execution of Analytical SPARQL Queries on Pig

Spyros Kotoulas[1 2 *], Jacopo Urbani[2 *], Peter Boncz[3 2], and Peter Mika[4]

[1] IBM Research, Ireland `Spyros.Kotoulas@ie.ibm.com`
[2] Vrije Universiteit Amsterdam, The Netherlands `jacopo@cs.vu.nl`
[3] CWI Amsterdam, The Netherlands `boncz@cwi.nl`
[4] Yahoo! Research Barcelona, Spain `pmika@yahoo-inc.com`
[*] *these authors have contributed equally to this work*

Abstract. We describe a system that incrementally translates SPARQL queries to Pig Latin and executes them on a Hadoop cluster. This system is designed to work efficiently on complex queries with many self-joins over huge datasets, avoiding job failures even in the case of joins with unexpected high-value skew. To be robust against cost estimation errors, our system *interleaves* query optimization with query execution, determining the next steps to take based on data samples and statistics gathered during the previous step. Furthermore, we have developed a novel skew-resistant join algorithm that replicates tuples corresponding to popular keys. We evaluate the effectiveness of our approach both on a synthetic benchmark known to generate complex queries (BSBM-BI) as well as on a Yahoo! case of data analysis using RDF data crawled from the web. Our results indicate that our system is indeed capable of processing huge datasets without pre-computed statistics while exhibiting good load-balancing properties.

1 Introduction

The amount of Linked Open Data (LOD) is strongly growing, both in the form of an ever expanding collection of RDF datasets available on the Web, as well semantic annotations increasingly appearing in HTML pages, in the form of RDFa, microformats, or microdata such as schema.org.

Search engines like Yahoo! crawl and process this data in order to provide more efficient search. Applications built on or enriched with LOD typically employ a warehousing approach, where data from multiple sources is brought together, and interlinked (e.g. as in [11]).

Due to the large volume and, often, dirty nature of the data, such Extraction, Transformation and Loading (ETL) processes can easily be translated into "killer" SPARQL queries that overwhelm the current state-of-the-art in RDF database systems. Such problems typically come down to formulating joins that produce huge results, or to RDF database systems that calculate the wrong join order such that the intermediate results get too large to be processed. In this paper, we describe a system that scalably executes SPARQL queries using the Pig Latin language [16] and we demonstrate its usage on a synthetic benchmark and on crawled Web data. We evaluate our method using both a standard cluster and the large MapReduce infrastructure provided by Yahoo!.

In the context of this work, we are considering some key issues:

Schema-less. A SPARQL query optimizer typically lacks all schema knowledge that a relational system has available, making this task more challenging. In a relational system, schema knowledge can be exploited by keeping statistics, such as table cardinalities and histograms that capture the value and frequency distributions of relational columns. RDF database systems, on the other hand, cannot assume any schema and store all RDF triples in a table with Subject, Property, Object columns (S,P,O). Both relational column projections as well as foreign key joins map in the SPARQL equivalent into self-join patterns. Therefore, if a query is expressed in both SQL and SPARQL, on the physical algebra level, the plan generated from SPARQL will typically have many more self-joins than the relational plan has joins. Because of the high complexity of join order optimization as a function of the number of joins, SPARQL query optimization is more challenging than SQL query optimization.

MapReduce and Skew. Linked Open Data ETL tasks which involve cleaning, interlinking and inferencing have a high computational cost, which motivates our choice for a MapReduce approach. In a MapReduce-based system, data is represented in files and that can come from recent Web crawls. Hence, we have an initial situation *without* statistics and *without* any B-trees, let alone multiple B-trees. One particular problem in raw Web data is the high skew in join keys in RDF data. Certain subjects and properties are often re-used (most notorious are RDF Schema properties) which lead to joins where certain key-values will be very frequent. These keys do not only lead to large intermediate results, but can also cause one machine to get overloaded in a join job and hence run out of memory (and automatic job restarts provided by Hadoop will fail again). This is indeed more general than joins: in the sort phase of MapReduce, large amounts of data might need to be sorted on disk, severely degrading performance.

SPARQL on Pig. The Pig Latin language provides operators to scan data files on a Hadoop cluster that form tuple streams, and further select, aggregate, join and sort such streams, both using ready-to-go Pig relational operators as well as using user-defined functions (UDFs). Each MapReduce job materializes the intermediate results in files that are replicated in the distributed filesystem. Such materialization and replication make the system robust, such that the jobs of failing machines can be restarted on other machines without causing a query failure. However, from the query processing point of view, this materialization is a source of inefficiency. The Pig framework attempts to improve this situation by compiling Pig queries into a minimal amount of Hadoop jobs, effectively combining more operators in a single MapReduce operation. An efficient query optimization strategy must be aware of it and each query processing step should minimize the number of Hadoop jobs.

