PigReuse: A Reuse-based Optimizer 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, operates on a particular algebraic representation of Pig Latin scripts. 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. Languages that have gained wide adoption include Pig Latin [21], HiveQL [31], or Jaql [4].

In this work, we consider Pig Latin 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 [22], which is included in all leading Hadoop distributions e.g., HDP [12], CDH [5].

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 Pig Latin script batch, whereas there is obviously an opportunity for enhancing performance by identifying common subexpressions, executing them only once, and reusing the results of the computation 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 algorithms to solve them.

Motivating example. A Pig Latin script consists of a set of binding expressions and store expressions. Each binding expression follows the syntax \texttt{var = op}, meaning that the expression \texttt{op} will be evaluated, and the bag of tuples thus generated will be bound to the variable \texttt{var}. Then, \texttt{var} can be used by follow-up expressions in a script.

Consider the following Pig Latin script \texttt{a1}:

\begin{verbatim}
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. S = FOREACH R GENERATE user, time, zip;
5. T = JOIN A BY user LEFT, B BY name;
6. STORE T INTO 'a1out1';
7. STORE S INTO 'a1out2';
\end{verbatim}

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

The following script \texttt{a2} only executes a left outer join over the same inputs:

\begin{verbatim}
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';
\end{verbatim}
The script $b$ that we introduce next produces the same outputs as $a_1$ and $a_2$:

1. $A = \text{LOAD} \text{ 'page_views'} \ \text{AS} \ (\text{user}, \text{time}, \text{www});$
2. $B = \text{LOAD} \text{ 'users'} \ \text{AS} \ (\text{name}, \text{zip});$
3. $R = \text{COGROUP} \ A \ \text{BY} \ \text{user}, \ B \ \text{BY} \ \text{name};$
4. $T = \text{FOREACH} \ A \ \text{GENERATE} \ \text{flatten}(A), \ \text{flatten}(B);$  
5. $T = \text{FOREACH} \ B \ \text{GENERATE} \ \text{user}, \ \text{time}, \ \text{zip};$
6. $\text{STORE} \ T \ \text{INTO} \ \text{’a1out1'};$
7. $U = \text{FOREACH} \ R \ \text{GENERATE} \ \text{flatten} \ (A),$
8. $\text{STORE} \ U \ \text{INTO} \ \text{’a1out2'};$
9. $\text{STORE} \ U \ \text{INTO} \ \text{’a2out'};$

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 \textsc{cogroup}$^3$ operation (line 3) and \textsc{foreach} operations (lines 4 and 7-8). The interest of \text{cogroup} is that through some simple restructuring, one can carve out of the \text{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 \text{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 produced the desired output of scripts $a_1$ (line 9) and $a_2$ (line 10).

Figure 1 depicts the integration of our reuse-based optimization into the Pig Latin architecture; modules, indicated by dashed lines, belong to the original Pig Latin query processor. As illustrated, our reuse-based optimizer works on the algebraic representation of Pig Latin scripts. Thus, our proposal 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.

\textbf{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 (DAGs) of algebraic representation of a batch of Pig Latin scripts. After identifying such reutilization opportunities, PigReuse produces an optimal merged plan where redundant computations have been eliminated. PigReuse relies on Binary Integer Linear Programming 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.

\textbf{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.

\section{PRELIMINARIES}

\textbf{Pig Latin operations translation.} 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 [10] (NRAB, for short) to represent Pig Latin scripts.

We consider a subset of the NRAB algebra and extend it with other operators. Table 1 lists and describes all basic operators of NRAB (top part) and the additional operators we introduce (bottom part). All additional operators but \text{scan} and \text{store} are redundant, i.e., they can be expressed using the basic operators. We decided to introduce additional operators for two main reasons: (i) allowing a one-to-one representation of Pig Latin scripts into the algebra, as complex operators are efficiently executed by underlying execution engines (e.g., \text{cogroup}), and (ii) giving our algorithm additional opportunities to detect common subexpressions by exploring different rewritings. For instance, any type of \text{join} can (also) be expressed by a combination of \text{cogroup}, \text{restructure}, and \text{bag destroy}. Using this alternative join representation simplifies matching it with any other operators.

\textbf{Pig Latin scripts translation.} We have formalized (and implemented) the entire Pig Latin-to-NRAB translation process; formal details of the translation are presented in Appendix B.

To illustrate, Figure 2.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 \text{page_views}, \text{users}, and \text{power_users}; from now on, we denote these relations as $A$, $B$, and $C$. Consider $s_1$ in Figure 2. Its translation yields the following set of of NRAB binding expressions:

\begin{align*}
\Gamma &= \{ A = \text{scan}(\text{’page_views’}), \\
B &= \text{scan}(\text{’users’}), \\
R &= \forall (\text{user}, \text{name})(A, B), \\
S &= \pi (\text{user}, \text{time}, \text{zip})(R), \\
&\text{store}(\text{’out’}))(S) \}
\end{align*}

In turn, we represent a set $\Gamma$ of NRAB binding expressions obtained from a Pig Latin program as follows:
\[ \begin{array}{|c|c|c|c|} \hline \text{Notation} & \text{Name} & \text{Input arity} & \text{Output description} \\ \hline \epsilon & \text{Duplicate elimination} & \text{Unary} & \text{Distinct tuples from the input relation.} \\ \map(\varphi) & \text{Restructure} & \text{Unary} & \text{All the tuples in the input after applying a function } \varphi. \\ \sigma(p) & \text{Selection} & \text{Unary} & \text{All the tuples in the input that satisfy the boolean predicate } p. \\ \triangledown & \text{Additive union} & n\text{-ary, } n \geq 2 & \text{Union of input relations, including duplicates.} \\ \unlhd & \text{Subtraction} & \text{Binary} & \text{Difference between relations, including duplicates.} \\ \times & \text{Cartesian product} & n\text{-ary, } n \geq 2 & \text{Cartesian product of input relations, including duplicates.} \\ \delta & \text{Bag-destroy function} & \text{Unary} & \text{Unnest one level for the tuples in the input relation.} \\ \hline \end{array} \]

<table>
<thead>
<tr>
<th>Notation</th>
<th>Name</th>
<th>Input arity</th>
<th>Output description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan(fileID)</td>
<td>Load</td>
<td>-</td>
<td>Reads a file and loads it as a relation.</td>
</tr>
<tr>
<td>store(dir)</td>
<td>Store</td>
<td>Unary</td>
<td>Writes the contents of the tuples for an input relation to a file.</td>
</tr>
<tr>
<td>π(a₁,...,aₙ)</td>
<td>Projection</td>
<td>n-ary,</td>
<td>Groups tuples together from input relations based on the equality of their values for attributes (a₁,...,aₙ).</td>
</tr>
<tr>
<td>⋈(a₁,...,aₙ)</td>
<td>Join</td>
<td>n-ary, n ≥ 2</td>
<td>Returns the combination of tuples from input relations based on the equality of their values for attributes (a₁,...,aₙ).</td>
</tr>
<tr>
<td>⋈(a₁,a₂)</td>
<td>Left outer join</td>
<td>Binary</td>
<td>Returns the combination of tuples from input relations for which a₁=a₂, and the tuples in the left relation without a matching right tuple.</td>
</tr>
<tr>
<td>⋈(a₁,a₂)</td>
<td>Right outer join</td>
<td>Binary</td>
<td>Returns the combination of tuples from input relations for which a₁=a₂, and the tuples in the right relation without a matching left tuple.</td>
</tr>
<tr>
<td>⋈(a₁,a₂)</td>
<td>Full outer join</td>
<td>Binary</td>
<td>Returns the combination of tuples from input relations for which a₁=a₂, the tuples in the left relation without a matching right tuple, and the tuples in the right relation without a matching left tuple.</td>
</tr>
<tr>
<td>mapconcat(φ)</td>
<td>Restructure and concatenate</td>
<td>Unary</td>
<td>Applies map(φ) and concatenates its result to the original tuple.</td>
</tr>
<tr>
<td>empty</td>
<td></td>
<td></td>
<td>Returns true if and only if the input relation is empty.</td>
</tr>
<tr>
<td>sum, max, min, count</td>
<td>Aggregate functions</td>
<td>Unary</td>
<td>Returns the sum of integer values for an attribute field in an input relation, maximum integer value, minimum integer value of an attribute field in an input relation, and total number of tuples in an input relation.</td>
</tr>
</tbody>
</table>

Table 1: Basic NRAB operators (top) and proposed extension (bottom) to express Pig Latin semantics.

\[ \begin{array}{|c|c|c|c|c|} \hline S1 & A = LOAD \text{‘page_views’} \ AS \text{ ‘user, time, www’}; & B = LOAD \text{ ‘users’} \ AS \text{ ‘name, zip’}; & R = JOIN A BY user, B BY name; & S = FOREACH R GENERATE user, time, zip; \text{ STORE S INTO ‘s1out’}; \\ S2 & A = LOAD \text{‘page_views’} \ AS \text{‘user, time, www’}; & B = LOAD \text{‘users’} \ AS \text{‘name, zip’}; & C = LOAD \text{‘power_users’} \ AS \text{‘id, phone’}; & R = JOIN A BY user, B BY name; & S = FOREACH R GENERATE user, time, id, phone; \text{ STORE T INTO ‘s2out’}; \\ S3 & A = LOAD \text{‘page_views’} \ AS \text{‘user, time, www’}; & B = LOAD \text{‘users’} \ AS \text{‘name, zip’}; & R = FOREACH A GENERATE user, time; & S = JOIN R BY user LEFT, B BY name; \text{ STORE S INTO ‘s3out’}; \\ S4 & A = LOAD \text{‘page_views’} \ AS \text{‘user, time, www’}; & B = LOAD \text{‘users’} \ AS \text{‘name, zip’}; & C = LOAD \text{‘power_users’} \ AS \text{‘id, phone’}; & R = JOIN A BY user, B BY name, C BY id; \text{ STORE S INTO ‘s4out’}; \hline \end{array} \]

Figure 2: Sample Pig Latin scripts (a) and their corresponding algebraic DAG representation (b).

