Reuse-based Optimization for Pig Latin

Jesús Camacho-Rodríguez
Hortonworks
Santa Clara, CA, USA

Dario Colazzo
Université Paris-Dauphine,
PSL Research University,
CNRS, LAMSADE
Paris, France

Melanie Herschel
IPVS, University of Stuttgart
Stuttgart, Germany

Ioana Manolescu
INRIA & LIX, École
Polytechnique, CNRS
Palaiseau, France

Soudip Roy Chowdhury
Fractal Analytics
Mumbai, India

ABSTRACT

Pig Latin is a popular language which is widely used for parallel processing of massive data sets. Currently, subexpressions occurring repeatedly in Pig Latin scripts are executed as many times as they appear, and the current Pig Latin optimizer does not identify reuse opportunities.

We present a novel optimization approach aiming at identifying and reusing repeated subexpressions in Pig Latin scripts. Our optimization algorithm, named *PigReuse*, identifies subexpression merging opportunities, selects the best ones to execute based on a cost function, and reuses their results as needed in order to compute exactly the same output as the original scripts. Our experiments demonstrate the effectiveness of our approach.

Keywords

Reuse-based Optimization; Linear Programming; PigLatin

1. INTRODUCTION

The efficient processing of very large volumes of data has lately relied on massively parallel processing models, of which MapReduce is the most well known. However, the simplicity of these models leads to relatively complex programs to express even moderately complex tasks. Thus, to facilitate the specification of data processing tasks to be executed in a massively parallel fashion, several higher-level query languages have been introduced.

In this work, we consider Pig Latin [18], which has raised significant interest from the application developers as well as the research community. Pig Latin provides dataflow-style primitives for expressing complex analytical data processing tasks. Pig Latin programs (also named *scripts*) are automatically optimized and compiled into parallel processing jobs by the Apache Pig system [19], which is included in all leading Hadoop distributions e.g., HDP [10], CDH [4].

Part of this work was performed while the authors were with Université Paris-Sud and INRIA.

http://dx.doi.org/10.1145/2983323.2983669

In a typical batch of Pig Latin scripts, there may be many identical (or equivalent) sub-expressions, that is: script fragments applying the same processing on the same inputs, but appearing in distinct places within the same (or several) scripts. While the Pig Latin engine includes a query optimizer, it is currently not capable of recognizing such repeated subexpressions. As a consequence, they are executed as many times as they appear in the script batch, whereas there is obviously an opportunity for enhancing performance by identifying common subexpressions, executing them only once, and reusing the results in every script needing them.

Identifying and reusing common subexpressions occurring in Pig Latin scripts automatically is the target of the present work. The problem bears obvious similarities with the known multi-query optimization and workflow reuse problems; however, as we discuss in Section 6, the Pig Latin primitives lead to several novel aspects of the problem, which lead us to propose dedicated, new algorithms to solve them.

Motivating example. A Pig Latin script consists of a set of *binding expressions* and *store expressions*. Each binding expression var = op means that the expression op is evaluated, and the resulting bag of tuples is bound to the variable var, which can be used by follow-up expressions in a script.

Consider the following Pig Latin script a_1:

```
1 A = LOAD 'page_views' AS (user, time, www);
2 B = LOAD 'users' AS (name, zip);
3 R = JOIN A BY user, B BY name;
4 S = FOREACH R GENERATE user, time, zip;
5 STORE S INTO 'a1out1';
6 T = JOIN A BY user LEFT, B BY name;
7 STORE T INTO 'a1out2';
```

Line 1 loads data from a file `page_views` and creates a bag of tuples that is bound to variable `A`. Each of these tuples consists of three attributes `(user,time,www)`. Line 2 loads data from a second file, and binds the resulting tuple bag to `B`. Line 3 joins the tuples of `A` and `B` based on the equality of the values bound to attributes `user` and `name`. The next line uses the Pig Latin operator `FOREACH`, that applies a function on every tuple of the input bag. In this case, line 4 projects the attributes `user`, `time` and `zip` of every tuple in `R`. Then the result is stored in the file `a1out1`. In turn, line 6 executes a left outer join over the tuples of `A` and `B` based on the equality of the values bound to the same attributes `user` and `name`, and the result is stored in `a1out2`.