Our work addresses these challenges and proposes an efficient translation of some crucial operators into Pig Latin, namely joins, making them robust enough to deal with the issues typical of large data collections.

Contributions. We can summarize the contributions of this work as follows. *(i)* We have created a system that can compute complex SPARQL queries on huge RDF datasets. *(ii)* We present a runtime query optimization framework that is optimized for Pig in that it aims at reducing the number of MapReduce jobs, therefore reducing query latency. *(iii)* We describe a skew-resistant join method that can be used when the runtime query optimization discovers the risk for a skewed join distribution that may lead to structural machine overlap in the MapReduce cluster. *(iv)* We evaluate the system on a standard cluster and a Yahoo! Hadoop cluster of over 3500 machines using synthetic benchmark data, as well as real Web crawl data.

Outline. The rest of the paper is structured as follows. In Section 2 we present our approach and describe some of its crucial parts. In Section 3 we evaluate our approach on both synthetic and real-world data. In Section 4 we report on related work. Finally, in Section 5, we draw conclusions and discuss future work.

2 SPARQL with Pig: overview

In this section, we present a set of techniques to allow efficient querying over data on Web-scale, using MapReduce. We have chosen to translate the SPARQL 1.1 algebra to Pig Latin instead of making a direct translation to a physical algebra in order to readily exploit optimizations in the Pig engine. While this work is the first attempt to encode full SPARQL 1.1 in Pig, a complete description of such process is elaborate and goes beyond the scope of this paper.

The remaining of this section is organized as follows: in Sections 2.1 and 2.2, we present a method for runtime query optimization and query cost calculation suitable for a batch processing environment like MapReduce. Finally, in subsection 2.3, we present a skew detection method and a specialized join predicate suitable for parallel joins under heavy skew, frequent on Linked Data corpora.

2.1 Runtime query optimization

We adapt the ROX query optimization algorithm [1, 10] to SPARQL and MapReduce. ROX interleaves query execution with join sampling, in order to improve result set estimates. Our specific context differs to that of ROX in that:

- SPARQL queries generate a large number of joins, which often have a multi-star shape [5].
- The overhead of starting MapReduce jobs in order to perform sampling is significant. The start-up latency for *any* MapReduce job lies within tens of seconds and minutes.
- Given the highly parallel nature of the environment, executing several queries at the same time has little impact on the execution time of each query.

Algorithm 1 Runtime optimization

```
1   $J$: Set of join operators in the query
2   $L$: List of sets of (partial) query plans
3   void optimize joins($J$) {
4    execute(J)
5    $L_0$:=(J)
6    $i$:=1
7    while ($L_{i-1} \neq \emptyset$)
8     for (j $\in L_{i-1}$)
9      for ($I \in \{L_0 ... L_{i-1}\}$)
10      for (k $\in I$)
11       if (j $\neq$ k)
12        $L_i$.add(construct(j,k))
13     if (stddev(cost($L_i$))/mean(cost($L_i$)) $>$ t)
14      prune($L_i$)
15      sample($L_i$)
16     $i$:=$i+1$
17  }
```

Algorithm 1 outlines the basic block of our join order optimization algorithm. To cope with the potentially large number of join predicates in SPARQL queries, we draw from dynamic programming and dynamic query optimization techniques, constructing the plans bottom-up and partially executing them.

Initially, we extract from the dataset the binding for all the statement patterns in the query and calculate their cardinalities. From initial experiments, given the highly parallel nature of MapReduce, we have concluded that the cost of this operation is amortized over the execution of the query since we are avoiding several scans over the entire input. Then, we perform a series of *construct-prune-sample* cycles. The construct phase generates new solutions from the partial solutions in the previous cycles. These are then pruned according to their estimated cost. The remaining ones are sampled and/or partially executed. The pruning and sampling phases are optional. We will only sample if $stddev(costs)/mean(costs)$ is higher than some given threshold, so as to avoid additional optimization overhead if the cost estimates for the candidate plans are not significantly different.

Construct During the construct phase (lines 6-10 in Algorithm 1), the results of the previous phases are combined to generate new candidate (partial) plans. A new plan is generated by either adding an argument to an existing node when possible (e.g. making a 2-way join a 3-way join) or by creating a new join node.

Prune We pick the k% cheapest plans in the previous phase, using the cost calculation mechanism described in Section 2.2. The remaining plans are discarded.

Sample To improve the accuracy of the estimation, we fully execute the plan up to depth 1 (i.e. the entire plan minus the upper-most join). Then, we use Algorithm 2 to perform bi-focal sampling [6].

