LODOP – Multi-Query Optimization for Linked Data Profiling Queries

Benedikt Forchhammer[1], Anja Jentzsch[1], and Felix Naumann[1]

Hasso-Plattner-Institute firstname.lastname@hpi.uni-potsdam.de

Abstract. The Web of Data contains a large number of different, openly-available datasets. In order to effectively integrate them into existing applications, meta information on statistical and structural properties is needed. Examples include information about cardinalities, value patterns, or co-occurring properties. For Linked Datasets such information is currently very limited or not available at all. Data profiling techniques are needed to compute respective statistics and meta information. However, current state of the art approaches can either not be applied to Linked Data, or exhibit considerable performance problems.
We present LODOP, a framework for computing, optimizing, and benchmarking data profiling techniques based on MapReduce with Apache Pig. We implemented 15 of the most important data profiling tasks, optimized their simultaneous execution, and evaluate them with four typical datasets from the Web of Data. Our optimizations focus on reducing the amount of MapReduce jobs and minimizing the communication overhead between multiple jobs. Our evaluation shows the significant potential in optimizing the runtime costs for Linked Data profiling.

1 Introduction

Over the past years, an increasingly large number of datasets has been published as part of the Web of Data. This trend, together with the inherent heterogeneity of datasets and their schemata, makes it increasingly time-consuming to find and understand datasets that are relevant for integration. In order for users to be able to integrate Linked Data, they first need an easy way to discover and understand relevant datasets on the Web of Data.

Data profiling is an umbrella term for methods that compute meta-data for describing datasets [1]. Traditional data profiling tools for relational databases have a wide range of features ranging from the computation of cardinalities, such as the number of values or distinct values in a column, to the calculation of inclusion dependencies between multiple columns or sets of columns; they calculate histograms on numeric values, determine value patterns, and gather information on used data types; some tools also determine the uniqueness of column values, and find and validate keys and foreign keys.

Use cases for data profiling can be found in various areas concerned with data processing and data management [1].

Query optimization is concerned with finding optimal execution plans for database queries. Cardinalities and value histograms can help to estimate the costs of such execution plans. Such metadata can also be used in the area of Linked Data, e.g., for optimizing SPARQL queries.

Data cleansing can benefit from discovered value patterns. Violations of detected patterns can reveal data errors, and respective statistics help measure and monitor the quality of a dataset. For Linked Data, data profiling techniques help validate datasets against vocabularies and schema properties, such as value range restrictions.

Data integration is often hindered by the lack of information on new datasets. Data profiling metrics reveal information on, e.g., size, schema, semantics, and dependencies of unknown datasets. This is a highly relevant use case for Linked Data, because for many openly available datasets only little information is available[1].

Schema induction: Raw data, e.g., data gathered during scientific experiments, often does not have a known schema at first; data profiling techniques need to determine adequate schemata, which are required before data can be inserted into a traditional DBMS. For the field of Linked Data, this applies when working with datasets that have no dereferencable vocabulary. Data profiling can help induce a schema from the data, which then can be used to find a matching existing vocabulary or create a new one.

The process of running data profiling tasks for large Linked Datasets can take hours to days, depending on the complexity of task and the size of the respective datasets. Data set characteristics highly influence the profiling task runtime. As an example, our *Property Cooccurrence by Resource* script (see Sec. 3) runs 16 hours for only 1 million triples of the Web Data Commons RDFa dataset in contrast to 5 min on Freebase and 9 min on DBpedia.

We have compiled a list of 56 data profiling tasks implemented in Apache Pig to be executed on Apache Hadoop. At this point Apache Pig only applies some basic logical optimization rules, like removing unused statements [2]. We present Lodop, a framework for executing, optimizing, and benchmarking such a set of profiling tasks, highlight reasons for poor performance when executing the scripts sequentially, and develop a number of optimization techniques. In particular, we developed and evaluated three multi-script optimization rules for combining logical operators in the execution plans of profiling scripts.

2 Related Work

While many tools and algorithms already exist for data profiling in general, most of them can, unfortunately, not be used for graph datasets, because they assume a relational data structure, a defined schema, or simply cannot deal with very large datasets. Nonetheless, some Linked Data profiling tools already exist. Most of them focus on solving specific use cases instead of data profiling in general.

[1] `http://lod-cloud.net/state/#data-set-level-metadata`

One relevant use case is schema induction, because not having a fixed and well-defined schema is a common problem with Linked Datasets. One example for this field of research is the ExpLOD tool [3]. ExpLOD creates summaries for RDF graphs based on class and predicate usage. It can also help understand the amount of interlinking between datasets based on the `owl:sameAs` predicate. Li describes a tool that can induce the actual schema of an RDF dataset [4]. It gathers schema-relevant statistics like cardinalities for class and property usage, and presents the induced schema in a UML-based visualization. Its implementation is based on the execution of SPARQL queries against a local database. Like ExpLOD, the approach is not parallelized in any way. It is evaluated with different datasets of up to 13M triples and reportedly faster than ExpLOD. However, both solutions still take approximately 10h to process a 10M triples dataset with 13 classes and 90 properties. These results illustrate performance as a common problem with large RDF datasets, and indicate that there is a need for parallelized, distributed execution.