**Definition.** The DAG representation of a set of bindings \( T = \{ \text{var}_1=A_1, \ldots, \text{var}_n=A_n \} \) is a pair \((V,E)\) where \( V \) is a set of \((\text{var}_i, \text{op}_j^A)\) tuples such that for each \( v_i \in V \):

- \( \text{var}_i \) is the variable associated to the node (thus, it is the unique identifier of a node);
- \( \text{op}_j^A \) is the top-most algebraic operator in the expression bound to \( \text{var}_i \).

Further, \( E \) is a set of edges representing the data flow among the nodes of \( V \). Specifically, there is an edge \( e_{i,j} \in E \) from \( v_i \) to \( v_j \), iff the operation \( \text{op}_j^A \) is applied on the bag of tuples produced by \( \text{op}_i^A \).

In our DAG representation, a source i.e., a node with no incoming edges, always contains a \text{scan} operator. In turn, a sink i.e., a node with no outgoing edges, always corresponds to a \text{store} operator. For instance, after connecting the different algebraic expressions generated from \( s_1 \), we obtain the DAG query \( q_1 \) shown in Figure 2.b, also including the DAG-based representations of \( s_2-s_4 \).

3. **REUSE-BASED QUERY OPTIMIZATION.**

We have previously shown how to translate Pig Latin scripts into NRAB DAGs. Based on this, 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:

**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 actually evaluated. Our approach is independent of the particular cost function chosen; we discuss in Section 5.2 the functions that we have implemented for PigReuse.

The final output of PigReuse is an optimized plan that contains (i) the operator nodes leading to minimizing the cumulated 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 [8, 27]. In the EG, an AND-node corresponds to an algebraic operation (e.g., selection, projection etc.). An OR-node is introduced whenever a set of expressions \( e_1, e_2, \ldots, e_k \) have been identified as equivalent; in the EG, \( o \) has as children 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 2.** An equivalence graph (EG) is a DAG, defined by the pair \((O \cup A \cup T_0, E)\), with \( O, A, \) and \( T_0 \) disjoint sets of nodes, and:

- \( O \) is the set of equivalence nodes, \( A \) is the set of operator nodes, \( T_0 \) is the set of sink nodes.
- \( E \subseteq (O \times A) \cup (A \times O) \cup (A \times T_0) \) 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 \( t_o \in T_0 \) 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 \( T_0 \) nodes.

An important point to stress here is that equivalence nodes with more than one child amount to optimization opportunities as they indicate that several operator nodes have a common (equivalent) child subexpression. In this case, we can choose the “best” way to compute the result of the subexpression among the choices given by the OR-node. The choice is based on a cost model, where the best plan corresponds to the plan with overall minimal cost. Optimal plan selection is discussed in detail in the next section.

**Building the equivalence graph.** To build the equivalence graph, we need to identify equivalent expressions within the input NRAB query set \( Q \). We reuse the classical notion of query equivalence here; i.e., two expressions are equivalent iff their result is provably the same regardless of the data on which they are computed.

We build the EG in the following fashion. First, we create the EG \( eg \) with a single equivalence node \( o_s \), i.e., the EG source. We take every NRAB query \( q \in Q \) and perform a breadth-first traversal of its nodes. Each source node \( s \in q \) is added to \( eg \), and an edge \((o_s, s)\) is created.

Figure 3: EG corresponding to NRAB DAGs \( q_1 \cdots q_4 \).

Subsequently, for each node \( n \) having the source node \( s \) as an input, we verify whether there exists a node \( n_{eq} \) in \( eg \), such that the expression rooted in \( n \) is equivalent to the one rooted in \( n_{eq} \).

If such an equivalence is detected, we connect \( n \) to the equivalence node \( o \) that \( n_{eq} \) feeds. If no such equivalent node is found, \( n \) is added to \( eg \), a new equivalence node \( o \) is added to \( eg \), and an edge \((n, o)\) is created. In either case, for each node \( n' \) that is a parent of \( n \) in the original query, \( n' \) is added to \( eg \) and an edge \((o, n')\) is created. Within a set of equivalent nodes, each node is the root of a sub-DAG that represents a NRAB expression; the expressions corresponding to all these nodes are equivalent.

In the spirit of [18], we rely on data structures representing logical properties to check whether two expressions \( A, A' \) rooted at nodes \( n \) and \( n' \) are equivalent. The logical properties are set properly by the known commutativity, associativity etc. laws that have been extensively studied for the bag relational algebra [3, 9, 24, 25]. If \( A \) and \( A' \) are equivalent, they become children of the same equivalence node. Our equivalence search algorithm is sound but not complete; details about the equivalences we are capable of detecting can be found in Appendix D. Recall that the problem of checking equivalence of two arbitrary Relational Algebra expressions is undecidable [30], and so the problem is for NRAB. Thus, no terminating equivalence checking algorithm exists for NRAB. However, as our experimental evaluation shows, the equivalences detected by PigReuse allow it to bring significant performance savings.

As mentioned above, the equivalence detection rules we apply are those previously identified for NRAB, i.e., they only cover operators that have been previously defined as (extensions of) NRAB operators (i.e., \( \psi, \neg, \times, \epsilon, \delta, map, \sigma, \pi \), and \( \delta \); see Table 1). As we will discuss in Section 4, we provide a set of new equivalence rules involving operators we introduced in this work (e.g., cogroup and outer join variants). These rules allow identifying more equivalences in an efficient way, and thus improve over the baseline PigReuse algorithm presented in this section.

Figure 3 depicts the EG corresponding to the NRAB DAGs \( q_1 \cdots q_4 \) in Figure 2.b. In Figure 3, we use boxes to represent equivalence nodes, while sink nodes are represented by shadowed triangles. 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 \( \text{user} \) and \( \text{name} \) are also inputs to the same equivalence node.

### 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, on any input instance.

We call the output plan a result equivalence graph (or REG, in short).

**Definition 3.** 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 operator node in-degree in the REG is equal to its in-degree in the EG. 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 4 shows a possible REG produced for the EG depicted in Figure 3. This REG could have been for instance obtained by using a cost function based on counting the operator nodes in the optimized script. In the REG, each equivalence node has exactly one input edge, i.e., the scans and other operator nodes are shared across queries, whenever possible. In Section 5, we consider different cost functions and compare them experimentally.

Minimize \( C = \sum_{e \in E} C_e x_e \) subject to:

\[
x_e \in \{0, 1\} \quad \forall e \in E \tag{1}
\]

\[
\sum_{e \in E_{in}^o} x_e = 1 \quad \forall t_o \in T_o \tag{2}
\]

\[
\sum_{e \in E_{in}^o} x_e = x_{E_{out}^o} \times |E_{in}^o| \quad \forall a \in A \tag{3}
\]

\[
\sum_{e \in E_{out}^o} x_e = \max_{e \in E_{out}^o} x_e \quad \forall o \in O \tag{4}
\]

Figure 5: BIP reduction of the optimization problem.

\[
d_{e \in E_{out}^o} \in \{0, 1\} \quad \forall o \in O \tag{4.1}
\]

\[
\sum_{e \in E_{out}^o} x_e \geq x_{E_{out}^o} \quad \forall o \in O \tag{4.2}
\]

\[
\sum_{e \in E_{out}^o} x_e \leq (x_e - d_e + 1)_{e \in E_{out}^o} \quad \forall o \in O \tag{4.3}
\]

\[
\sum_{e \in E_{out}^o} d_e = 1 \quad \forall o \in O \tag{4.4}
\]

Figure 6: BIP representation of the \( \max \) constraint.

### 3.3 Cost minimization based on binary integer programming

We model the problem of finding the minimum-cost REG relying on Binary Integer Programming (BIP), a well-explored branch of mathematical optimizations that has been used previously to solve many optimization problems in the database literature [15, 33]. Broadly speaking, a typical 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.

Such problems can be tackled by dedicated binary integer program solvers, some of which are extremely efficient, benefiting from many years of research and development.

**Generating the result equivalence graph.** Given an input EG, for each of its nodes \( n \in O \cup A \cup T_o \), we denote by \( E_{in}^o \) and \( E_{out}^o \) 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 a variable \( x_e \) can only take values within \( \{0, 1\} \), our problem is formulated as a BIP problem. Further, for each edge \( e \in E \), we denote by \( C_e \) the cost assigned to \( e \) by some cost function \( C \). Importantly, the model we present in the following is independent of the chosen cost function.

Our optimization problem is stated in BIP terms in Figure 5. 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.