Algorithm 2 Bi-focal sampling in Pig

```
DEFINE bifocal_sampling(L, R, s, t)
 RETURNS FC {
LS = SAMPLE L s;
RS = SAMPLE R s;
LSG = GROUP LS BY S;
RSG = GROUP RS BY S;
LSC = FOREACH LSG GENERATE flatten(group), COUNT(LSG) as c;
RSC = FOREACH RSG GENERATE flatten(group), COUNT(RSG) as c;
LSC = FOREACH LSC GENERATE group::S as S ,c as c;
RSC = FOREACH RSC GENERATE group::S as S ,c as c;
SPLIT LSC INTO LSCP IF c>=t, LSCNP IF c<t;
SPLIT RSC INTO RSCP IF c>=t, RSCNP IF c<t;
// Dense
DJ = JOIN LSCP BY S, RSCP BY S using 'replicated';
DJ = FOREACH RA GENERATE LSCP::c as c1, RSCP::c as c2;
// Left sparse
RA = JOIN RSC BY S, LSCNP BY S;
RA = FOREACH RA GENERATE LSCNP::c as c1, RSC::c as c2;
// Right sparse
LA = JOIN LSC BY S, RSCNP BY S;
LA = FOREACH LA GENERATE LSC::c as c1, RSCNP::c as c2;
// Union results
AC = UNION ONSCHEMA DA, RA, LA;
$FC = FOREACH AC GENERATE c1*c2 as c;}
```

There is a number of salient features in our join optimization algorithm:

– There is a degree of speculation, since we are sampling only after constructing and pruning the plans. We do not select plans based on their calculated cost using sampling, but we are selecting plans based on the cost of their 'sub-plans' and the operator that will be applied.
– Nevertheless, our algorithm will not get 'trapped' into an expensive join, since we only fully execute a join after we have sampled it in a previous cycle.
– Since we are evaluating multiple partial solutions at the same time, it is essential to re-use existing results for our cost estimations and to avoid unnecessary computation. Since the execution of Pig scripts and our run-time optimization algorithm often materialize intermediate results anyway, the latter are re-used whenever possible.

2.2 Pig-aware cost estimation

Using a MapReduce-based infrastructure gives rise to new challenges in cost estimation. First, queries are executed in batch and there is significant overhead in starting new batches. Second, within batches, there is no opportunity for sideways information passing [14], due to constraints in the programming model. Third, when executing queries on thousands of cores, load-balancing becomes very important, often outweighing the cost for calculating intermediate results. Fourth, random access to data is either not available or very slow since there are no data indexes. On the other hand, reading large portions of the input is relatively cheap, since it is an embarrassingly parallel operation.

In this context, we have developed a model based on the cost of the following: Writing a tuple (w); Reading a tuple (r); The cost of a join per tuple. In Hadoop, a join can be performed either during the reduce phase (j_r), essentially a combination of a hash-join between machines and a merge-join on each machine, or during the map phase (j_m), by loading one side in the memory of all nodes, essentially a hash-join. Obviously, the latter is much faster than the former, since it does not require repartitioning of the data on one side or sorting, exhibits good load-balancing properties, and requires that the input is read and written only once; The depth of the join tree(d), when considering only the reduce-phase joins. This is roughly proportional to the number of MapReduce jobs required to execute the plan. Considering the significant overhead of executing a job, we consider this separately from reading and writing tuples.

The final cost for a query plan is calculated as the weighted sum of the above, with indicatory weights being 3 for w, 1 for r, 10 for j_r, 1 for j_m and a value proportional to the size of the input for d.

2.3 Dealing with Skew

The significant skew in the term distribution of RDF data has been recognized as a major barrier to efficient parallelization [12]. In this section, we are presenting a method to detect skew and a method for load-balanced joins in Pig.

Detecting skew To detect skew (and estimate result set size), we are presenting an implementation of bi-focal sampling [6] for Pig and report the pseudocode in Algorithm 2. Similar to join optimization, one of the main goals is to minimize the number of jobs. L, R, s and t refer to the left side of the join, the right side of the join, the sampling rate and the number of tuples that the memory can hold respectively. Initially, we sample the input (lines 3-4), group by the join keys (lines 6-7) and count the number of occurrences of each key (lines 7-10). We split each side of the join by key popularity using a fixed threshold, which is dependent on the amount or memory available to each processing node (lines 11-12). We then perform a join between tuples with popular keys (lines 14-15) and a join for each side for tuples with non-popular keys and the entire input (lines 17-21).

This algorithm generates seven MapReduce jobs out of which two are Map-only jobs that can be executed in parallel and four are jobs with Reduce phases that happen concurrently in pairs. In fact, it is possible to implement our algorithm in two jobs, programming directly on Hadoop instead of using Pig primitives.

Determining join implementation In Pig, it is up to the developer to choose the join implementation. In our system, we choose according to the following:

– If all join arguments but one fit in memory, then we perform a replicated join. Replicated joins are performed on the Map side by loading all arguments except for one into main memory and streaming through the remaining one.
– If we have a join with more than two arguments and more than one of them does not fit in memory, we are performing a standard (hash) join.
– If the input arguments or the results of the (sampled) join present significant skew, we perform the skew-resistant join described in the following section.