An example for the query optimization use case is presented in [5]. The authors present RDFStats, which uses Jena's SPARQL processor for collecting statistics on RDF datasets. These include histograms for subjects (URIs, blank nodes) and histograms for properties and associated ranges. These statistics are used to optimize query execution on the (discontinued) Semantic Web Integrator and Query Engine. Others have worked more generally on generating statistics that describe datasets on the Web of Data and thereby help understanding them. LODStats computes statistical information for datasets from The Data Hub[2] [6]. It calculates 32 simple statistical criteria, e.g., cardinalities for different schema elements and types of literal values (e.g., languages, value data types). Approximation techniques are used when memory limits are reached. No detailed information about the performance of LODStats is reported.

In [7] the authors use MapReduce to automatically create VoID descriptions for large datasets. They manage to profile the Billion Triple Challenge 2010 dataset in about an hour on Amazon's EC2 cloud, showing that parallelization can be an effective approach to improve runtime when profiling large amounts of data. Finally, the ProLOD++ tool allows to navigate an RDF dataset via an automatically computed hierarchical clustering [8] and along its ontology [9]. Data profiling tasks are performed on each cluster independently, which serves not only as a means to derive meaningful results, but also improves efficiency.

3 Linked Data Profiling Tasks

This section lists and explains a set of useful data profiling tasks to profile Linked Data sets. We have implemented a total of 56 data profiling scripts, which compute 15 different statistical properties across different subsets of the input dataset. These subsets are determined via the following types of groupings:

Overall: no grouping.

[2] `http://datahub.io`

Resource: grouping based on the triple subject; usually paired with some *top-k* list of resources.

Property: grouping based on the property type.

Class: grouping based on the value of the `rdfs:type` predicate on a resource.

Class & Property: grouping based on both class and property type.

Datatype: grouping based on the (declared) data type of the object value; only typed literals are considered.

Language: grouping based on the (declared) language of the object value; only language literals are considered.

Context URL: grouping based on the N-Quads context attribute. Values being aggregated based on the context URL are grouped three times: based on the full URL, the pay-level-domain (PLD) part of the URL, and the top-level-domain (TLD) part of the UR.

Vocabulary: grouping based on the vocabularies used for predicates or classes.

Object URI: grouping based on the value of the object. Only URI values are considered.

The following data profiling tasks are computed. Unless stated differently, examples have been generated from a 1 million triples subset of DBpedia. More information on datasets can be found in Section 6.

Number of triples for each of the following groupings: overall, resource, property, object URI, context URL. This highlights, e.g., that the largest resources in our DBpedia subset is `http://dbpedia.org/resource/2010-11_SC_Freiburg_season` with 867 triples, and the most used property type is `rdfs:type` with 86,856 triples. (7 scripts)

Average number of triples per resource within each of the following groupings: overall, context URL, class. This reveals that resources in our DBpedia subset are made up of 5.9 triples on average, and that some classes, such as `http://dbpedia.org/ontology/YearInSpaceflight`, have resources with an average number of 502 triples. (5 scripts)

Average number of triples per object URI highlights how often certain URI objects are used across the dataset. For our DBpedia subset, each URI object is (re-)used as the object value for an average of 2.6 triples. For our 1 million triples subset of Freebase, this number is lower (1.9), i.e., fewer URI object values are reused. (1 script)

Average number of triples per URL computes the average number of triples per value of the graph context URL. For our DBpedia subset, each graph context contains an average of 47.8 triples. This means that, in this case, multiple resources share the same graph context, because the average number of triples per resource is only 5.9 as mentioned above. (1 script)

Number of property types within each of the following groupings: overall, context URL, resource, class, data type. This tasks tells us how many property types are used in different contexts; our DBpedia subset has a total of 7,844

different property types, of which 4,970 are used on resources of the class `owl:Thing`. It also reveals that 2,929 property types point to triples having typed object values with the `xsd:int` datatype. (7 scripts)

Average number of property values per property type (i.e., triples per predicate) within each of the following groupings: overall, context-URL, resource, class, property, class, property. For example, resources having the class `http://umbel.org/umbel/rc/Event` have an average number of 115 values for the `http://dbpedia.org/property/time` predicate in our DBpedia subset. (8 scripts)

Number of resources in each of the following groupings: property, class, datatype, language, vocabulary. On our subset of DBpedia, these profiling tasks tell us that there are a total of 169,035 resources, that the `owl:sameAs` property type is used on 43,840 resources, and the `http://xmlns.com/foaf/0.1/Person` class on 4,140 resources. (6 scripts)