Since \( \max \) is not directly supported in the BIP model, the actual BIP constraints which we use to express (4) are shown in Figure 6. These constraints encode the \( \max \) constraint as follows. Equation (4.1) introduces a binary variable \( d_{e \in E_{out}^o} \) used to model the \( \max \) function. Equation (4.2) states that if an outgoing edge of an equivalence node is selected, then
4. EFFECTIVE REUSE-BASED OPTIMIZATION

We present a set of techniques for identifying and exploiting additional subexpression factorization opportunities that go beyond those possible with the standard NRAB operators. The three extensions we bring to the basic PigReuse algorithm are: normalization, join decomposition, and aggressive merge.

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. 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.

Figure 7 spells out the conditions under which a π can be swapped with its parent operator. Each column in the toprmost row represents a parent operator with which the child π may be swapped, and the value of each cell represents different conditions under which the swap is possible. For example, a child π can be swapped with a parent σ, iff the selection predicate does not carry over the attributes projected in π.

A special case is the cogroup operator. Since cogroup nests the input relations, reordering π with this operator requires complex rewriting. In particular, we will rewrite it into a map that applies the projection π on the bag of tuples corresponding to the input relation. map operators containing only combinations of map and π can still be pushed up following the conditions in Figure 7. This means that, in general, during normalization, one may need to introduce map operators nested more than two levels deep. Although the Pig Latin query language does not allow more than two levels of nesting, our NRAB representation allows it; furthermore, as we have found examining the code for executable plans within the Pig Latin engine, more

Figure 7: Reordering and rewriting rules for π.

one of its incoming edges is selected too. (4.3) states that if no outgoing edge of an equivalence node is selected, then none of its incoming edges is selected. Further, (4.3) and (4.4) together ensure that if an outgoing edge of an equivalence node is selected, only one of its incoming edges will be selected. Observe that we can model the max function in this fashion since in equation (4), max is computed over a set of inputs whose values are in [0, 1].

Observe that operators such as ε or ω restrict the possibilities of moving π operators across the DAG. It turns out also that they do not commute with the other algebraic operators; we term these “unmovable” operators, bordering operators in the sense that they raise borders to the moving of π across the DAG.

After our reuse-based algorithm produces the optimized REG, to avoid the performance loss incurred by manipulating many attributes at all levels (due to the pulling up of the projections), we push the π operators back, as close to the scan as possible. As our normalization algorithm may rewrite π operators using map, we extended the Pig Latin optimizer to support the (unnesting) rewriting of such cases, so that the π can be pushed back down through the plan. Recall that even if they cannot be pushed back down, the resulting plan (no matter how many levels the π operators are nested) will be executable by the Pig engine.

To illustrate the advantages of our normalization phase, Figure 8 shows the EG generated by PigReuse on the normalized NRAB DAGs q1→q4. Comparing this EG with the one shown in Figure 3, we see that due to the swapping of the π operator corresponding to q2, our algorithm can identify an additional common subexpression between q2 and q4, by determining the equivalence between the joins over A, B, and C; the corresponding equivalence node is highlighted in Figure 8.

Join decomposition. The semantics of Pig Latin’s JOIN operators e.g., ⋈, ⋈, ⋈, or ⋈ allow rewriting (or decomposing) 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 joins results, can be shared across different kinds of joins. The map will be different in each case.

Join decomposition. The semantics of Pig Latin’s JOIN operators e.g., ⋈, ⋈, ⋈, or ⋈ allow rewriting (or decomposing) 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 joins results, can be shared across different kinds of joins. The map will be different in each case.

The class pig.newplan.logical.relational.LOForEach, representing the FOREACH operator, has a field called innerPlan which in our tests could contain another LOForEach and so on on several levels. The purpose of the language-level restriction may have been to prevent programmers from writing deeply-nested loops whose performance could be poor.
depending on the join type, but the most expensive component of computing the join, namely the \( \text{cogroup} \), will be factorized. Further, there is no noticeable performance difference between executing a certain join or its decomposed rewritten version, as the overhead introduced by the \( \text{map} \) operators is negligible.

Figure 9 shows the decomposition rules that are applied on the input NRAB DAGs. Rule (LI) rewrites an inner \( \pi \) into two operators. The first one is a \( \text{cogroup} \) on the attributes used by the join predicate. The second one is a \( \text{map} \) that does the following for each input tuple: (i) project each bag of tuples corresponding to the \( \text{cogroup} \) input relations; (ii) apply a \( \delta \) operation on each of those bags; and (iii) perform a cartesian product among the tuples resulting from unnesting those bags. Observe that if a bag is empty, e.g., the input relation did not contain any value for the given grouping value, the \( \delta \) operator does not produce any tuple, and thus the tuples from the other bags for the given tuple are discarded. Thus, this rewriting produces the exact same result as the original \( \Join \) operator.

The rest of the rules use the aggregation function \( \text{empty} \), that checks if an input bag is empty. For instance, the expression \( \text{empty}(\var)(\langle \bot \rangle) : \var \) is a conditional assignment, that is: if \( \var \) is empty, a bag with a null tuple (\( \bot \)), i.e., a tuple whose values are bound to null values, conforming to \( \var \) schema is assigned, otherwise the bag \( \var \) is assigned.

Rule (LOJ) rewrites a left outer join \( \Join^L \) into a \( \text{cogroup} \) on the attributes used by the join predicate, followed by a \( \text{map} \) operator that (i) unnests the bag associated to the left input of the \( \text{cogroup} \); (ii) if the bag associated to the right input \( \var \) 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 \( \delta \) operations in order to generate the \( \Join^L \) result. Rule (ROJ) rewrites a right outer join \( \Join^R \) in a similar fashion.

Finally, rule (FOJ) rewrites a full outer join \( \Join^F \) following the same principle as for the two previous operators. The difference is that in (FOJ) we check the bags from both inputs by means of the \( \text{empty} \) function.

Figure 10 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 \( \Join \) operators from \( q_1 \) and \( q_2 \), and the \( \Join^F \) operator from \( q_3 \) leads to an additional sharing opportunity, as the result of the \( \text{cogroup} \) on attributes \( \text{user} \) and \( \text{name} \) can be shared by the subsequent \( \text{map} \) operations (highlighted equivalence node).

**Aggressive merge.** This optimization is based on the observation that it is possible to derive the results of a \( \Join \) or \( \text{cogroup} \) operator from the results of a \( \text{cogroup}' \) operator, as long as the former relies on a subset of the input relations and attributes of \( \text{cogroup}' \). This means that these rewritings rely on the notion of \( \text{cogroup} \) 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 \( \Join \) or \( \text{cogroup} \) operator, we add the appropriate operator on top of \( \text{cogroup}' \); this can be seen as a limited instance of query rewriting using views, where \( \text{cogroup}' \) plays the role of a view. In contrast to the previous extensions that are applied on the input NRAB DAGs, aggressive merge is applied while creating the EG.

Figure 11 shows the rewritings considered by our aggressive merge algorithm. Rule (CG-CG) states that if a query contains a \( \text{cogroup}' \) operator with two or more input relations, any other \( \text{cogroup} \) (with at least one input relation, part of the \( \text{cogroup}' \) input) can be derived from the previous one in the following fashion. First, a \( \pi \) operator projects the attributes needed for the result of the \( \text{cogroup} \) operator. Then, a \( \sigma \) operator discards the tuples where all the bags associated to each input relation are empty.

Rules (LJ-CG), (LOJ-CG), and (ROJ-CG) are similar to those shown in Figure 9; the only difference is that the \( \text{map} \) operators take only a subset of the bag attributes in the original \( \text{cogroup} \). Note that we do not have a rule for the \( \Join^R \) operator since we are able to generate its output directly from the result of the \( \text{cogroup} \).

Figure 12 depicts the EG produced by PigReuse using the aggressive merge extensions, when normalization and decomposition has been applied to the NRAB plans \( q_1 \) to \( q_4 \). The new connections created by aggressive merge are highlighted. The figure shows how the results for the \( \text{cogroup} \), \( \Join \), and \( \Join^F \) operators on \( A \) and \( B \) relations are derived from the \( \text{cogroup} \) operator on \( A \), \( B \), and \( C \).
Figure 11: Rules for aggressive merge.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure11.png}
\caption{EG generated by PigReuse applying aggressive merge on the normalized and decomposed NRAB DAGs $q_1$-$q_4$.}
\end{figure}

5. EXPERIMENTAL EVALUATION

We have implemented PigReuse, our reuse-based optimization approach, 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 [22], which relied on the Hadoop platform 1.1.2 [11]. The cost-based plan selection algorithm (Section 3.2) uses the Gurobi BIP solver 5.6.2 (www.gurobi.com).

Section 5.1 describes our experimental setup. Then, Section 5.2 presents the two alternative cost functions that we implemented and experimented with. Finally, Section 5.3 presents our experimental results.

5.1 Experimental setup

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. The nodes run Linux CentOS 6.4. Each node has two 600GB SATA hard disks where HDFS is mounted.