The following script a_2 only executes a left outer join over the same inputs:

```
1 A = LOAD 'page_views' AS (user, time, www);
2 B = LOAD 'users' AS (name, zip);
3 R = JOIN A BY user LEFT, B BY name;
4 STORE R INTO 'a2out';
```

The script b that we introduce next produces the same outputs as a_1 and a_2:

```
1  A = LOAD 'page_views' AS (user, time, www);
2  B = LOAD 'users' AS (name, zip);
3  R = COGROUP A BY user, B BY name;
4  S = FOREACH R GENERATE flatten(A), flatten(B);
5  T = FOREACH S GENERATE user, time, zip;
6  STORE T INTO 'a1out1';
7  U = FOREACH R GENERATE flatten(A),
8                 flatten (isEmpty(B) ? {(null,null,null)} : B);
9  STORE U INTO 'a1out2';
10 STORE U INTO 'a2out';
```

However, *b's execution time is* 45% *of the combined running time of* a_1 *and* a_2. The reason is twofold. First, observe that the joins are rewritten into a COGROUP[1] operation (line 3) and FOREACH operations (lines 4 and 7-8). The interest of cogroup is that through some simple restructuring, one can carve out of the cogroup output various flavors of joins (natural, outer, nested, semijoin etc.) This restructuring operation differs depending on whether we want to generate the join between A and B needed for script a_1 (line 4), or the left outer join between A and B for scripts a_1 and a_2 (lines 7-8). The detailed semantics of these restructuring operations will become clear in Section 4. Thus, the first reason for the speedup of b w.r.t. a_1 and a_2 is that the COGROUP output is reused to generate the result for both joins. The second reason is that in b, the left outer join is computed only once, and its result is used to produce the desired output of scripts a_1 (line 9) and a_2 (line 10).

A typical feature of our optimizer is that it works on the algebraic representation of Pig Latin scripts. Thus, it is orthogonal to the Pig Latin query evaluation and execution process. This allows our approach (*i*) to benefit from the Pig Latin optimizer, and (*ii*) to apply our optimization independently of the underlying Pig Latin query compilation and execution engines.

Contributions. The technical contributions of this work are the following.

- We propose PigReuse, a multi-query optimization algorithm that merges equivalent subexpressions it identifies in Directed Acyclic Graph of algebraic representation of a batch of Pig Latin scripts. PigReuse produces then an optimal merged plan where redundant computations have been eliminated, by relying on Binary Integer Linear Programming in order to select the best plan based on the provided cost function.
- We present techniques to improve *effectiveness* of our baseline PigReuse optimization approach.
- We have implemented PigReuse as an extension module within the Apache Pig system. We present an experimental evaluation of our techniques using two different cost functions to select the best plan.

Outline. Section 2 is dedicated to preliminaries. Section 3 presents the main techniques over which PigReuse relies on, while Section 4 presents strategies to enhance it. Section 5 describes our experimental evaluation. Finally, Section 6 discusses related work, and then we conclude.

[1] COGROUP is a generalization of the *group-by* operation on two or more relations: for every distinct value of the grouping key occurring in any of the inputs, it outputs a tuple that includes an attribute *group* bound to the grouping key, and a bag of tuples for each input R_i such that the bag R_i includes all tuples in R_i that contain the value of the grouping key.

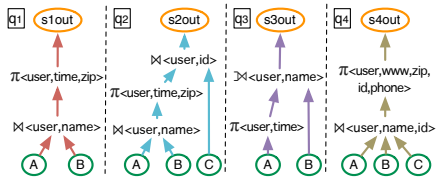
s_1
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
R = JOIN A BY user, B BY name;
S = FOREACH R GENERATE user, time, zip;
STORE S INTO 's1out';
```

s_2
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
C = LOAD 'power_users' AS (id, phone);
R = JOIN A by user, B BY name;
S = FOREACH R GENERATE user, time, zip;
T = JOIN S BY user, C by id;
STORE T INTO 's2out';
```

s_3
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
R = FOREACH A GENERATE user, time;
S = JOIN R by user LEFT, B by name;
STORE S INTO 's3out';
```

s_4
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
C = LOAD 'power_users' AS (id, phone);
R = JOIN A BY user, B by name, C by id;
S = FOREACH R GENERATE user, www, zip, id, phone;
STORE S INTO 's4out';
```

(a)



(b)

Figure 1: Sample Pig Latin scripts (a) and their corresponding algebraic DAG representation (b).