Number of context URLs in each of the following groupings: property, class, vocabulary. For DBpedia the context URL usually matches to the corresponding Wikipedia page. This task can reveal how many Wikipedia pages lead to certain classes; for example, the `foaf:Person` class occurs in 4,140 different context URLs in our subset of data. (3 scripts)

Number of context PLDs in each of the following groupings: property, class, vocabulary. Alternative script versions additionally group by the TLD of the context URL. Similar to the previous task, this task can tell us how many different pay-level domains are responsible for certain properties, classes, or vocabularies. For our WDC RDFa subset (1m triples) this reveals, e.g., that the `http://www.facebook.com/2008/` predicate is used by 452 domains, which makes it the most-used property type in this subset. (6 scripts)

Property co-occurrence computes a list of pairs of property types that are used together: a) on the same resource (by resource), b) within the same context URL (by url), or c) pointing to the same resource (by object URI). This reveals patterns, such as `http://dbpedia.org/ontology/artist` and `http://dbpedia.org/property/title` being used together on resources describing artistic work. (3 scripts)

Inverse properties computes a list of properties that are inverse to each other. For example, the relationship between a musician and his band can be declared on both resources with two inverse property types `http://dbpedia.org/ontology/associatedBand` and `http://dbpedia.org/ontology/bandMember`. (1 script)

URI-literal ratio computes the ratio between the number of URI values and literals within the following groupings: overall, class, context URL. Our DBpedia subset contains almost as many URI object values as it contains literal values (ratio 1.1); however, the RDFa subset contains far more literals than URI values (ratio 0.2). (5 scripts)

Property value ranges of literal values per property type (numeric and temporal values only). For example, the `http://dbpedia.org/ontology/bedCount` property has numeric values between 76 and 785 in our DBpedia subset. (2 scripts)

Average value length of literals per property type can reveal schema properties. For example, the `http://dbpedia.org/ontology/title` property has values with an average length of 34 characters, whereas values for the `http://dbpedia.org/property/longTitle` property have an average of 288 characters. (1 script)

Number of inlinks computes the number of URI values pointing to other resources within the same dataset/file. For DBpedia, there are 207,712 values pointing to resources within the same dataset; for our WDC RDFa subset, there are far fewer inlinks (35,329). (1 script)

Each of these profiling tasks have been implemented as an Apache Pig script and are availabe at `https://github.com/bforchhammer/lodop`. The runtime of these scripts even on 1 million triples might take up to hours, e.g., for the property co-occurrence determination. Also, scripts often have the same preprocessing steps, e.g., filtering or grouping the dataset. Thus there is a large incentive and potential to optimize the execution of multiple scripts.

4 Multi-query optimization for Apache Pig

A prevalent goal for relational database optimization is to reduce the amount of required full table scans, which for file-based database systems effectively means reducing the amount of disk operations. Sellis introduces Multi-Query Optimization for relational databases as the process of optimizing a set of queries which may share common data [10]. The goal is to execute these queries together and reduce the overall effort by executing similar parts only once. The optimization process consists of two parts: identifying shared parts in multiple queries and finding a globally optimal execution plan that avoids superfluous computation.

Apache Pig[3] is a platform for performing large-scale data transformations on top of Apache Hadoop clusters. It provides a high-level language (called *Pig Latin*) for specifying data transformations, e.g., selections, projections, joins, aggregations and sorting on datasets. Pig Latin scripts are compiled into a series of MapReduce tasks and executed on a cluster.

The main goals for our multi-query optimization rules for Pig are the following two: First, we attempt to minimize the dataflow between operators. In our evaluation (Sec. 6) we identified the dataflow between MapReduce jobs as a reasonable indicator for the performance of Pig scripts, as it is closely related to the amount of required HDFS operations. Second, we try to avoid performing identical or similar operations multiple times. The idea behind this is to free up cluster resources for other tasks. All optimization rules presented in this section, are based on optimizing the logical plans of Pig scripts.

Three optimization rules have been implemented: Rule 1 merges identical operators in logical plans of different scripts, Rule 2 combines `FILTER` operators, and Rule 3 combines aggregations, i.e., `FOREACH` operators. Rule 1 is a prerequisite for the other two rules, which work on pairs of siblings operators, i.e.,

[3] `http://pig.apache.org/`

operators that have the same parent operator in a respective logical plan. For all optimization rules, it was important to make sure that their usage does not affect the intended output of scripts.

Rule 1 – Merge identical operators: In order to better utilize cluster resources, it makes sense to submit jobs to Apache Hadoop in parallel. Lodop supports this by merging logical plans of different scripts into a single large plan. In our experiments, executing scripts in parallel as part of one large plan cuts execution time down to 25-30% of the time required to execute scripts sequentially. Once all plans have been merged together, it's possible to also merge identical operators.For 52 of our Pig scripts, this reduces the number of operators from 365 to 267.