Skew-resistant join As a by-product of the bi-focal sampling technique presented previously, we have the term distribution for each side of the join and an estimate of the result size for each term. Using this information, we can estimate the skew as the ratio of the maximum number of results for any key to the average number of results over all keys. Hadoop has some built-in resistance to skewed joins by means of rescheduling jobs to idle nodes, which is sufficient for cases where some jobs are slightly slower than others. Furthermore, Pig has a specialized join predicate to handle a skewed join key popularity [7], by virtue of calculating a key popularity histogram and distributing the jobs according to this. Nevertheless, neither of these algorithms can effectively handle skewed joins where a very small number of keys dominates the join. We should further note that, since there is no communication between nodes after a job execution has started, a skewed key distribution will cause performance problems even if the hit rate for those keys is low. This is because MapReduce will still need to send all the tuples corresponding to these popular keys to a single reduce task.

Our algorithm executes a replicated join for the keys that have a highly skewed distribution in the input and a standard join for the rest. In other words, joining on keys that are responsible for load unbalancing is done by replicating one side on each machine and performing a local hash join. For the remaining keys, the join is executed by grouping the two sides by the join key and assigning the execution of each group to a different machine (as is standard in Pig). In Algorithm 3, we present the Pig Latin code for an example join of expressions A and B[5], on positions O and S respectively. Initially, we sample and extract the top-k popular terms for each side (lines 3-12), PopularA and PopularB respectively. Then, for each side of the input, we perform two left joins to associate tuples with PopularA and PopularB (lines 14-17). This allows us to split each of our inputs to three sets (lines 19-25), marked accordingly in Figure 1:

1. The tuples that correspond to keys that are popular on the other side (e.g. for expression A the keys in PopularB). For side B, we put an additional requirement, namely that the key is *not* in PopularB. This is done to avoid producing the results for tuples that are popular twice.
2. The tuples that correspond to keys that are popular on the same side (e.g. for expression A the keys in PopularA).
3. The tuples that do not correspond to any popular keys on either side.

[5] for brevity, we have omitted some statements that project out columns that are not relevant for our algorithm

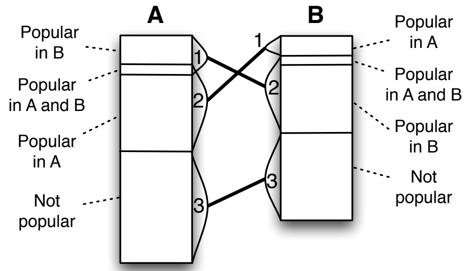

Fig. 1: Schematic representation of the joins to implement the skew-resistant join

We use the above to perform replicated joins for the tuples corresponding to popular terms and standard joins for the tuples that are not. The tuples in A corresponding to popular keys in A (APopInA) are joined with the tuples in B that correspond to popular keys in A using a replicated join (line 28). The situation is symmetric for B (line 29). The tuples that do not correspond to popular keys from either side are processed using a standard join (line 31). The output of the algorithm is the union of the results of the three joins.

We should note that our algorithm will fail if APopInB and BPopInA are not small enough to be replicated to all nodes. But this can only be true if there are some keys that are popular in both sets. Joins with such keys would anyway lead to an explosion in the result set (since the result size of each of these popular keys is the product of their appearances in each side).

3 Evaluation

We present an evaluation of the techniques presented in this paper using synthetic and real data, and compare our approach to a commercial RDF store.

We have used two different Hadoop clusters in our evaluation: a modest cluster, part of the DAS-4 distributed supercomputer, and a large cluster installed at Yahoo!. The former was used to perform experiments in isolation and consists of 32 dual-core nodes, with 4GB of RAM and dual HDDs in RAID-0. The Yahoo! Hadoop cluster we have used in our experiment consists of over 3500 nodes, each with two quad-core CPUs, 16 GB RAM and 900GB of local space. This cluster is used in a utility computing fashion and thus we do not have exclusive access, meaning that we can not exploit the full capacity of the cluster and our runtimes at any point might be (negatively) influenced by the jobs of other users. We thus only report actual, but not best possible performance.

In order to compare our approach with existing solutions, we deployed Virtuoso v7 [4], a top-performing RDF store, on an extremely high-end server: a 4-socket 2.4GHz Xeon 5870 server (40 cores) with 1TB of main memory and 16 magnetic disks in RAID5, running Red Hat Enterprise Linux 6.0.