2. PRELIMINARIES

Since our approach strictly depends on rewriting Pig Latin expressions into equivalent ones, we rely on algebraic representation of Pig Latin scripts. Actually, the Pig Latin data model features complex data types (e.g., tuple, map etc.) and nested relations with duplicates (bags). Thus, we rely on the Nested Relational Algebra for Bags [8] (NRAB, for short) to represent Pig Latin scripts. We consider a subset of the NRAB algebra and extend it with other operators, such as *cogroup*. In turn, we have formalized (and implemented) the entire Pig Latin-to-NRAB translation process. Formal details of the translation are presented in [3].

We represent a set of NRAB binding expressions obtained from a Pig Latin program as a Directed Acyclic Graph (DAG); each binding is represented as a node, while edges represent the data flow among nodes.

To illustrate, Figure 1.a introduces four different Pig Latin scripts s_1-s_4; we will reuse them throughout the paper. The scripts read data from the three input relations page_views, users, and power_users; from now on, we denote these relations as A, B, and C.

After connecting the different algebraic expressions generated from s_1, we obtain the DAG query q_1 shown in Figure 1.b, also including the DAG-based representations of s_2-s_4. We denote the NRAB projection operator as π, while the selection operation is denoted as σ. In turn, inner join is denoted as $\bowtie$, while left outer join is denoted as $\rtimes$.

3. REUSE-BASED OPTIMIZATION

Based on the previously introduced representation of Pig Latin scripts as NRAB DAGs, we now introduce our PigReuse algorithm that optimizes the query plans corresponding to a batch of scripts by reusing results of repeated subexpressions. More specifically, given a collection of NRAB DAG queries Q, PigReuse proceeds in two steps:

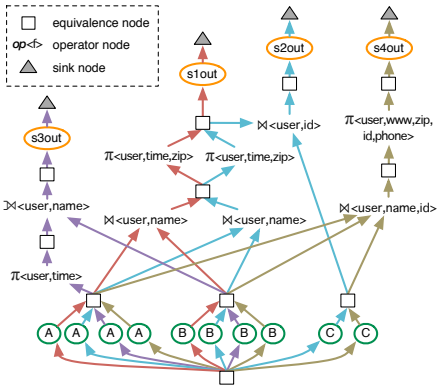


Figure 2: EG corresponding to NRAB DAGs q_1-q_4.

Step (1). *Identify and merge all the equivalent subexpressions in* Q. To this end, we use an AND-OR DAG, in which an AND-node (or operator node) corresponds to an algebraic operation in Q, while an OR-node (or equivalence node) represents a set of subexpressions that generate the same result.
Step (2). *Find the optimal plan from the AND-OR DAG.* Based on a cost model, we make a globally optimal choice of the set of operator nodes to be evaluated. Our approach is independent of the particular cost function chosen.

The final output of PigReuse is an optimized plan that contains (*i*) the operator nodes leading to *minimizing the cummulated cost* of all the queries in Q, while *producing, together, the same set of outputs* as the original Q, and (*ii*) equivalence nodes that represent *result sharing* of an operator node with other operators in Q. In the following sections, we describe each step of our reuse-based optimization algorithm in detail.

3.1 Equivalence-based merging

To join all detected equivalent expressions in Q, we build an AND-OR DAG, which we term *equivalence graph* (EG, in short); the construction is carried out in the spirit of previous optimization works [7, 21]. In the EG, an AND-node corresponds to an algebraic operation (e.g., selection, projection etc.). An OR-node o is introduced whenever a set of expressions $e_1, e_2, \ldots, e_k$ have been identified as equivalent; in the EG, the children of o are the algebraic nodes at the roots of the expressions $e_1, e_2, \ldots, e_k$. In the following, we refer to AND-nodes as *operator nodes*, and OR-nodes as *equivalence nodes*. Formally, we define an EG as follows.

Definition 1. An equivalence graph (EG) is a DAG, defined by the pair $(O \cup A \cup T_o, E)$, with O, A, and T_o disjoint sets of nodes, and:

- O is the set of equivalence nodes, A is the set of operator nodes, T_o is the set of sink nodes.
- $E \subseteq (O \times A) \cup (A \times O) \cup (A \times T_o)$ is a set of directed edges such that: each node $a \in A$ has an in-degree of at least one, and an out-degree equal to one; each node $o \in O$ has an in-degree of at least one, and an out-degree of at least one; each node $t_o \in T_o$ has an in-degree of at least one. ◇

Observe that in an EG, O nodes can only point to A nodes, while A nodes can point to O or T_o nodes.