Rule 2 – Combine filters: `FILTER` operators reduce the amount of data that needs to be processed in later steps of the execution pipeline. This optimization rule aims to avoid iterating over large sets multiple times. From our selection of profiling scripts, 25 scripts perform filtering operations on the full initial dataset.

First, we identify all suitable sibling filters, i.e., all `FILTER` operators that have the same parent operator. Second, a combined filter is created and we attach it to the same parent operator. This combined filter contains all boolean expressions of existing filters concatenated via `OR`. The expression of the combined filter is cleaned up by transforming it into disjunctive normal form. Finally, we rearrange all previous filters and move them after the combined filter.

Rule 3 – Combine aggregations: `FOREACH` operators can be used for projections and aggregations. Some instances perform identical aggregations, but project different properties. This can happen, e.g., if the aggregation itself is only a preprocessing step to another aggregation. These operators are not exactly identical, so the rule for merging identical operators will not be able to merge them. However, these cases can be optimized by separating the aggregation from the projection, i.e., performing the aggregation only once with all projected columns, and then projecting the exact columns afterwards. For our set of scripts, this rule can be applied in seven different cases and combines varying numbers of `FOREACH` operators from the minimum of two to a maximum of eleven siblings operators.

While our goal is to optimize the performance of profiling tasks, the optimization rules can be applied on any Pig script.

5 The LODOP System

Lodop comprises four major components. Figure 1 shows how they interact with the Hadoop cluster. The *Compiler* is responsible for compiling Pig Latin scripts into Logical Plans and eventually into MapReduce jobs; the *Optimiser* takes care of optimizing logical plans; the *ScriptRunner* schedules and monitors the execution of jobs on the Hadoop cluster; finally, the *Reporting* component turns raw statistics into human-readable formats.

Lodop is built to be easily configurable via command line arguments, allowing the execution of different combinations of datasets, scripts, and optimization

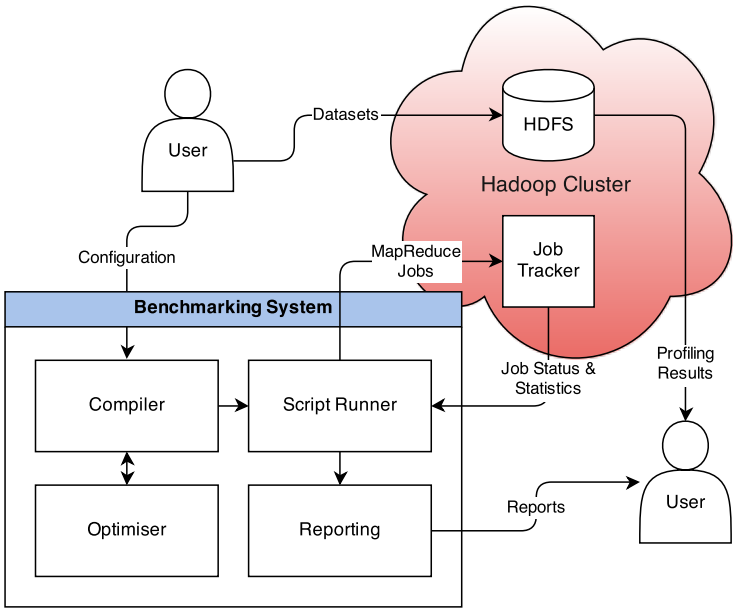

Fig. 1. LODOP component overview

rules. The benchmarking system parses the arguments supplied by the user, and passes them on to the compiler. The main responsibility of the compiler is to make sure that Apache Pig plays well with the optimization component. It takes over Pig's standard script compilation workflow, and adds in missing features, such as the support for multiple scripts and custom optimization rules.

The compiler first loads the declared list of scripts and compiles them into respective logical plans. This compilation step is largely taken care of by Apache Pig's compiler, which parses Pig Latin and builds a respective directed acyclic graph of logical operations. Once these logical plans have been created, our standardized loading and storage functions are injected into each plan. The loading function assumes that all scripts work with the same input file type and schema (N-Quads files). This is an essential prerequisite for merging different scripts together in the optimization component. Similarly, outputs are also handled automatically, by always storing the result of the last statement in each script. The respective storing function writes results directly to Hadoop's distributed file system (HDFS).

Rule-based optimization is now done in two steps: First, all existing logical plans are merged into one large logical plan. At this point, each script's logical data flow is still separate from other scripts, but this allows the ScriptRunner component to submit jobs for different scripts at the same time, and thus execute them in parallel. Second, optimization rules are applied to the merged logical plan. The rules function similar to other optimization rules that are already part of Pig: First the logical plan is searched for applicable patterns in order to gain a list of possible optimization targets. Each match is then checked to determine

whether the rule can be applied to the specific group of operators. Finally, if all checks are positive, the rule is applied and the logical plan adjusted accordingly.