Setup. For validation, we used data sets and scripts provided by the PigMix [23] 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. The first one (denoted $W_1$) comprises 12 scripts taken directly from the PigMix benchmark, namely $l_2$-$l_7$ and $l_{11}$-$l_{16}$; these only use operators supported by our current implementation, e.g., JOIN, COGROUP, FILTER etc. Each script has on average 7 operators. The second workload ($W_2$) includes $W_1$, to which we add 8 extra scripts which feature many JOIN flavours, COGROUP on many relations etc. These scripts are added created to give opportunities to validate our algorithm on a wider variety of operators. Further details about these workloads can be found in Appendix E.

5.2 Cost functions and experiment metrics

We now present the two cost functions that are implemented currently in PigReuse, focusing on the number of logical operators, and the number of MapReduce jobs, respectively. Although more elaborated cost functions can be envisioned [13], these two already lead to considerable gains due to reuse, as our experiments shortly show.

Operator-based cost function A first cost function characterizing the effort required by the evaluation of a batch of Pig Latin scripts is the number of operators in the equivalent NRAB expression eventually evaluated, that is:

$$C_e = 1 \quad \forall e \in E^\text{out}_{\alpha}, \forall \alpha \in A \quad C_e = 0 \quad \text{for all the rest}$$

Above, we assign a cost of 1 to the execution of every algebraic operator $\alpha$, and we attach this cost to its outgoing edge. All the other edges, i.e., incoming edges to an operator node, have a cost of 0.

MapReduce jobs-based cost function Our second cost function is closely related to the Pig execution engine on top of MapReduce. The function minimizes the MapReduce jobs needed to compute the results of the input Pig Latin scripts, as some groups of operators are executed by Pig as part of the same job. For instance, $\sigma$, $\pi$, and map do not generate
a new MapReduce job, which is very convenient for our decomposition and aggressive merge extension techniques that introduce these operators quite aggressively when rewriting.

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.

5.3 Experimental results

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

Figure 13 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.

In Figure 13.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 13.b) increases for the multi-client alternative. This is because the number of slots needed for map tasks is very large, so the scheduler cannot overlap significantly the map phases of multiple queries. Thus, their execution remains quite sequential.

For the workloads we considered, our extensions reduced the total work over the baseline PigReuse algorithm (Figure 13.b). However, this was not always the case for the execution time (Figure 13.a). The reason is that some of the requiring more effort, had less execution 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, if the minmr cost function is used, PigReuse chooses an alternative plan that executes faster even though it has more operators.

Table 2 provides some important metrics concerning the EGs and REGs created by PigReuse algorithm. PigReuse reduces the total number of logical operators by an average of 30%, using any of the two cost functions. The REGs generated by PigReuse using the minop or minmr cost functions have the same number of operators, except when aggressive merge is used (PR+NDA). The reason is that all the connections that we establish through the aggressive merge strategy do not result in extra MapReduce jobs. Thus, using that strategy and the minmr cost function, a plan that contains more nodes but translates into less MapReduce jobs is selected. As we have seen before, this alternative plan leads to considerable execution time savings.

Concerning the total compile time overhead of using PigReuse (i.e. the time needed to generate the optimal set of scripts starting from the input workload), it stays below 125ms in all considered cases. So compile time is negligible compared to running time of the workloads, ranging from 28 minutes to 2 hours 35 minutes. Fast compile time is ensured by the adoption of a fast BIP solver, and by the fact that techniques we have devised to detect common sub-expressions admit fast implementation.

6. RELATED WORK

Relational multi-query optimization. Our work directly relates to multi-query optimization (MQO), seeking to improve the performance of query batches with common sub-expressions. The early works [14, 28] proposed exhaustive, expensive algorithms which were not integrated with existing system optimizers. The technique presented in [27] was the first to integrate MQO into a Volcano-style optimizer, while [34] presents a completely integrated MQO solution accounting also for the usage and maintenance of materialized views. The approach of [29] takes into account the physical requirements (e.g., data partitioning) 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 optimises workloads expressed in terms of the richer NRAB algebra. As seen in Section 4, the presence of nested expressions involving operators like cogroup and map, introduces issues that are typical of NRAB. Also, differently from [27, 29] our approach does not need fundamental modifications to the query optimiser ([27]) nor it needs to rely on assumptions at the physical level of the query engine [29]. As a consequence our approach has the advantage to be able to be directly applied to alternative implementations of Pig Latin (or of any other language based on NRAB).

The above considerations still hold for the Shared Workload optimization (SWO) approach proposed in [7], which relies on sharing physical operators (such as scan, build, probe, and several flavours of join) in large SQL workloads. Unlike PigReuse, this work both identifies sharing opportunities and optimizes the queries in order to improve global performance. To this end it deeply depends on the cost model of the query engine and on statistics about data (in order to estimate selectivity in join operations). As pointed out in [7], the global optimization + 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 (although this does not apply other optimizations such as join reordering etc.). The approach presented in [18] 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. Addressing simultaneously the optimization and reordering problem for Pig Latin is an interesting area of future work for which we laid foundations by formalizing the translation from Pig Latin, a language strictly more expressive than the SQL considered in [7, 18], to NRAB. Another interesting note is: while, unlike [7, 18], we do not explore operator reordering (other than \(\sigma\) and \(\pi\) and strictly for the needs of factorization), we shared with these works the need for quickly determining which operators are not likely to be equivalent; along the lines of [18], we used a set of interesting operator properties, e.g., the relations they join and the predicates they apply, to quickly prune out comparisons when looking for sharing opportunities.

Recycling techniques for a pipelined query engine are presented in [17], which represents dynamic SQL workloads as AND-DAGs. PigReuse DAGs are more complex as they include OR nodes, and our rewriting rules are more sophisticated.

**Re-use-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, 19, 32], or need some annotations to the jobs to inform about sharing opportunities [6, 16]. 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, 20]. 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. Pig Latin optimization is discussed in [20]. Partial result sharing is considered especially from a scheduling perspective, that is: how to schedule programs in order to best profit from the shared computations. In our work, we assume the complete workload is examined and optimized, and then turned to the Pig Latin engine which schedules it independently for execution. The core of our work thus is concerned with identifying common sub-expression and examining the global sharing problem, which are not addressed in [20]. Approaches presented in [2, 20] are complementary to PigReuse and could be combined with it for better performance.

7. **CONCLUSIONS AND FUTURE WORK**

We have presented a novel approach for identifying and reusing common subexpressions occurring in 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. Our algorithm allows plugging any cost function and its output is a merged script reducing its value. Our experimental results demonstrate the value of our reuse-based algorithms and optimization strategies.

We see several interesting extensions to this work. First, adding better support for the optimization of Pig Latin scripts that contain calls to user-defined functions (UDFs). Our preliminary investigation (see Appendix C for details) revealed that the extension is feasible, and we postpone to future work its implementation. Second, we would like to add more (complex) cost functions in order to identify the ones leading to the most interesting total work reductions.

8. **REFERENCES**


cwiki.apache.org/confluence/display/PIG/PigMix.


APPENDIX

A. EXTENDED NESTED RELATIONAL ALGEBRA FOR BAGS

First, we recall the NRAB [10] data model in Section A.1, while we present the subset of its operators that we use to represent Pig Latin semantics in Section A.2. Then, Section A.3 extends NRAB with the Pig Latin operators, whose semantics are defined using the subset of NRAB operators that we introduce previously.

A.1 Data model

Let us assume the existence of a set of domain names $D_1, \ldots, D_n$ and an infinitive set of attributes $a_1, a_2, \ldots$. Further, the domain names are associated with domains $D_1, \ldots, D_n$. The elements of the domains can be of either atomic type or complex type. A type is associated with each instance of a domain. Formally, types and values are defined as follows:

- If $\hat{D}_i \in \hat{D}$ is a domain name, then $\hat{D}_i$ denotes the domain type. For each database relation $R$ in domain $\hat{D}_i$, the type of $R$ is $\hat{D}_i$.
- If $T_1, \ldots, T_n$ are types and $a_1, \ldots, a_n$ are distinct attribute names for tuples in a database relation $R$, then $R = \{ [a_1 : T_1, \ldots, a_n : T_n] \}$ is a bag of tuples in which $[a_1 : T_1, \ldots, a_n : T_n]$ is a tuple type. If $v_1, \ldots, v_n$ are values of types $T_1, \ldots, T_n$, respectively, then $[a_1 : v_1, \ldots, a_n : v_n]$ is value of the tuple type. We also include $T_i$ as a type; the only value of this type is [], the empty tuple.
- A bag is a (homogeneous) collection of tuples that may contain duplicates. If $T$ is a tuple type, then $\{ [T] \}$ is a bag type, whose domain is a set of bags containing homogeneous tuples of type $T$. We say that an element $\alpha$ $n$-belongs to a bag, if element $\alpha$ has $n$ occurrences in that bag.
- A bag database is a set of named bags. A bag schema is an expression $B : T$, where $B$ is a bag name and $T$ is a bag type. An instance of $B$ is a bag of type $T$.

A.2 Basic operators

NRAB operators. We now describe the NRAB operators [10] that we use to express Pig Latin semantics. The input and output types of all these operators are bag type.