Building the equivalence graph. To build the equivalence graph, we need to identify equivalent expressions within the input NRAB query set Q. We rely on the standard notion of query equivalence, i.e., two expressions are *equivalent* iff their result is provably the same regardless of the data on which they are computed. Our equivalence search algorithm is sound but not complete; see [3] for a detailed discussion.

Figure 2 depicts the EG corresponding to the NRAB DAGs q_1 to q_4 in Figure 1.b. In Figure 2, all the leaf nodes in the NRAB DAGs that correspond to the same *scan* operation (namely, nodes A, B, and C) feed the same equivalence node. The equi-joins coming from DAGs q_1 and q_2 on relations A and B over attributes *user* and *name* are also inputs to the same equivalence node.

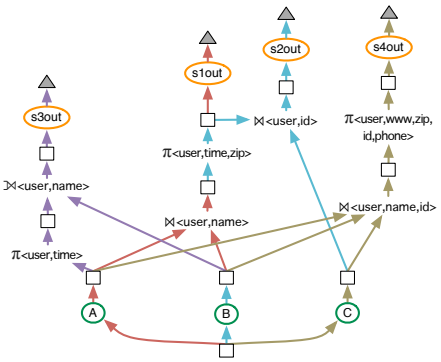


Figure 3: Possible REG for the EG in Figure 2.

3.2 Cost-based plan selection

Once an EG has been generated from a set of NRAB queries, our goal is to find the best alternative plan (having the smallest possible cost) computing the same outputs as the original scripts. We call the output plan a *result equivalence graph* (or REG, in short).

Definition 2. A result equivalence graph (REG) with respect to an EG defined by $(O \cup A \cup T_O, E)$ is itself a DAG, defined by the pair $(O^* \cup A^* \cup T_O, E^*)$ such that:

- $O^* \subseteq O$, $A^* \subseteq A$, $E^* \subseteq E$.
- The set of sink nodes T_o is identical in EG and REG. Each sink node has an in-degree of exactly one.
- Each operator node in-degree in EG and REG is equal. Each equivalence node has an in-degree of exactly one, and an out-degree of at least one. ◇

In the REG, we choose exactly one among the alternatives provided by each EG equivalence nodes; the REG produces the same outputs as the original EG, as all sink nodes are preserved. Further, each REG can be straightforwardly translated into a NRAB DAG which is basically an executable Pig Latin expression. The latter expression is the one we turn to Pig for execution.

The choice of which alternative to pick for each equivalence node is guided by a *cost function*, the overall goal being to minimize the global cost of the plan. We assign a cost (weight) to each edge $n_1 \rightarrow n_2$ in the EG, representing all the processing cost (or effort) required to fully build the result of n_2 out of the result of n_1.

Figure 3 shows a possible REG produced for the EG depicted in Figure 2. This REG can be obtained by using a cost function based on counting the operator nodes in the optimized script. In Section 5, we consider different cost functions and compare them experimentally.

3.3 Cost minimization

We model the problem of finding the minimum-cost REG relying on *Binary Integer Programming* (BIP), that has

Minimize $\mathcal{C} = \sum_{e \in E} \mathcal{C}_e x_e$ subject to:

$$x_e \in \{0,1\} \qquad \forall e \in E \qquad (1)$$

$$\sum_{e \in E^{in}_{t_o}} x_e = 1 \qquad \forall t_o \in T_o \qquad (2)$$

$$\sum_{e \in E^{in}_a} x_e = x_{E^{out}_a} \times |E^{in}_a| \qquad \forall a \in A \qquad (3)$$

$$\sum_{e \in E^{in}_o} x_e = \max_{e \in E^{out}_o} x_e \qquad \forall o \in O \qquad (4)$$

Figure 4: BIP reduction of the problem.

already been used to solve database optimization problems [12, 25]. Broadly speaking, a linear programming problem can be expressed as: **given** a set of linear inequality constraints over a set of variables **find** value assignments for the variables **such that** the value of an objective function depending on these variables is minimized. Many years of research and development efforts have led to the implementation of extremely efficient BIP solvers.