LODOP currently does not perform cost-based optimization to determine whether a rule should be applied or not; we simply apply rules repeatedly until they cannot be applied any more. To ease debugging and help understand logical plans, the system additionally visualizes plans after different steps in the compilation and optimization process.

After the optimization step, logical plans are compiled into MapReduce plans and then handed to the ScriptRunner component. The ScriptRunner submits MapReduce jobs to the Hadoop cluster, monitors their execution, and gathers performance statistics. Scheduling and monitoring are handled by Apache Pig itself. Statistics on the performance of MapReduce jobs are provided by Hadoop and only need to be retrieved from the cluster.

6 Evaluation

In this section we evaluate the effect of applying the optimization rules of Sec. 4 and investigate under which circumstances the performance is improved. We evaluate on four selected Linked Data sets that provide a wide range of characteristics, ranging from cross-domain to domain-specific as well as ranging from well-defined to loosely defined semantics.

6.1 Datasets and experimental setup

DBpedia[4] is one of the largest Linked Data sets and contains structured information extracted from Wikipedia. As such it covers a wide range of different topics with a large number of property types and a very large number of classes compared to other datasets. Almost 50% of property values are URIs.

Freebase[5] is a community-maintained database of "well-known people, places, and things". It contains data harvested from sources such as Wikipedia, ChefMoz, MusicBrainz, and others. It has a fairly small average number of triples per resource (4.4), but the highest number of URI values.

The Web Data Commons[6] project extracts RDF triples directly from information embedded on websites via RDFa, Microdata, or Microformats. We focus on only the RDFa subset (roughly 500 million triples) which is mostly unstructured and has a small schema compared to other datasets. Most property values are literal values (83%) and there are very few in-links (3%). Any faulty definitions on crawled websites can lead to inconsistencies and errors in the triple graph, e.g., resources that do not conform to well-known schema definitions.

The species dataset of the European Environment Agency (EUNIS)[7] is part of a database on species, habitat types, and sites of interest for biodiversity.

[4] `http://dbpedia.org`
[5] `http://www.freebase.com`
[6] `http://webdatacommons.org`
[7] `http://eunis.eea.europa.eu`

The species subset is well-structured with only one class and 16 property types. Compared to other datasets it has a very large average number of triples per resource (15.2).

All evaluations were performed on an Apache Hadoop cluster with one head node and ten slaves. Each node has a 2-core processor (Core 2 Duo) with 2GB of memory and runs CentOS 5.5. We use Apache Hadoop 1.1.2 and Apache Pig 0.11.1 with Java 1.7. In order to explain our observations, we particularly look at the properties identified as relevant to performance, i.e., the number of operators and MapReduce jobs, as well as the amount of I/O activity on HDFS.

6.2 Base performance analysis

This section analyzes the overall runtime of scripts, when executed without any custom optimizations. We thus establish a baseline against which our custom optimization rules can be compared, and give insights into the reasons for current performance.

Figure 2 shows an overview of execution times for all scripts executed on 1 million triples of DBpedia, Freebase, WDC RDFa, and EUNIS. The figure shows that execution times vary depending on the dataset and the script. Overall, one can see that many scripts finish in under 5 minutes. Some scripts have to perform more complex computations and hence take longer to complete. This includes the scripts for computing property co-occurrence and the *URI-Literal ratio by Class* script. These UDF-based scripts dominate the overall execution time, are not amenable to our rules, and are thus excluded from further evaluation.

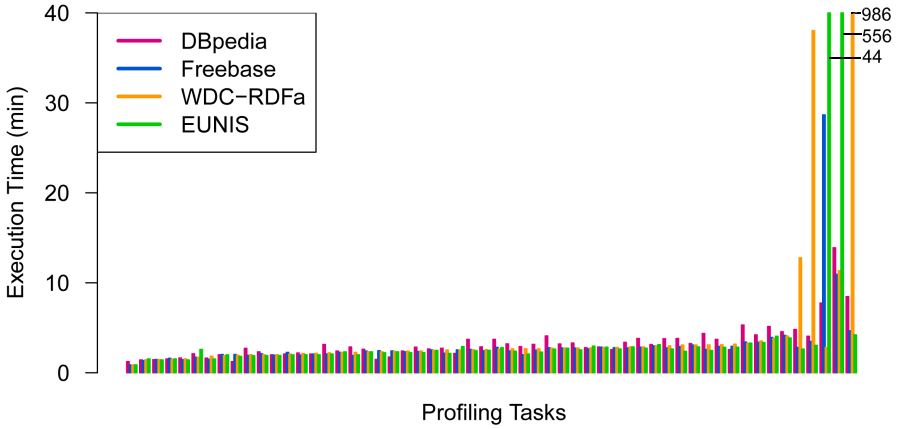

Fig. 2. Runtime overview for all datasets (1M triples)

6.3 Effectiveness of merging identical operators

As described in Section 5, Lodop supports two ways for executing multiple scripts: First, they can simply be loaded, compiled, and executed sequentially.