- Duplicate elimination ($\epsilon$). This operator extracts the distinct tuples in a relation. $\epsilon(R)$ is a bag containing exactly one occurrence of each tuple in $R$ i.e., an element $\alpha$ $1$-belongs to $\epsilon(R)$ if $\alpha$ $p$-belongs to $R$ for some $p > 0$, and $0$-belongs to $\epsilon(R)$ otherwise.
- Restructuring (map). map($\varphi$)($R$) returns a bag of type $\{ [T] \}$, constructed by applying a function $\varphi$ on each element of $R$. This operation is introduced for performing restructuring of complex values, which may include the application of functions to substructures of the values. map is a higher order operation with a function parameter $\varphi$ that describes the restructuring.
- Selection ($\sigma$). Given a bag $R$ and a boolean valued predicate condition $p$, $\sigma(p)$($R$) denotes the select operation that returns a bag containing all the elements of $R$ that satisfy the condition $p$. Only unary predicates...
can be used as parameters for the select; we refer to them as select specifications.

- Additive union (\(\omega\)). This operator deals with the union of bags with possibly duplicate elements. If \(R\) and \(S\) are two input relations of bag type \(\{\{T\}\}\), then \(R \cup S\) is a bag of type \(\{\{T\}\}\), such that a tuple \(t\) of type \(T\) \(n\)-belongs to \(R \cup S\), iff \(t\) \(p\)-belongs to \(R\) and \(q\)-belongs to \(S\) and \(n = p + q\).

- Substraction (\(-\)). If \(R\) and \(S\) are two input relations of bag type \(\{\{T\}\}\), then \(R - S\) is a bag of type \(\{\{T\}\}\), such that a tuple \(t\) of type \(T\) \(n\)-belongs to \(R - S\), iff \(t\) \(p\)-belongs to \(R\), \(q\)-belongs to \(S\) and \(n = \text{max}(0, p - q)\).

- Cartesian product (\(\times\)). If \(R\) and \(S\) are bags containing tuples of arity \(k\) and \(k'\) respectively, then \(R \times S\) is a bag containing tuples of arity \(k + k'\), such that the new relation \(X\) becomes, \(X = R \times S = \{\{a_1, a_2, a_3, \ldots, a_{k+k'}\}\}, \) where \(A_1, A_2, A_3, \ldots, A_{k+k'}\) is a tuple type. Tuple \(t = [a_1, a_2, a_3, \ldots, a_{k+k'}]\) \(n\)-belongs to \(R \times S\) if \(t_1 = [a_1, a_2, a_3, \ldots, a_k]\) \(p\)-belongs to \(R\) and \(t_2 = [a_{k+1}, \ldots, a_{k+k'}]\) \(q\)-belongs to \(S\) and \(n = pq\).

- Bag-destroy function (\(\delta\)). \(\delta\) unests one level of bag nesting. If \(R\) is a bag of type \(\{\{S : \{\{T\}\}\}\}\), then \(\text{map}(\delta(S))(R)\) results a bag of type \(\{\{T\}\}\).

NRAB functions. Function definition in NRAB has two parts: a class of base functions and function constructors that are used for constructing more complex function expressions.

First, we describe the base functions. In our algebra, constants \(c\), and database relation names \(R\) are considered as functions. Additionally, each attribute of the input relation is also considered as a function expression. We use \(\text{id}\) for denoting the identity function. For example, \(\text{map}(R \cup \text{id})(S)\), denotes that additive union of \(R\)'s element is performed recursively on each of \(S\)'s elements, where \(S\) is a bag of tuples. Here, \(\text{id}\) indicates each element in \(S\). The algebraic operations, except select and restructuring, are function expressions. Select and restructuring are function constructors, which are discussed next.

In our algebra, complex functions are constructed by using one of the function construction operators (select and restructuring). If \(\varphi\) is a unary function, then \(\text{map}(\varphi)(R)\) is a function. Similarly, if \(p\) is a unary boolean-valued function then \(\text{sigma}(p)(R)\) is also a function. We use tuple construction as a function constructor i.e., if \(f_1, \ldots, f_n\) are unary functions, then \(f_1 \ldots f_n\) is a unary function, whose meaning is defined by \(f_1 \ldots f_n(x) = f_1(f_2(\ldots f_n(x)\ldots))\). Our algebra supports labeled tuple construction as a function constructor too, i.e., formation of expressions like \(\text{id} = f_1 \ldots f_n\) is allowed; note that the \(\text{id}\) here are not functions but labels. The semantics is given by \(\text{id} = f_1 \ldots f_n\) if \(f_1 \ldots f_n(x) = [a_1 : f_1(x), \ldots, a_n : f_n(x)]\). This implies that every function is unary, unless its input is a tuple.

A.3 Additional operators and functions

In the following, we extend the basic NRAB set of operators to encapsulate the semantics of more complex operations that are supported by the Pig Latin language.

- Scan (\(\text{scan}\)). \(\text{scan}(\text{fileID})\) is an operator introduced to represent a data source that reads a file \(\text{fileID}\).

- Store (\(\text{store}\)). \(\text{store}(\text{dir})(R)\) is an operator introduced to represent a data sink that writes the bag \(R\) to directory \(\text{dir}\).

- Projection (\(\pi\)). \(\pi(a_1, \ldots, a_n)(R)\) projects attributes with names \(a_1, \ldots, a_n\) from the tuples in bag \(R\). Formally:

  \[
  \pi(a_1, \ldots, a_n)(R) \equiv \text{map}([a_1, \ldots, a_n])(R)
  \]

- Cogroup (\(\text{cogroup}\)). In order to define the semantics of the \(\text{cogroup}\) operator, we first define a \(G\) operator that works on a single bag. In particular, \(G(a)(R)\) groups the tuples in \(R\) by the value bound to \(a\). The result of the expression is a bag with tuples containing two elements: a \(\text{group} \) attribute associated to the grouping value, and a \(\text{R} \) attribute associated to the bag of tuples whose attribute \(a\) was bound to that value. Formally:

  \[
  G(a)(R) \equiv \text{map}((\text{map}(\text{sigma}(\text{group}=a)(\text{id}))(R))
  \]

  \[
  (\text{map}(\text{group} = a, R \equiv R))(R)
  \]

  \(\text{cogroup}(a_1, \ldots, a_n)(R_1, \ldots, R_n)\) groups together tuples from multiple bags \(R_1, \ldots, R_n\), based on the values of their attributes \(a_1, \ldots, a_n\), respectively. The result of a \(\text{cogroup}\) operation is a bag containing a \(\text{group} \) attribute, bound to values of attributes \(a_1, \ldots, a_n\), followed by one bag of grouped tuples for each relation in \(R_1, \ldots, R_n\). Without loss of generality, we define it formally for two input relations; the extension for more than two inputs is straightforward. Thus:

  \[
  \text{cogroup}(a_1, a_2)(R_1, R_2) \equiv A_9
  \]

  where:

  \[
  A_1 := G(a_1)(R_1) \quad A_2 := G(a_2)(R_2) \quad A_3 := (\text{group}=a_1)(A_1, A_2) \quad A_4 := \pi(\text{group})(A_3)
  \]

  \[
  A_5 := \pi(\text{group})(A_1) \quad A_6 := (\text{group}=a_2)(A_4, A_1) \quad A_7 := \pi(\text{group})(A_2)
  \]

  \[
  A_8 := (\text{group}=a_2)(A_7 - A_4, A_2) \quad A_9 := A_3 \uplus A_6 \uplus A_8
  \]

- Inner join (\(\bowtie\)). \(\bowtie(a_1, a_2, \ldots, a_n)(R_1, R_2, \ldots, R_n)\) creates the cartesian product between the tuples in bags \(R_1, R_2, \ldots, R_n\), and filters the resulting tuples based on condition \(a_1 = a_2 = \ldots = a_n\). Thus, \(\bowtie\) is formalized as:

  \[
  \bowtie(a_1, a_2, \ldots, a_n)(R_1, R_2, \ldots, R_n) \equiv \sigma(a_1 = a_2 = \ldots = a_n)(R_1 \times R_2 \times \ldots \times R_n)
  \]

- Left outer join (\(\bowkt\)). \(\bowkt(a_1=a_2)(R_1, R_2)\) returns the cartesian product of tuples from input relations \(R_1\) and \(R_2\) for which boolean condition \(a_1=a_2\) is true, and the tuples in \(R_1\) without a matching right tuple. Formally:

  \[
  \bowkt(a_1, a_2)(R_1, R_2) \equiv A_5
  \]

  where:
\[ A_1 := \times (a_1, a_2)(R_1, R_2) \quad A_2 := \pi(a_1)(A_1) \]
\[ A_3 := \pi(a_1)(R_1) \quad A_4 := \times (a_1, a_1)(A_3 - A_2, A_1) \]
\[ A_5 := A_1 \uplus A_4 \]

- **Right outer join (\(\times\)).** \(\times (a_1=a_2)(R_1, R_2)\) returns the cartesian product of tuples from input relations \(R_1\) and \(R_2\) for which boolean condition \(a_1=a_2\) is true, and the tuples in \(R_2\) without a matching right tuple. Formally:

\[ \times (a_1, a_2)(R_1, R_2) \equiv A_5 \]

where:

\[ A_1 := \times (a_1, a_2)(R_1, R_2) \quad A_2 := \pi(a_2)(A_1) \]
\[ A_3 := \pi(a_2)(R_2) \quad A_4 := \times (a_2, a_2)(A_4 - A_2, A_1) \]
\[ A_5 := A_1 \uplus A_4 \]