Generating the result equivalence graph. Given an input EG, for each node $n \in O \cup A \cup T_o$, we denote by E^{in}_n and E^{out}_n the sets of incoming and outgoing edges for n, respectively. For each edge $e \in E$, we introduce a variable x_e, denoting whether or not e is part of the REG. Since in our specific problem formulation any x_e takes values in $\{0,1\}$, our problem is formulated as a BIP problem. Further, for each edge $e \in E$, we denote by $\mathcal{C}_e$ the cost assigned to e by some cost function $\mathcal{C}$. Importantly, the model we present next is independent of the chosen cost function.

Our optimization problem is stated in BIP terms in Figure 4. Equation (1) states that each x_e variable takes values in $\{0,1\}$. (2) ensures that every output is generated exactly once. (3) states that if the (only) outgoing edge of an operator node is selected, all of its inputs are selected as well. This is required in order for the algebraic operator to be capable of correctly computing its results. Finally, (4) states that if an equivalence node is generated, it should be used at least once, which is modeled by means of a *max* expression (details about its BIP encoding can be found in [3]).

4. EFFECTIVE PIGREUSE

In the following, we present three extensions to the basic PigReuse algorithm which allow for identifying and exploiting additional factorization opportunities (formal details can be found in [3]).

Normalization of the input NRAB DAGs is carried out by *reordering* π operator nodes as follows: we push them away from *scan* operators or closer to *store* operators. We do this by visiting all operator nodes in a NRAB DAG, starting from a *scan*, and by moving each π operator up one level at a time if possible. Although pushing projections up through a plan is counterintuitive from the classical optimization point of view, *it increases the chances to find equivalent subexpressions*, as we will shortly illustrate. Further, after our reuse-based algorithm produces the optimized REG, we push the π operators back down to avoid the performance loss incurred by manipulating many attributes at all levels.

Since *cogroup* nests the input relations, reordering π with this operator requires complex rewriting. Specifically, we use the NRAB restructuring operator in the rewriting, denoted as $map\langle\varphi\rangle$, which applies a function φ to all tuples in the input. Thus, we rewrite the project π into a *map* that applies π on the corresponding bag of tuples in the input relation.

To illustrate the advantages of this phase, Figure 5.a shows the EG generated by PigReuse over the *normalized* NRAB DAGs q_1 to q_4. Comparing this EG with the one shown in Figure 2, we see that due to the swapping of the π operator corresponding to q_2, our algorithm can identify an additional common subexpression between q_2 and q_4, by determining the equivalence between the joins over A, B, and C; the corresponding equivalence node is highlighted.

Join decomposition. The semantics of Pig Latin's join operators e.g., $\bowtie$, $\rtimes$, $\ltimes$, or $\rtimes\ltimes$ allow rewriting these operators into combinations of *cogroup* and *map* operators. The advantage of decomposing the joins in this way is that the result of the *cogroup* operation, which does the heavy-lifting of assembling groups of tuples from which the *map* will then build join results, can be shared across different kinds of joins. The *map* will be different in each case depending on the join type, but the most expensive component of computing the join, namely the *cogroup*, will be factorized.

Figure 5.b shows the EG generated by PigReuse after applying *normalization and decomposition* to the NRAB DAGs q_1 to q_4. One can observe that the decomposition of the $\bowtie$ operators from q_1 and q_2, and the $\rtimes$ operator from q_3 leads to an additional sharing opportunity, as the result of the *cogroup* on attributes *user* and *name* can be shared by the subsequent *map* operations (highlighted equivalence node).

Observe that $\bowtie$ operators were rewritten into two operators. The first one is a *cogroup* on the attributes used by the join predicate. The second one is a *map* that does the following for each input tuple: (i) project each bag of tuples corresponding to the *cogroup* input relations; (ii) apply a NRAB bag destroy operation, denoted as δ, which unnests one level those bags; and (iii) perform a cartesian product $\times$ among the tuples resulting from unnesting those bags. If a bag is empty, e.g., the input relation did not contain any value for the given grouping value, the δ operator does not produce any tuple, and thus the tuples from the other bags for the given tuple are discarded. Thus, the rewriting produces the exact same result as the original operator.