In this case, the monitoring and scheduling overhead of Apache Pig takes some additional time between jobs and also between scripts, during which the cluster is idle. In order to better utilize cluster resources, it makes sense to submit jobs to Apache Hadoop in parallel instead. LODOP supports this by merging logical plans of different scripts together into one large plan. By executing multiple jobs simultaneously, Apache Pig can start a MapReduce job, as soon as all its dependent jobs have finished. Figure 3 shows that this has a large impact on the overall runtime of the profiling process, cutting execution time down to 25-30% of the execution time required to execute scripts sequentially.

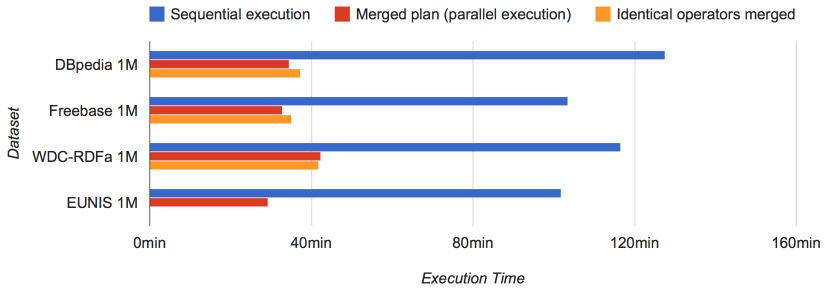

Fig. 3. Execution time for 52 scripts, sequential execution versus merged plan execution

Once all plans have been merged, it is possible to apply the first optimization rule, merge identical operators. For 52 scripts, this reduces the number of operators from 365 to 267. Table 1 shows that we are able to merge, amongst others, 14 `COGROUP`s, 12 `FOREACH`s and 1 `JOIN` into respective identical operators. In terms of MapReduce jobs, merging identical operators reduces the number of jobs from 176 jobs to 140.

Pig Operator	Defined in scripts	Identical operators merged
COGROUP	66	52
ORDER BY	44	44
STORE	52	52
JOIN	6	5
DISTINCT	15	11
FOREACH	98	86
FILTER	28	13
LOAD	52	1
UNION	4	3

Table 1. Number of operators before and after identical operators merged. (52 scripts)

Unfortunately, this operator merging has little to no effect on the overall performance, as can be seen in Figure 3. The following observations give reasons for this: First, we end up with a tree of MapReduce jobs after merging identical operators; while this was an intended effect of this optimization rule, it also means that there is only one root node, which all other jobs depend upon. Because Apache Pig tries to execute as many operators as possible on early Hadoop jobs, we end up with one very large job at the beginning of the workflow. The

reason for this is the process used by Apache Pig to translate logical plans into MapReduce plans: it generates one MapReduce job for each `COGROUP` operation, then moves most other operators into the reduce function of the previous job; all operators before the first `COGROUP` are pushed into the map function of the first job. Executing this large initial job can take up a significant amount of time. For example, for 1 million triples of the DBpedia subset, executing the merged root node takes almost 10 minutes (570s), which is about 30% of the total execution time for the merged plan. Compared to the simultaneous execution of unmerged scripts, this hinders parallelism.

Note that the large initial job is also the reason for the missing execution time value on the EUNIS species dataset in Figure 3: the respective cluster node ran out of memory during the computation of the respective first job.

Second, the amount of data being transferred between jobs can actually increase. When scripts are executed simultaneously in the merged plan, multiple jobs load and process the same input file. Most scripts in this case manage to reduce the input size significantly during their respective first job already. In contrast, when all identical operators are merged, only the root node loads the full input file. However, in this case 16 of the 36 children of the root job still need to work with the full number of input tuples as well. Most of them will have some columns projected out, so in terms of transferred bytes it is not the full dataset, but compared to executing scripts in parallel more data needs to be materialized on HDFS.

Merging identical operators is a prerequisite step to applying the remaining two optimization rules. However, it may be more beneficial to only combine identical operations based on a cost-based approach, which decides whether it is worth merging identical operators based on the possibility of applying other rules and based on an estimated performance gain.

6.4 Effectiveness of combining filters

In order to evaluate the effect of combining `FILTER` operators, we look at an example of two scripts: *Property value ranges (temporal)* and *Property value ranges (numeric)*. Both scripts load the full input dataset and filter out only triples with typed object values. The former script further restricts the set by temporal data types, the latter by numeric data types. The two can be optimized by filtering the input dataset only once and applying the additional restriction on the data type afterwards.