We chose two datasets for the evaluation. Firstly, the Berlin SPARQL benchmark [3], Business Intelligence use-case v3.1 (BSBM-BI). This benchmark consists of 8 analytical query patterns from the e-commerce domain. The choice for

Algorithm 3 Skew-resistant join

```
DEFINE skew_resistant_join(A, B, k)
 RETURNS result {
SA = SAMPLE A 0.01; //Sample first side
GA = GROUP SA BY O;
GA2 = FOREACH GA GENERATE COUNT_STAR(SA), group;
OrderedA = ORDER GA2 BY $0 DESC;
PopularA = LIMIT OrderedA k;
SB = SAMPLE B 0.01; //Sample second side
GB = GROUP SB BY S;
GB2 = FOREACH GB GENERATE COUNT_STAR(SB), group;
OrderedB = ORDER GB2 BY $0 DESC;
PopularB = LIMIT OrderedB k;

PA = JOIN A BY O LEFT, PopularA BY O USING 'replicated';
PPA= JOIN PA BY O LEFT, PopularB BY S USING'replicated';
PB = JOIN B BY S LEFT, PopularB BY S USING 'replicated';
PPB= JOIN PB BY S LEFT, PopularA BY S USING 'replicated';

SPLIT PPA INTO APopInA IF PopularA::O is not null,
   APopInB IF PopularB::S is not null, ANonPop IF
   PopularA::0 is not null and PopularB::S is not null;

SPLIT PPB INTO BPopInB IF PopularB::S is not null, BPopInA
   IF PopularA::O is not null and PopularB::S is null, BNonPop
   IF PopularA::0 in not null and PopularB::S is not null;

 // Perform replicated joins for popular keys
 JA= JOIN BPopInB BY S, APopInB BY O USING 'replicated';
 JB= JOIN APopInA BY S, BPopInA BY S USING 'replicated';
// Standard join for non−popular keys
 JP = JOIN ANonPop BY O, BNonPop BY S;

$result = UNION ONSCHEMA JA, JB, JP; }
```

Query	1B DAS4	1B Y!	10B Y!
1	38m	1h50m	1h17m
2	13m	23m	31m
3	17m	18m	24m
4	38m	1h34m	54m
5	1h4m	3h10m	1h52m
6	48m	34m	1h19m
7	26m	43m	46m
8	1h	1h59m	1h38m

(a) Execution time of the BSBM queries on 1B data on the DAS-4 and Yahoo! cluster

Query	Cold runtime	Warm runtime
1	3m46s	1m48s
2	41s	15s
3	29m	24m5s
4	52m6s	50m55s
5	11m42s	5m19s
6	6s	0.06s
7	50s	9ms
8	39m58s	36m22s

(b) Runtime of the BSBM queries using Virtuoso

Fig. 2: BSBM query execution time

this benchmark is based on the scope of this work, namely complex SPARQL queries from an analytical RDF workload.

Secondly, we also used our engine for some analytical queries on RDF data that Yahoo! has crawled from the Web. This data is a collection of publicly available information on the Web encoded or translated in RDF. The dataset that we used consists of about 26 billion triples that correspond to about 3.5 terabytes of raw data (328 gigabytes compressed).

3.1 Experiments

In our evaluation, we measured: (i) *the performance of our approach for large datasets.* To this end, we launched and recorded the execution time of all the queries on BSBM datasets of 1 and 10 billion triples. (ii) *The effectiveness of our dynamic optimization technique.* To this purpose, we measured the cost of this process and what is its effect on the overall performance. (iii) *The load-balancing properties of our system.* To this end, we have performed a high-level evaluation of the entire querying process in terms of load balancing, and we have further focused on the performance of the skew-resistant join, which explicitly addresses load-balancing issues.

General performance. We have launched all 8 BSBM queries on 1 billion triples on both clusters and on 10 billion triples using only the Yahoo! cluster.

In Figure 2a, we report the obtained runtimes. We make the following observations: First of all, the fastest queries (queries 2 and 3) have a runtime of a bit less than 20 minutes on the DAS-4 cluster. The slowest is query 5, with a runtime of about one hour. For the 1B-triple dataset, the execution times on the Yahoo! cluster are significantly higher than those on the DAS-4 cluster. This is due to (a) the fact that the Yahoo! cluster is shared with other users, so, we often need to wait for the jobs of other users to finish execution and (b) the much larger size of the Yahoo! cluster, introducing additional coordination overhead.

Listing 1.1: Exploratory SPARQL queries

```
SELECT (count(?s) as ?f) (min(?s) as ?ex) ?ct ?di ?mx ?mi{
    {SELECT ?s (count(?s) as ?ct) (count(distinct ?p) as ?di) (max(?p) as ?mx)
        (min(?p) as ?mi) {?s ?p ?o.} GROUP BY ?s}
    GROUP BY ?ct ?di ?mx ?mi ORDER BY desc(?f) LIMIT 10000}

SELECT ?p (COUNT(?s) AS ?c) {?s ?p ?o.} GROUP BY ?p ORDER BY ?c

SELECT ?C (COUNT(?s) AS ?n ) {?s a ?C.} GROUP BY ?C ORDER BY ?n
```

We also note that the runtime does not proportionally increase with the data size on the Yahoo! cluster: the runtimes for the one and ten billion-triple datasets are comparable. Such behavior is explained by the fact that a proportional amount of resources is allocated to the size of the input and the (significant) overhead to start MapReduce jobs does not increase. Our approach, combined with the large infrastructure at Yahoo! allows us to scale to much larger inputs while keeping the runtime fairly constant.