For the left outer join $\rtimes$ rewriting, *cogroup* is followed by a *map* operator that (i) unnests the bag associated to the left input of the *cogroup*; (ii) if the bag associated to the right input ($\underline{\mathsf{var}}_2$) is empty, it replaces it with a bag with a null tuple, otherwise it keeps the bag as it is; (iii) unnests the bag resulting from the previous operation; and (iv) performs a cartesian product on the tuples resulting from the δ operations in order to generate the $\rtimes$ result.

Aggressive merge is based on the observation that it is possible to derive the results of a join or *cogroup* operator from the results of a *cogroup*$'$ operator, as long as the former relies on *a subset* of the input relations and attributes of *cogroup*$'$. This means that these rewritings rely on the notion of *cogroup containment*. In particular, this entails checking the containment relationship between respective sets of input relations and attributes. Then, in order to generate the result of the original $\bowtie$ or *cogroup* operator, we add the appropriate operator on top of *cogroup*$'$. Differently from previous extensions, *aggressive merge* is applied while creating the EG.

Figure 5.c depicts the new EG produced by PigReuse using aggressive merge; the new connections are highlighted. The figure shows how the results for the *cogroup*, $\bowtie$, and $\rtimes$ operators on A and B relations are derived from the *cogroup* operator on A, B, and C.

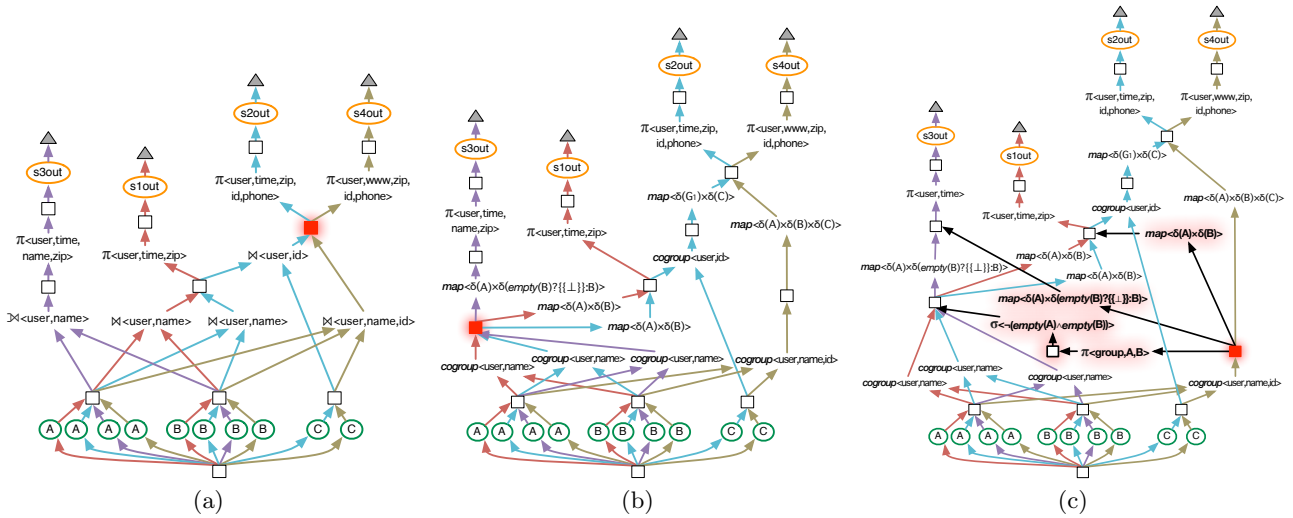


Figure 5: EG generated by PigReuse on (a) the *normalized* DAGs q_1-q_4, (b) the *normalized* and *decomposed* DAGs q_1-q_4, and (c) applying *aggressive merge* on the *normalized* and *decomposed* DAGs q_1-q_4.

5. EXPERIMENTAL EVALUATION

We have implemented PigReuse in Java 1.6. The source code amounts to about 8000 lines and 50 classes. It works on top of Apache Pig 0.12.1 [19], which relied on the Hadoop platform 1.1.2 [9]. The cost-based plan selection algorithm uses the Gurobi BIP solver 5.6.2 (www.gurobi.com).

Deployment. All our experiments run in a cluster of 8 nodes connected by a 1GB Ethernet. Each node has a 2.93GHz Quad Core Xeon processor and 16GB RAM, runs Linux CentOS 6.4, and has two 600GB SATA hard disks where HDFS is mounted.