- **Full outer join (\(\times\)).** \(\times (a_1=a_2)(R_1, R_2)\) returns the cartesian product of tuples from input relations \(R_1\) and \(R_2\) for which boolean condition \(a_1=a_2\) is true, the tuples in \(R_1\) without a matching right tuple, and the tuples in \(R_2\) without a matching left tuple. Formally:

\[ \times (a_1, a_2)(R_1, R_2) \equiv A_8 \]

where:

\[ A_1 := \times (a_1, a_2)(R_1, R_2) \]
\[ A_2 := \pi(a_1)(A_1) \quad A_3 := \pi(a_1)(R_1) \]
\[ A_4 := \times (a_1, a_1)(A_4 - A_2, A_1) \]
\[ A_5 := \pi(a_2)(A_1) \quad A_6 := \pi(a_2)(R_2) \]
\[ A_7 := \times (a_2, a_2)(A_6 - A_5, A_1) \]
\[ A_8 := A_1 \uplus A_4 \uplus A_7 \]

- **Restructuring and concatenation (\(\text{mapconcat}\)).** The operation \(\text{mapconcat}(\varphi)(R)\) applies \(\text{map}(\varphi)(R)\) and concatenates its result to the original tuple. Thus:

\[ \text{mapconcat}(\varphi)(R) \equiv \text{map}(\text{id}, \varphi)(R) \]

- **Empty (\(\text{empty}\)) and aggregate functions (\(\text{aggr}\)).** The boolean function \(\text{empty}(R)\) returns true iff \(R\) is empty. In turn, aggregate functions \(\text{aggr}\) include \(\text{count}, \text{max}, \text{min}\) and \(\text{sum}\). \(\text{count}(R)\) calculates the number elements in a bag of tuples \(R\). \(\text{max}(a)(R)\) returns the maximum integer value of an element \(a\) in a bag of tuples \(R\). \(\text{min}(a)(R)\) returns the minimum integer value of an element \(a\) in a bag of tuples \(R\). \(\text{sum}(a)(R)\) returns the sum of integer values for an element \(a\) in a bag of tuples \(R\). Each of these functions can be described in NRAB.

**B. Pig Latin - TO - NRAB TRANSLATION**

Along the lines of [26], we define our Pig Latin to NRAB translation by means of deduction (or translation) rules. In a nutshell, a rule describes how the translation is performed when some conditions are met over the input. Our rules rely on translation judgments, noted as \(J_i\), and are of the form:

\[ J_1 \ldots J_n \]

stating that the translation \(J\) (conclusion) is recursively made in terms of translations \(J_1 \ldots J_n\) (premises). The translation judgments \(J_i\) are optional.

For ease of presentation, we split the rules in two sets: the first one deals with the translation of programs as ordered sequences of expressions, while the second set details the translation of a single Pig Latin operation. Below, we present the rules sets in turn.

**Pig Latin scripts translation.** Rules in the first set are presented in Figure 14. They rely on judgments of the form \([P]^\Gamma \Rightarrow \Gamma\)', meaning that a Pig Latin program \(P\) is translated to a set of named NRAB expressions \(\Gamma\), in the context of a given set of named NRAB expressions \(\Gamma\). By rules definition, it easily follows that \(\Gamma\) always includes \(\Gamma\). A named NRAB expression is a binding of the form \(\text{var} = A\) where \text{var} is a name given to the algebraic expression \(A\). During the application of the translation rules, every binding expression \(\text{var} = \text{op}\) belonging to the Pig Latin program is translated into a named algebraic expression \(\{\text{var} = A\}\), where \(A\) is the NRAB expression corresponding to the operation \(\text{op}\) (and obtained by applying the second set of translation rules).

Binding expressions in the Pig Latin program are translated one after the other, according to their order in the program. Each time a named algebraic expression \(\{\text{var} = A\}\) is created, it is added to the context \(\Gamma\). The context holds all variables which may be encountered while translating subsequent Pig Latin binding expressions of the program; we assume that \text{var} is a fresh variable, i.e., it is not already bound in the context.

Figure 14 shows the rules used by the high-level translation process outlined above. The rules are rather simple; note that the rule corresponding to \text{store} adds to the context a dummy binding. This rule records the fact that a bag has been saved on the disk, thus the symbol \(\top\) is used instead of a variable symbol, which is not needed in this case.

**Pig Latin operations translation.** The second set of rules translates the operator \(\text{op}\) from a binding expression \(\text{var} = \text{op}\) into a NRAB expression \(A\). These rules are defined over judgements of the form \(\text{op} \Rightarrow A\), meaning that the Pig Latin operation \(\text{op}\) is translated to the NRAB expression \(A\).

A special case is the \(\text{foreach}\) operator, whose translation is not trivial as it is the main way to write complex programs in Pig Latin, e.g., it allows applying nested operations. The translation rules for this operator are shown in Figure 15. We use three different rules depending on the form of the \(\text{foreach}\) expression:

- The first rule (\(\text{Projection ForEach}\)) deals with the case of an iteration simply projecting \(n\) fields of the input relation. The rule specific to this case enables the generation of NRAB projections, playing an important role in our optimization technique. In Figure 15, \(\text{var}_1, \text{var}_2, \ldots, \text{var}_n\) are the fields to be projected from the input relation denoted by the name \(\text{var}\).

- If the previous rule does not apply, and if the \(\text{foreach}\) operator contains a \(\text{generate}\) clause with functions applied on the input relation \(\text{var}\), the second rule (\(\text{Simple ForEach}\)) is applied. In this rule, every function definition \(f\) inside the \(\text{generate}\) clause is translated to an algebraic expression \(A'\) and these expressions are applied with a \(\text{map}\) operator on each tuple in \(\text{var}\) (recall Table 1).
Rule (Complex ForEach) in Figure 15 considers FOREACH expressions containing one or more binding expressions before the GENERATE clause. Each Pig Latin operator op$_i$ is translated first into an algebraic expression $A'_i$. These algebraic expressions are then used by a mapconcat operator, which applies $A'_i$ on each tuple in $A'_{i-1}$ (or $\text{var}_i$ initially) and appends the result to the input tuple; the use of mapconcat is necessary to use local contextual information that is visible only in the scope of the translated FOREACH expression. Every function definition $f$ inside the GENERATE clause is then translated to an algebraic expression $A_n$, which are applied on each of the resulting tuples from the algebraic expression $A_n$.

We provide below an example that illustrates the (Simple FOREACH) and (Complex FOREACH) translation rules depicted in Figure 15. Recall that a complex FOREACH expression consists of one or more binding expressions before the GENERATE clause.

Consider the following Pig Latin script:

```
1 A = LOAD 'page_views' AS (user, time, wvu);
2 B = LOAD 'users' AS (name, zip);
3 R = COGROUP A BY user, B BY name;
4 S = FOREACH R {
5   X = FILTER A BY time > 300;
6   Y = FOREACH X GENERATE max(time);
7   Z = FILTER B BY zip == 9000;
8   GENERATE group, Y, count(Z);
9 }
10 STORE Z INTO 's1out';
```

Line 1 loads data from a file page_views and creates a bag of tuples that is bound to variable $A$; in turn, line 2 loads data from a second file, and binds the resulting tuple bag to $B$. Line 3 groups together the tuples of $A$ and $B$ based on the equality of the values bound to attributes user and name; recall that the tuples output by the COGROUP operator consist of attributes group, $A$, and $B$. Lines 4-9 contain a complex FOREACH expression. In particular, for each tuple in $R$, line 5 creates a nested bag $X$ with the tuples in $A$ with a value bound to time that is greater than 300; then, line 6 projects the maximum value bound to time and binds it to variable $Y$; line 7 creates a nested bag $Z$ with the tuples in $B$ with a value bound to zip that is equal to 9000; finally, in line 8 the attributes group, $Y$, and the number of tuples in bag $Z$ are generated. The result is stored in $s1out$.

The translation of the previous script yields:

$$
\Gamma = \{ A = \text{scan}('\text{page\_views}'), \\
B = \text{scan}('\text{users}'), \\
R = \text{cogroup(user, name)}(A, B), \\
S = \text{map(group, Y, count(Z))}(
\text{mapconcat}(Z = \sigma(zip == 9000)(B))(
\text{mapconcat}(Y = \text{map}(\max(time)(X))(
\text{mapconcat}(X = \sigma(time > 300)(A))(R)))), \\
\text{store('s1out')}(S) \}
$$

In the resulting context, $S$ has been generated by the (Complex ForEach) translation rule; in turn, $Y$ inside $S$ has been generated by the (Simple ForEach) rule.

Other Pig Latin operators have a one-to-one correspondence with NRAB operators, and their translation (Figure 16) is commented in the sequel.

Rule (LOAD) translates a LOAD expression into a scan that generating a new bag that satisfies the schema description in the input expression.

Rule (DISTINCT) translates DISTINCT into a $\epsilon$ operator on the input relation $\text{var}_i$.

![Figure 14: Translation rules for Pig Latin scripts and basic Pig Latin constructs.](image)

Rule (FILTER) translates a Pig Latin FILTER operator into a selection $\sigma$ with a condition $p$ on $\text{var}_i$.