To verify the performance in a real-world scenario and on messy data, we have launched three non-selective SPARQL queries, reported in Listing 1.1, over an RDF web crawl of Yahoo!. The first query is used for identifying "characteristic sets" [13]: frequently co-occurring properties with a subject. The second identifies all the properties used in the dataset and sorts them according to their frequency. The third identifies the classes with the most instances. These queries are typical of an exploratory ETL workload on messy data, aimed at creating a basic understanding of the structure and properties of a web-crawled dataset.

From the computation point of view, the first two queries have non-bound properties and the last one has a very non-selective property (*rdf:type*) . Therefore they will touch the entire dataset, including problematic properties that cause skew problems. The runtime of these three queries was respectively 1 hour and 21 minutes, 29 minutes and 25 minutes. The first query required 6 MapReduce jobs to be computed. The second and third each required 3 jobs.

It is interesting to compare the performance for BSBM against a standard RDF store (Virtuoso), even if the approaches are radically different. We loaded 10 billion BSBM triples on the platform described previously. This process took 61 hours (about 2.5 days) and was performed in collaboration with the Virtuoso development team to ensure that the configuration was optimal.

We executed the 8 BSBM queries used in this evaluation and we report the results in Figure 2b. For some queries, Virtuoso is many orders of magnitude faster than our approach (namely, for the simpler queries like queries 1,2, 6 and 7). For the more expensive queries, the difference is less pronounced. However, this performance comes at the price of a loading time of **61 hours**, necessary to create the database indexes. To load the data and run all the queries on Virtuoso, the total runtime amounts to 63 hours, while in our MapReduce approach, it amounts to 8 hours and 40 minutes. Although we can not generalize this con-

Query	Cost input extraction	Cost dyn. optimizer	Final query execution
1	5m58s	12m12s	19m43s
2	4m22s	n.a.	9m5s
3	5m46s	n.a.	11m58s
4	6m22s	14m41s	16m55s
5	7m37s	34m3s	23m21s
6	8m12s	14m25s	25m10s
7	5m17s	6m17s	14m3s
8	8m7s	25m14s	27m2s

Table 1: Breakdown of query runtimes on 1B BSBM data

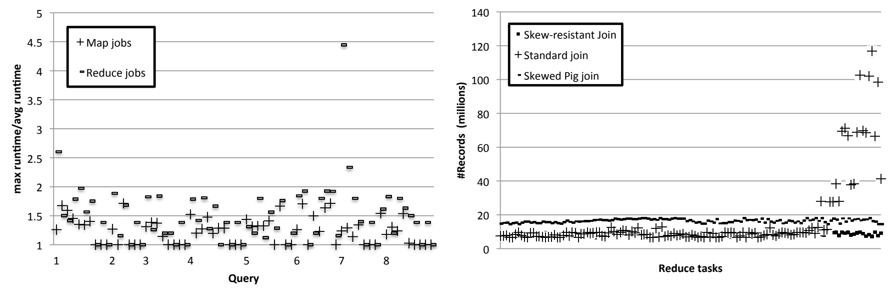

Fig. 3: Maximum task runtime divided by the average task runtime for a query mix (left). Comparison of load distribution between the skew-resistant join and the standard Pig join (right).

clusion to other datasets, the loading cost of Virtuoso is not amortized for a single query mix in BSBM-BI.

In this light, the respective advantages of the two systems are in running many cheap queries for Virtuoso and running a limited number of expensive queries for our system. Furthermore, our system can exploit existing Hadoop installations and run concurrently with heterogenous workloads.

Dynamic optimizer. As discussed in Section 2.1, query execution consists of three phases: (a) triple pattern extraction, (b) best execution plan identification using dynamic query optimization and (c) full query plan execution.

In Table 1, we summarize the execution time of each of these three phases for the 8 BSBM queries on the DAS-4 cluster, using the one billion triple dataset. We observe that extracting the input patterns is an operation that takes between 4 and 8 minutes, depending on the size of the patterns. Furthermore, the cost of dynamic optimization is significant compared to the total query execution time and largely depends on the complexity of the query, although, in our approach, part of the results are calculated during the optimization phase.

Load balancing. As the number of processing nodes increases, load-balancing becomes increasingly important. In this section, we present the results concerning the load balancing properties for our approach.