Most of the datasets evaluated have only a small set of triples with typed object values ($< 15\%$). Therefore, filtering the input dataset only once significantly reduces the selectivity for the additional selections. Statistics on the amount of HDFS I/O for this example show that the un-optimized plan reads almost twice as much data from HDFS than the optimized version (see Table 2). Yet Figure 4 shows that the effect on the overall execution time is small.

Even though the merging of filter operators does not improve on the amount of HDFS I/O compared to merging identical operators, the graph still shows a consistent improvement of about 10-20s, compared to only merging identical

Dataset	Merged Plan	Identical operators merged	Filter operators combined
DBpedia 10M	480MB / 1MB	241MB / 1MB	241MB / 1MB
Freebase 10M	194MB / 0MB	97MB / 0MB	97MB / 0MB
WDC-RDFa 10M	293MB / 0MB	146MB / 0MB	146MB / 0MB

Table 2. HDFS I/O statistics for both *Property value ranges* scripts.

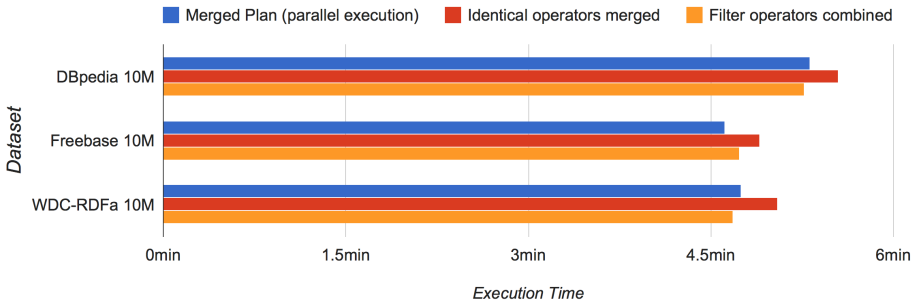

Fig. 4. Execution time for *Property value ranges* scripts.

operators. This can be explained by improved CPU time, i.e., less time spent on actual computation on the respective cluster node. In fact, the total amount of seconds spent on computations (additive, not considering simultaneous execution) is reduced noticeably: For 10M triples of the DBpedia subset, the CPU time is reduced from 390s for the merged plan to about 240s for the plan with merged identical operators and finally to about 220s for the plan with combined filter statements. While this presents an improvement compared to the plan with identical operators merged, it does not show a significant improvement on the runtimes of the merged plan, which executes scripts in parallel.

6.5 Effectiveness of combining aggregations

In order to evaluate the effect of combing `FOREACH` operators we regard three closely related example scripts: *Average number of property values by class*, *by property*, and *by class and property*. All three group the input dataset by subject and predicate and then count the number of triples in each group. The only difference in this aggregation step is the projected columns for each of the scripts.

The goal of combing aggregations is to reduce the amount of data required to be read from HDFS, by having only one combined `FOREACH` operator iterating over the full input dataset. Table 3 shows that this is achieved for the given example. Compared to the merged plan, the amount of data being read from as well as being written to HDFS is reduced considerably for all 10M datasets. As explained earlier, execution plans for which we only merged identical operators exhibit higher amounts of I/O activity. Figure 5 shows that this optimization rule improves overall runtime for all datasets by 1-2 minutes.

It should be noted that the amount of HDFS activity is improved far less for other combinations of scripts. So, while this example demonstrates the potential benefit of combining aggregations, it also motivates the need for an advanced, cost-based optimizer that applies optimizations only in certain cases.

Dataset	Merged Plan	Identical operators merged	Aggregations combined
DBpedia 10M	19.9GB / 14.6GB	22.4GB / 14.0GB	15.6GB / 11.8GB
Freebase 10M	11.7GB / 7.8GB	14.2GB / 7.0GB	8.8GB / 5.2GB
WDC-RDFa 10M	12.4GB / 7.5GB	15.4GB / 6.8GB	8.8GB / 4.6GB

Table 3. HDFS I/O statistics for *Average number of property values by class / property / class* and *property* scripts.

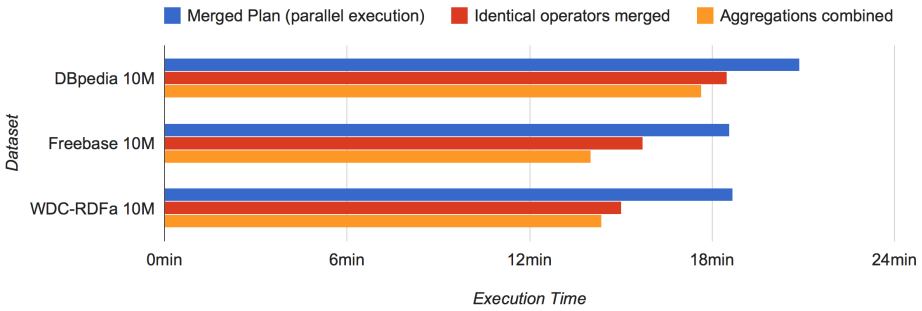

Fig. 5. Execution time for *Average number of property values by class / property / class and property* scripts.