Rule (FLATTEN FUNCTION) translates FLATTEN into a $\delta$ function that unnest the bag $\text{var}$. Rule (EMPTY FUNCTION) translates $\emptyset$ Pig Latin function, while rule (AGGREGATION FUNCTION) translates Pig Latin aggregation functions into their NRAB operators counterparts. Finally, rule (ATTRIBUTE FUNCTION) translates a Pig Latin attribute name into its corresponding NRAB function expression.

The functions introduced in the last four rules are blocks that need to be used in the algebra in conjunction with one of the algebra construction operators, e.g., restricting (map) or select ($\sigma$).

Rule (CROSS) translates a CROSS into a cartesian product between $\text{var}_1, \ldots, \text{var}_n$.

Rule (GOGROUP) translates a Pig Latin COGROUP operation to its algebraic equivalence cogroup that groups the tuples in $\text{var}_1, \ldots, \text{var}_n$ based on the values of attributes bound to $a_1, \ldots, a_k$.

Rule (INNER JOIN) translates an inner join JOIN operator into its algebraic counterpart $\bowtie$. Rule (LEFT OUTER JOIN) translates a Pig Latin left outer join expression into a $\bowtie$ operator, while rule (RIGHT OUTER JOIN) translates a Pig Latin right outer join expression into a $\bowtie$ operator. Finally, rule (FULL OUTER JOIN) translates a Pig Latin full outer join expression into a $\bowtie$ operator. Observe that outer joins can only be binary in Pig Latin.

C. EXTENSION TO UDFS

User-defined functions (UDFs) are extensively used in Pig Latin. These are functions that can be defined by users, implementing specific interfaces of the Pig Latin framework. Two common types of custom functions are aggregate functions that are applied to bags of tuples and return a scalar value, and filter functions that are applied to one or many attributes and return a boolean value.

Currently, PigReuse does not support UDFs; however, in future work we envision extensions to our technique to enable UDFs in PigReuse. We briefly sketch next the main points behind these extensions.

Our approach relies on distinguishing functional UDFs from non functional. Functional UDFs are those whose invocation does not have side effects, and whose result only depends on the input. This ensures that the result of the function call on a given input does not depend on when, and the context in which, the function is called. If an UDF has side effects or depends on values out of the input (i.e., current time or date provided by the system) then the UDF is not functional.

In the extension of NRAB, we assume that functions have associated a label to indicate whether they are functional
Figure 15: Translation rules for foreach operator.

Additional equivalence rules. To better integrate within our framework the reutilization of results comprising map operators, including those containing udf functions, new equivalence rules could be added to the existing set, as the following two ones, where we assume the function \( \varphi \) being functional. The first rule comes from commutativity of projection and restructuring

\[
\pi\langle a_1, \ldots, a_i\rangle(mapx(\varphi)(\var)) \equiv mapx(\varphi)(\pi\langle a_1, \ldots, a_i\rangle(\var))
\]

where \( a_1, \ldots, a_i \) includes all the attributes on which \( \varphi \) depends. A second rule comes from commutativity of selection and restructuring:

\[
\sigma(p)(mapx(\varphi)(\var)) \equiv mapx(\varphi)(\sigma(p)(\var))
\]

where \( p \) and \( \varphi \) depends on two disjoint sets of attributes, and \( \varphi \) preserves all attributes used by \( p \).

Changes to PigReuse normalization. Given that the information about the fields that a UDF accesses is available to PigReuse, accommodating e.g., an map operator containing a UDF udf in the normalization step is straightforward.

In particular, as it happens with built-in functions, a child \( \pi \) operator can be swapped with the parent operator map operator containing a UDF udf if and only if none of the fields used by the parent operator is projected by \( \pi \).

A special case of syntactic dependency arises when the given UDF consults tuple metadata, e.g., counting the number of fields in a tuple, which is not possible in the traditional algebraic context. In this case, we could consider that all fields in the tuple are accessed, and thus, the child \( \pi \) operator cannot be swapped with the parent operator.

D. ALGEBRA EQUIVALENCES

To detect algebra expressions equivalences, PigReuse relies on the logical properties of these expressions [18]. In the following, we enumerate the different laws that PigReuse uses to set these logical properties, which have been extensively studied previously [3, 9, 24, 25].

**Equivalence 1. Casing of selections:**

\[
\sigma(p_1)(\sigma(p_2)(\ldots(\sigma(p_n)(\var)\ldots)) \equiv \sigma(p_1 \land p_2 \land \ldots \land p_n)(\var)
\]

**Equivalence 2. Commutativity of selection:**

\[
\sigma(p_1)(\sigma(p_2)(\var)) \equiv \sigma(p_2)(\sigma(p_1)(\var))
\]

**Equivalence 3. Casing of projections:**

\[
\pi(C_1)(\pi(C_2)(\ldots(\pi(C_n)(\var)\ldots)) \equiv \pi(C_1)(\var)
\]
where $C_i$ is a set of columns such that $C_i \subseteq C_{i+1}$, $\forall i = 1, \ldots, n - 1$.

**Equivalence 4.** Cascading of additive union:
\[
\var_1 \uplus (\var_2 \uplus (\ldots \uplus (\var_{n-1} \uplus \var_n) \ldots)) \equiv \var_1 \uplus \var_2 \uplus \ldots \uplus \var_n.
\]

**Equivalence 5.** Commutativity of additive union:
\[
\var_1 \uplus \var_2 \equiv \var_2 \uplus \var_1.
\]

**Equivalence 6.** Associativity of additive union:
\[
\var_1 \uplus (\var_2 \uplus \var_3) \equiv (\var_1 \uplus \var_2) \uplus \var_3.
\]

**Equivalence 7.** Cascading of cross:
\[
\var_1 \times (\var_2 \times (\ldots \times (\var_{n-1} \times \var_n) \ldots)) \equiv \var_1 \times \var_2 \times \ldots \times \var_n.
\]

**Equivalence 8.** Commutativity of cross:
\[
\var_1 \times \var_2 \equiv \var_2 \times \var_1.
\]

**Equivalence 9.** Associativity of cross:
\[
(\var_1 \times \var_2) \times \var_3 \equiv (\var_1 \times \var_3) \times \var_2.
\]

**Equivalence 10.** Commutativity of cogroup:
\[
cogroup(a_1, a_2)(\var_1, \var_2) \equiv cogroup(a_2, a_1)(\var_2, \var_1).
\]

**Equivalence 11.** Cascading of inner join:
\[
\Join (a_1, a_2)(\var_1, \Join (a_2, a_3)(\var_2, \ldots, \\
\Join (a_{n-1}, a_n)(\var_{n-1}, \var_n) \ldots)) \equiv \\
\Join (a_1, a_2, \ldots, a_n)(\var_1, \var_2, \ldots, \var_n).
\]

**Equivalence 12.** Commutativity of inner join:
\[
\Join (a_1, a_2)(\var_1, \var_2) \equiv \Join (a_2, a_1)(\var_2, \var_1).
\]

**Equivalence 13.** Associativity of inner join:
\[
\Join (a_1, a_2)(\var_1, \Join (a_2, a_3)(\var_2, \var_3)) \equiv \Join (a_2, a_3)(\join (a_1, a_2)(\var_1, \var_2), \var_3).
\]

**Equivalence 14.** Cascading of full outer join:
\[
\Join (a_1, a_2)(\var_1, \Join (a_2, a_3)(\var_2, \var_3), \ldots, \\
\Join (a_{n-1}, a_n)(\var_{n-1}, \var_n) \ldots) \equiv \\
\Join (a_1, a_2, \ldots, a_n)(\var_1, \var_2, \ldots, \var_n).
\]

**Equivalence 15.** Commutativity of full outer join:
\[
\Join (a_1, a_2)(\var_1, \var_2) \equiv \Join (a_2, a_1)(\var_2, \var_1).
\]

**Equivalence 16.** Associativity of full outer join:
\[
\Join (a_1, a_2)(\var_1, \Join (a_2, a_3)(\var_2, \var_3)) \equiv \\
\Join (a_2, a_3)(\join (a_1, a_2)(\var_1, \var_2), \var_3).
\]

**Equivalence 17.** Commutativity of selection and projection:
\[
\sigma(p)(\pi(a_1, \ldots, a_n)(\var)) \equiv \pi(a_1, \ldots, a_n)(\sigma(p)(\var))
\]
where every attribute mentioned in $p$ must be included in $a_1, \ldots, a_n$.

**Equivalence 18.** Commutativity of selection and cross:
\[
\sigma(p)(\var \times \var_2) \equiv \sigma(p)(\var_1) \times \var_2
\]
where all attributes in $p$ belong to $\var$. In general, a selection can be replaced by a cascade of selections, and then some of the resulting selections might commute with the cross operator.

**Equivalence 19.** Commutativity of selection and inner join:
\[
\sigma(p)(\Join (a_1, a_2)(\var_1, \var_2)) \equiv \Join (a_1, a_2)(\sigma(p)(\var_1), \var_2)
\]
where all attributes in $p$ belong to $\var$. In general, a selection can be replaced by a cascade of selections, and then some of the resulting selections might commute with the join operator.

**Equivalence 20.** Commutativity of selection and left outer join:
\[
\sigma(p)(\Join (a_1, a_2)(\var_1, \var_2)) \equiv \Join (a_1, a_2)(\sigma(p)(\var_1), \var_2)
\]
where all attributes in $p$ belong to $\var$. A selection can only be pushed to the left input of a left outer join operator.