Due to the synchronization model of MapReduce, it is highly desirable that no task takes significantly longer than others. In Figure 3 (left), we present the maximum task execution time, divided by the average execution time for all tasks launched for a query mix on the DAS-4 cluster. The x axis corresponds both to time and the queries that correspond to each job, the y axis corresponds to the time it took to execute the slowest task divided by the time it took to execute a task, on average. In a perfectly load-balanced system, the values in y would be equal to 1. In our system, except for a single outlier, the slowest tasks generally take less than twice the average time, indicating that the load balancing of our system is good. A second observation is that the load imbalance is higher in the reduce jobs. This is expected, considering that, in our system, the map tasks are typically concerned with partitioning the data (and process data chunks of equal size), while the reduce tasks process individual partitions.

Data skew becomes increasingly problematic as the number of processing nodes increases, since it generates unbalance between the workload of each node [12]. In the set of experiments described in this section, we analyze the performance of the skew-resistent join that we have introduced in Section 2.3 to efficiently execute joins on data with high skew.

To this purpose, we launched an expensive join using the 1 billion BSBM dataset and we analyzed the performance of the standard and the skew-resistant join. Our experiments were performed on the DAS-4 cluster, since we required a dedicated cluster to perform a comparative analysis. Considering that this platform uses only 32 nodes in total, the effect on the Yahoo! cluster would have been much more pronounced (since it is several orders of magnitude bigger).

We launched a join that used the predicate of the triples as a key; namely, we have performed a join of pattern *(?s ?p ?o)* with pattern *(?p ?p1?o1)*. Such joins are common in graph analysis, dataset characterization [13] and reasoning workloads (e.g. RDFS rules 2-3 and 7).

The runtime using the classical join was of about 1 hour and 29 minutes. On the other side, the runtime using the skew-resistant join was about 57 minutes. Therefore, such a join has a significant impact on performance, in the presence of skew. The impact is even higher if we consider that the skew-resistant join requires 21 jobs to finish while the classical job requires a single one.

The reason behind such increase of performance lies in the way the join is performed, and in particular, the amount and distribution of work that the reduce tasks need to do, as reported in Figure 3 (right). We see that some reducers receive a much larger number of records than others (these are the ones at the end of the x axis), implying that some nodes will need to perform much more computation than others. With the skew-resistant join, all the joins among popular terms are performed in the map phase and as a result, all the reducers receive a similar amount of tuples in the input.

We also report reduce task statistics for the the skew-resistant join implementation in Pig, which calculates a histogram for join keys to better distribute them across the join tasks. Although the size of the cluster is small enough to ensure an even load-balance, we note that the standard Pig skewed join sends almost double the number of records to each reduce task. This is attributed to fact that our approach shifts much of the load for joining to the Map phase.

4 Related Work

We have compared our approach with previous results from three related areas: (i) MapReduce query processing (ii) adaptive and sampling-based query optimization and (iii) cluster-aware SPARQL systems.

In the relational context, similar efforts towards SQL query processing over a MapReduce cluster are e.g. Hive [21] and HadoopDB [2]. Both projects do not provide query optimization when data is raw and unprocessed. Data import is a necessary first step in HadoopDB and may be costly. In Hive, query optimization based on statistics is only available if the data has been analyzed as a prior step.

On-the-fly query optimization in Manimal [9] analyzes MapReduce jobs on-the-fly and tries to enhance them by inserting compression code and sometimes even on-the-fly indexing and index lookup steps.

Situations where there is absence of data statistics in the relational context of query optimization has led to work on sampling and run-time methods. Our work reuses the bi-Focal sampling algorithm [6] which came out of the work in the relational community to use sampling for query result size estimation. In this work, we have adapted the bi-focal algorithm using the Pig language.

The rigid structure of MapReduce and high latencies in starting new jobs led us to adjust the dynamic re-optimization strategies to these constraints. Other interesting run-time approaches are sideways-information passing [14] in large RDF joins. These are not easily adaptable to the constraints of MapReduce.

With the ever growing sizes of RDF data available, scalability has been a primary concern and major RDF systems such as Virtuoso [4], 4store [18], and BigData [20] have evolved to parallel database architectures targeting cluster hardware. RDF systems typically employ heavy indexing, going as far as creating replicated storage in all six permutations of triple order [15, 22], which makes data import a heavy process. Such choice puts them in a disadvantage when the scenario involves processing huge amounts of raw data. As an alternative to the parallel database approach, there are several other projects that process SPARQL queries over MapReduce. PIGSparQL [19] performs a direct mapping of SPARQL to Pig without focusing on optimization. RAPID+ [17] provides a limited form of on-the-fly optimization where *look-ahead processing* tries to combine multiple subsequent join steps. The adaptiveness of this approach is however limited compared to our sampling based run-time query optimization.