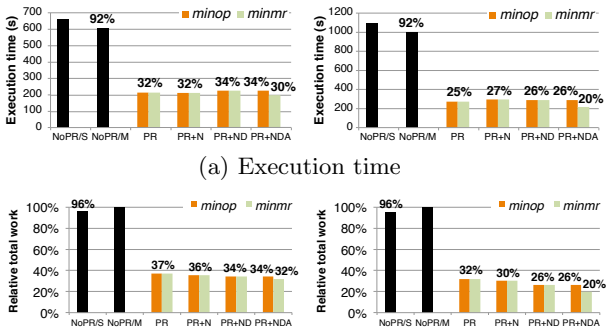
Setup. For validation, we used data sets and scripts provided by the PigMix [20] PigLatin performance benchmark. We created a `page_views` input file of 250 million rows; the benchmark includes other input files, which are based on the `page_views` file, and are much smaller than this one. The total size of the data set amounted to approximately **400 GB** before the 3-way replication applied by HDFS.

We run our algorithm with two different workloads, denoted W_1 and W_2, containing 12 and 20 Pig Latin scripts, respectively. Details about these workloads are given in [3].

Cost functions and experiment metrics. Currently we have implemented two cost functions in PigReuse, focusing on reducing the number of logical operators, and the number of MapReduce jobs, respectively. Beyond these two cost functions used by our PigReuse algorithm, we also quantify the performance of executing a PigLatin workload through the following standard metrics: the **Execution time** is the wall-clock time measured from the moment when the scripts are submitted to the Pig engine, until the moment their execution is completely finished; the **Total work** is the sum of the effort made on all the nodes, i.e., the total CPU time as returned by logs of the MapReduce execution engine.

Experimental results. We now study the benefits brought by the optimizations proposed in this work. The reported results are averaged over three runs.

Figure 6 shows the effectiveness of our baseline PigReuse algorithm (PR), PigReuse with *normalization* (PR+N), PigReuse with *normalization* and *decomposition* (PR+ND), and PigReuse applying all our extensions including *aggressive merge* (PR+NDA). The figure shows relative values for the execution time and total work metrics. The cost function that minimizes the total number of operators in the EG is denoted by *minop*, while the cost function that minimizes the total number of MapReduce jobs is denoted by *minmr*.



(a) Execution time

(b) Total work

Figure 6: PigReuse evaluation using workload W_1 (left) and W_2 (right).

In Figure 6.a, we notice that the total execution time is reduced by more than 70% on average among our PigReuse algorithms. Two alternative executions without PigReuse are shown. In the first one (NoPR/S), we execute *sequentially* every script in each workload using a single Pig client. In the second one (NoPR/M), we use multiple Pig clients that send *concurrently* the jobs resulting from the scripts to MapReduce. As it can be seen, the execution time for the second variant is lower as jobs resulting from multiple scripts are scheduled together, and thus the cluster usage is maximized. However, observe that the total work (Figure 6.b) increases for the multi-client alternative. The reason is that the scheduler cannot overlap significantly the map phases of multiple queries since there are too many map tasks. Thus, their execution remains quite sequential.

For the workloads we considered, our extensions reduced the total work over the baseline PigReuse algorithm (Figure 6.b). However, this was not always the case for the execution time (Figure 6.a). The reason is that some of the resulting plans requiring more effort had less sequential ex-

ecution steps, thus they could be parallelized easier by the MapReduce engine. When *aggressive merge* was applied, the execution time and the total work decreased only if the *minmr* cost function was used. The reason is that if the *minop* function is used, PigReuse generates the same REG for PR+ND and PR+NDA, namely, the REG with the minimum number of operators. However, with *minmr*, PigReuse chooses a plan with more operators but that runs faster.

6. RELATED WORK

Relational multi-query optimization. Our work directly relates to multi-query optimization (MQO). The early works [11, 22] proposed exhaustive, expensive algorithms which were not integrated with existing system optimizers. The technique presented in [21] was the first to integrate MQO into a Volcano-style optimizer, while [26] presents a completely integrated MQO solution accounting also for the usage and maintenance of materialized views. The approach of [23] takes into account the physical requirements of the consumers of common sub-expressions in order to propose globally optimal execution plans. While all of these works deals with the relational algebra, our approach optimizes workloads expressed in terms of the richer NRAB algebra including nesting. Further, differently from these approaches, ours can be directly applied to alternative implementations of Pig Latin or of any other language based on NRAB.