**Equivalence 21.** Commutativity of selection and right outer join:
\[
\sigma(p)(\Join (a_1, a_2)(\var_1, \var_2)) \equiv \Join (a_1, a_2)(\sigma(p)(\var_1), \var_2)
\]
where all attributes in $p$ belong to $\var$. A selection can only be pushed to the right input of a right outer join operator.

**Equivalence 22.** Commutativity of projection and inner join:
\[
\pi(a_1, \ldots, a_i, a_{i+1}, \ldots, a_n)(\Join (a_2, a_y)(\var_1, \var_2)) \equiv \\
\Join (a_2, a_y)(\pi(a_1, \ldots, a_i)(\var_1), \pi(a_{i+1}, \ldots, a_n)(\var_2))
\]
where attributes $a_1, \ldots, a_i$ belong to $\var_1$, while attributes $a_{i+1}, \ldots, a_n$ belong to $\var_2$. Note that the attributes $a_x, a_y$ must be contained in $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n$.

**Equivalence 23.** Commutativity of projection and left outer join:
\[
\pi(a_1, \ldots, a_i, a_{i+1}, \ldots, a_n)(\Join (a_2, a_y)(\var_1, \var_2)) \equiv \\
\Join (a_2, a_y)(\pi(a_1, \ldots, a_i)(\var_1), \pi(a_{i+1}, \ldots, a_n)(\var_2))
\]
where attributes $a_1, \ldots, a_i$ belong to $\var_1$, while attributes $a_{i+1}, \ldots, a_n$ belong to $\var_2$. Note that the attributes $a_x, a_y$ must be contained in $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n$.

**Equivalence 24.** Commutativity of projection and right outer join:
\[
\pi(a_1, \ldots, a_i, a_{i+1}, \ldots, a_n)(\Join (a_2, a_y)(\var_1, \var_2)) \equiv \\
\Join (a_2, a_y)(\pi(a_1, \ldots, a_i)(\var_1), \pi(a_{i+1}, \ldots, a_n)(\var_2))
\]
where attributes $a_1, \ldots, a_i$ belong to $\var_1$, while attributes $a_{i+1}, \ldots, a_n$ belong to $\var_2$. Note that the attributes $a_x, a_y$ must be contained in $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n$.

**Equivalence 25.** Commutativity of projection and full outer join:
\[
\pi(a_1, \ldots, a_i, a_{i+1}, \ldots, a_n)(\Join (a_2, a_y)(\var_1, \var_2)) \equiv \\
\Join (a_2, a_y)(\pi(a_1, \ldots, a_i)(\var_1), \pi(a_{i+1}, \ldots, a_n)(\var_2))
\]
where attributes $a_1, \ldots, a_i$ belong to $\var_1$, while attributes $a_{i+1}, \ldots, a_n$ belong to $\var_2$. Note that the attributes $a_x, a_y$ must be contained in $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n$. 
E. EXPERIMENTAL QUERY WORKLOADS

Workload $W_1$ consists of scripts 1-12, while workload $W_2$ consists of the 20 scripts that we introduce below.

**Script 1. l2.pig. Extract the estimated revenue for the pages visited by registered users.**

$$A = \text{LOAD} \ ('page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links));$$
$$B = \text{FOREACH} A \text{ GENERATE} \text{ user, estimated_revenue};$$
$$\text{alpha} = \text{LOAD} \ ('users' AS (name, phone, address, city, state, zip);}$$
$$\text{beta} = \text{FOREACH} \text{alpha GENERATE name; }$$
$$C = \text{JOIN} B \text{ BY user, beta BY name; }$$
$$\text{STORE} C \text{ INTO 'l2out';}$$

**Script 2. l3.pig. Extract the total estimated revenue per registered user.**

$$A = \text{LOAD} \ ('page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links));$$
$$B = \text{FOREACH} A \text{ GENERATE} \text{ user, action, estimated_revenue;}$$
$$\text{alpha} = \text{LOAD} \ ('users' AS (name, phone, address, city, state, zip);}$$
$$\text{beta} = \text{FOREACH} \text{alpha GENERATE name; }$$
$$C = \text{JOIN} B \text{ BY user, beta BY name; }$$
$$D = \text{GROUP} C \text{ BY user;}$$
$$E = \text{FOREACH} D \text{ GENERATE} \text{ group, SUM(C.estimated_revenue);}$$
$$\text{STORE} E \text{ INTO 'l3out';}$$

**Script 3. l4.pig. How many different actions has each registered user done?**

$$A = \text{LOAD} \ ('page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links));$$
$$B = \text{FOREACH} A \text{ GENERATE} \text{ user, action;}$$
$$C = \text{GROUP} B \text{ BY user }$$
$$\text{aleph} = \text{FILTER} B \text{ BY action IS null;}$$
$$\text{beth} = \text{DISTINCT} \text{aleph;}$$
$$\text{GENERATE} \text{ group, COUNT(beth);}$$
$$\text{STORE} D \text{ INTO 'l4out';}$$

**Script 4. l5.pig. List the page visitors that are not registered users.**

$$A = \text{LOAD} \ ('page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links));$$
$$B = \text{FOREACH} A \text{ GENERATE} \text{ user, action, timespent, query_term, estimated_revenue, page_info, page_links;}$$
$$\text{alpha} = \text{LOAD} \ ('users' AS (name, phone, address, city, state, zip);}$$
$$\text{beta} = \text{ FOREACH} \text{alpha GENERATE name; }$$
$$C = \text{GROUP} B \text{ BY user, beta BY name;}$$
$$D = \text{FILTER} C \text{ BY COUNT(beta) == 0};$$
$$\text{E} = \text{FOREACH} D \text{ GENERATE} \text{ group;}$$
$$\text{STORE} E \text{ INTO 'l5out';}$$

**Script 5. l6.pig. How long did visitors that queried for a certain term stayed in the page?**

$$A = \text{LOAD} \ ('page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links));$$
$$B = \text{FOREACH} A \text{ GENERATE} \text{ user, action, timespent, query_term, estimated_revenue, page_info, page_links;}$$
$$\text{alpha} = \text{LOAD} \ ('users' AS (name, phone, address, city, state, zip);}$$
$$\text{beta} = \text{FOREACH} \text{alpha GENERATE name; }$$
$$C = \text{GROUP} B \text{ BY query_term;}$$
$$D = \text{FOREACH} C \text{ GENERATE group, SUM(B.timespent);}$$
$$\text{STORE} D \text{ INTO 'l6out';}$$
Script 11. l15.pig. Extract the number of different actions, the average spent time, and the generated revenue, per registered user.

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action, timespent, estimated_revenue;
C = GROUP B BY user;
D = FOREACH C {
    beth = DISTINCT B.action;
    ts = DISTINCT B.timespent;
    rev = DISTINCT B.estimated_revenue;
    GENERATE group, COUNT(beth), AVG(ts), SUM(rev);
}
STORE D INTO 'e8out';

Script 12. l16.pig. How much revenue did each registered user generate?

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
C = GROUP B BY user;
D = FOREACH C {
    F = B.estimated_revenue;
    GENERATE group, SUM(F);
}
STORE D INTO 'l16out';

Script 13. e1.pig. List all the registered users together with their associated page views (if any).

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = JOIN B BY user RIGHT, beta BY name;
STORE C INTO 'e1out';

Script 14. e2.pig. List all the page views and all the registered users, associating them if possible.

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = JOIN B BY user FULL, beta BY name;
STORE C INTO 'e2out';

Script 15. e3.pig. How many different actions has each registered user done?

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action;
C = GROUP B BY user;
D = FOREACH C {
    aleph = B.action;
    beth = DISTINCT aleph;
    GENERATE group, COUNT(beth);
}
STORE D INTO 'e3out';

Script 16. e4.pig. List the page views per registered user, together with their information as advanced users (if any).

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action, timespent, estimated_revenue;
C = GROUP B BY user;
D = FOREACH C {
    beth = DISTINCT B.action;
    ts = DISTINCT B.timespent;
    rev = DISTINCT B.estimated_revenue;
    GENERATE group, COUNT(beth), AVG(ts), SUM(rev);
}
STORE D INTO 'e4out';

Script 17. e5.pig. How long did visitors with the same IP address stayed in the page?

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action, timespent, query_term, ip_addr, timestamp;
C = GROUP B BY ip_addr;
D = FOREACH C GENERATE group, SUM(B(timespent));
STORE D INTO 'e5out';

Script 18. e6.pig. Extract the estimated revenue for the pages visited per registered user.

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = COGROUP B BY user, beta BY name;
STORE C INTO 'e6out';

Script 19. e7.pig. List all the users in the dataset (without repetitions).

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user;
C = DISTINCT B;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
gamma = DISTINCT beta;
D = UNION C, gamma;
E = DISTINCT D;
STORE E INTO 'e7out';

Script 20. e8.pig. Extract the number of different actions, the average spent time, and the generated revenue, per registered user.

A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action, timespent, estimated_revenue;
C = GROUP B BY user;
D = FOREACH C {
    beth = DISTINCT B.action;
    ts = DISTINCT B.timespent;
    rev = DISTINCT B.estimated_revenue;
    GENERATE group, COUNT(beth), AVG(ts), SUM(rev);
}
STORE D INTO 'e8out';