5 Conclusions

In this paper, we have presented an engine for the processing of complex analytic SPARQL 1.1 queries, based on Apache Pig. In particular, we have developed: (i) a translation from SPARQL 1.1 to Pig Latin, (ii) a method for runtime join optimization based on a cost model suitable for MapReduce-based systems, (iii) a method for result set estimation and join key skew detection, and (iv) a method for skew-resistant joins written in Pig. We have evaluated our approach on a standard and a very large Hadoop cluster used at Yahoo! using synthetic benchmark data and real-world data crawled from the Web.

In our evaluation, we established that our approach can answer complex analytical queries over very large synthetic data (10 Billion triples from BSBM) and over the largest real-world messy dataset in the literature (26 Billion triples). We compared our performance against a state-of-the-art RDF store on a large-memory server, even though the two approaches bear significant differences. While our approach is not competitive in terms of query response time, our system has the advantage that it does not require a-priori loading of the data, and thus has far better loading plus querying performance. Furthermore, our system runs on a shared architecture of thousands of machines, significantly easing deployment and potentially scaling to even larger volumes of data. We verified that the load in our system is well-balanced and our skew-resistant join significantly outperforms the standard join of Pig for skewed key distributions in the input.

In this work, for the first time, it has been shown that MapReduce is suited for very large-scale analytical processing of RDF graphs and it is, in fact, better suited than a traditional RDF store in a setting where a relatively small number of queries will be posted on a very large dataset.

We see future work in optimizing our architecture to further reduce overhead. This could be achieved by turning to an approach that adaptively indexes part of the input or performs part of the computation outside of Pig so as to reduce the number of jobs. Similarly, parts of the skew-resistant join can be already calculated during Bi-focal sampling (e.g. sampling and extracting the popular terms for the input relations).

Although in this paper we have presented our algorithm to handle skewed joins in the context of Pig, we expect that the result is transferrable to a general parallel data-processing framework.

Summarizing, in this paper, we have presented a technique with which technologies like MapReduce and Pig can be employed for large-scale SPARQL querying. The presented results are promising and set the lead for a new viable alternative to traditional RDF stores for executing expensive analytical queries on large volumes of RDF data.

Acknowledgements. *Spyros Kotoulas and Jacopo Urbani were partially supported by EU project LarKC (FP7-215535) and the Yahoo! Faculty Research Engagement Program. Peter Boncz was partially supported by EU project LOD2 (FP7-257943).*

References

1. R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: run-time optimization of XQueries. *SIGMOD*, 2009.
2. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.
3. C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.
4. O. Erling. Virtuoso, a Hybrid RDBMS/Graph Column Store. *DEBULL*, 35(1):3–8, 2012.
5. M. Gallego, J. Fernández, M. Martínez-Prieto, and P. Fuente. An empirical study of real-world SPARQL queries. In *USEWOD2011* at *WWW* 2011.
6. S. Ganguly, P. Gibbons, Y. Matias, and A. Silberschatz. Bifocal sampling for skew-resistant join size estimation. *SIGMOD Record*, 25(2):271–281, 1996.
7. A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *PVLDB*, 2(2):1414–1425, 2009.
8. M. Ivanova, M. Kersten, N. Nes, and R. Gonçalves. An architecture for recycling intermediates in a column-store. *TODS*, 35(4):24, 2010.
9. E. Jahani, M. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *PVLDB*, 4(6):385–396, 2011.
10. R. Kader, M. van Keulen, P. Boncz, and S. Manegold. Run-time Optimization for Pipelined Systems. *Proceedings of the IV Alberto Mendelzon Workshop on Foundations of Data Management*, 2010.
11. G. Kobilarov, T. Scott, Y. Raimond, S. Oliver, C. Sizemore, M. Smethurst, C. Bizer, and R. Lee. Media meets semantic web — how the BBC uses DBpedia and linked data to make connections. ESWC 2009.
12. S. Kotoulas, E. Oren, F. van Harmelen, and F. van Harmelen. Mind the data skew: distributed inferencing by speeddating in elastic regions. *WWW*, 2010.
13. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. *ICDE*, 2011.
14. T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. *SIGMOD*, 2009.
15. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.
16. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. *SIGMOD*, 2008.
17. P. Ravindra, V. Deshpande, and K. Anyanwu. Towards scalable RDF graph analytics on MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, page 5. ACM, 2010.
18. M. Salvadores, G. Correndo, S. Harris, N. Gibbins, and N. Shadbolt. The design and implementation of minimal RDFS backward reasoning in 4store. *The Semanic Web: Research and Applications*, pages 139–153, 2011.
19. A. Schätzle and G. Lausen. PigSPARQL: mapping SPARQL to Pig Latin. *SWIM: the 3th International Workshop on Semantic Web Information Management*, 2011.
20. L. SYSTAP. Bigdata®.
21. A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive: a petabyte scale data warehouse using Hadoop. *ICDE*, 2010.
22. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.