6.6 Combined results

The number of MapReduce jobs and the amount of dataflow in the operator pipeline are good indicators for the performance of Apache Pig scripts. Our evaluation shows that improving only on these factors does not necessarily improve overall performance. Merging identical operators reduces both the total number of operators and the number of MapReduce jobs. It comes at the cost of less parallelism. Combining filter operators was shown to reduce the execution time of map/reduce functions (i.e., CPU time). Combining aggregations can reduce the amount of HDFS I/O, and improves overall execution time for certain combinations of scripts and datasets. Figure 6 shows execution times when optimizations are applied for all scripts. Overall in our experiments, executing scripts in parallel and applying all optimization rules cuts execution time down to 25-30% of the time required to execute scripts sequentially.

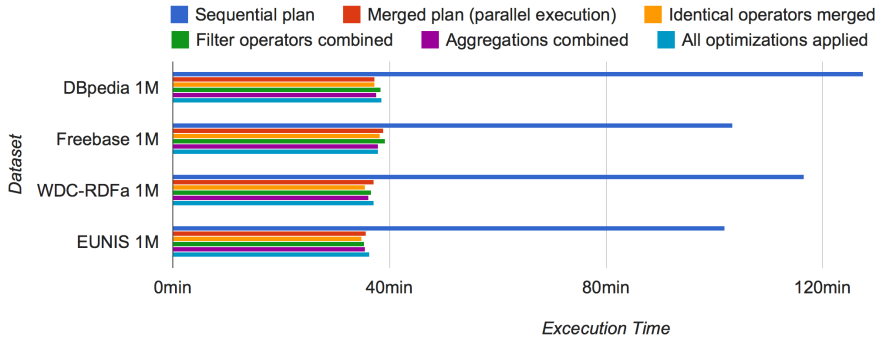

Fig. 6. Execution time for all optimizations (52 scripts)

7 Conclusion

Existing work on data profiling can either not be applied to the area of Linked Data or exhibits considerable performance problems. To overcome this gap we introduced a list of 56 data profiling tasks, implemented them as Apache Pig scripts, and analyzed the performance of executing them on an Apache Hadoop cluster. Thereby, we introduced three common techniques for improving performance, namely algorithmic optimization, parallelization and multi-script optimization. We experimentally demonstrated that they achieve their respective goals of optimizing the amount of MapReduce jobs or the amount of data materialized between jobs.

In addition to the optimization techniques described in this paper, other optimization rules could be implemented. For instance, simple projections could be ignored in the logical optimization plans to allow further merging of operators, and aggregations based on similar groupings could be optimised to reduce the amount of redundant computation. Further, an advanced optimization strategy should apply optimization rules only if the estimated overall performance gain is high enough and not as often as possible, as we do it now.

References

1. Naumann, F.: Data profiling revisited. SIGMOD Record **42**(4) (2013)
2. Gates, A., Dai, J., Nair, T.: Apache Pig's Optimizer. IEEE Data Eng. Bull. **36**(1) (2013) 34–45
3. Khatchadourian, S., Consens, M.P.: ExpLOD: Summary-based exploration of interlinking and RDF usage in the linked open data cloud. In: Proceedings of the Extended Semantic Web Conference (ESWC), Heraklion, Greece (2010) 272–287
4. Li, H.: Data Profiling for Semantic Web Data. In: Proceedings of the International Conference on Web Information Systems and Mining (WISM). (2012)
5. Langegger, A., Wöß, W.: RDFStats – An extensible RDF statistics generator and library. In: Proceedings of the International Workshop on Database and Expert Systems Applications (DEXA), Los Alamitos, CA, USA (2009) 79–83
6. Auer, S., Demter, J., Martin, M., Lehmann, J.: LODStats – an extensible framework for high-performance dataset analytics. In: Proceedings for the International Conference on Knowledge Engineering and Knowledge Management (EKAW). (2012)
7. Böhm, C., Lorey, J., Naumann, F.: Creating VoiD descriptions for web-scale data. Journal of Web Semantics **9**(3) (2011) 339–345
8. Böhm, C., Naumann, F., Abedjan, Z., Fenz, D., Grütze, T., Hefenbrock, D., Pohl, M., Sonnabend, D.: Profiling Linked Open Data with ProLOD. In: Proceedings of the International Workshop on New Trends in Information Integration (NTII). (2010)
9. Abedjan, Z., Grütze, T., Jentzsch, A., Naumann, F.: Mining and profiling RDF data with ProLOD++. In: Proceedings of the International Conference on Data Engineering (ICDE). (2014) Demo.
10. Sellis, T.K.: Multiple-query optimization. ACM Transactions on Database Systems (TODS) **13**(1) (1988) 23–52