The above considerations still hold for the approach proposed in [6], which relies on sharing *physical* operators in large SQL workloads. The approach presented in [15] also addresses both global optimization and sharing possibilities at once, but it aims at optimizing a single query, while extensions to multi-query are not trivial and not explored so far. As pointed out in [6], the global optimization plus sharing problem can not be expressed by a linear program and thus a branch-and-bound heuristic solution is proposed, whereas our sharing problem can be solved optimally through BIP.

[14] optimizes SQL workloads at runtime and tries to maximize recycling opportunities of intermediary results. Due to the complexity constraints of considering multiple rewritings for each query, [14] only considers the optimized plan for each of them, in contrast to PigReuse. This design choice comes at the price of missing sharing opportunities.

Reuse-based optimizations on MapReduce. Recent works have sought to avoid redundant processing for a batch of MapReduce jobs by sharing their scans or intermediary results. Since the semantics of the computation is not visible at the level of MapReduce programs, these works are either limited to detecting identical inputs and outputs of MapReduce tasks (without being able to reason on task equivalence) [1, 16, 24], or need some annotations to the jobs to inform about sharing opportunities [5, 13]. Our PigReuse algorithm works on the higher-level semantic representation of Pig Latin scripts. This enables more complex reuse-based optimizations, e.g., through algebraic expression rewriting.

MQO for higher-level languages based on MapReduce has been considered in [2, 17]. For Hive workloads, [2] shows by example that improving replication of frequently used data, re-ordering queries in a workload, and scheduling queries in parallel can improve performance. In turn, [17] studies how to schedule Pig Latin programs in order to best profit from the shared computations. The core of our work, instead, is concerned with identifying common sub-expression of a workload, and examining the global sharing problem, which are not addressed in [17].

7. CONCLUSIONS

We have presented a novel reuse-based optimization approach for Pig Latin scripts. Our PigReuse algorithm identifies sub-expression merging opportunities, and selects the best ones to merge based on a cost-based search process implemented with the help of a linear program solver. PigReuse allows plugging any cost function and its output is a merged script reducing its value. Our experimental results demonstrate the effectiveness of our optimization strategies.

8. REFERENCES

[1] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *PVLDB*, 2008.

[2] P. Alvaro, N. Conway, and A. Krioukov. Multi-query optimization for parallel dataflow systems, 2009.

[3] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. R. Chowdhury. PigReuse: A Reuse-based Optimizer for Pig Latin. Technical report, 2016. https://hal.inria.fr/hal-01353891.

[4] CDH. http://www.cloudera.com/.

[5] I. Elghandour and A. Aboulnaga. ReStore: reusing results of MapReduce jobs. *PVLDB*, 2012.

[6] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *PVLDB*, 2014.

[7] G. Graefe. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[8] S. Grumbach and T. Milo. Towards Tractable Algebras for Bags. In *PODS*, 1993.

[9] Apache Hadoop. http://hadoop.apache.org/.

[10] HDP. http://www.hortonworks.com/.

[11] M. Jarke. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems*. Springer, 1985.

[12] K. Karanasos, A. Katsifodimos, and I. Manolescu. Delta: Scalable data dissemination under capacity constraints. *PVLDB*, 2013.

[13] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic Physical Design for Big Data Analytics. In *SIGMOD*, 2014.

[14] F. Nagel, P. Boncz, and S. D. Viglas. Recycling in pipelined query evaluation. In *ICDE*, 2013.

[15] T. Neumann and G. Moerkotte. Generating optimal dag-structured query evaluation plans. *Computer Science-Research and Development*, 24(3):103–117, 2009.

[16] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: sharing across multiple queries in MapReduce. *PVLDB*, 2010.

[17] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX*, 2008.

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[19] Apache Pig. http://pig.apache.org/.

[20] http://cwiki.apache.org/confluence/display/PIG/PigMix.

[21] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*, 2000.

[22] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[23] Y. N. Silva, P.-A. Larson, and J. Zhou. Exploiting Common Subexpressions for Cloud Query Processing. In *ICDE*, 2012.

[24] G. Wang and C.-Y. Chan. Multi-Query Optimization in MapReduce Framework. *PVLDB*, 2013.

[25] J. Yang. Algorithms for materialized view design in data warehousing environment. *PVLDB*, 1997.

[26] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.