Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience

Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath,
Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed,
Santhosh Srinivasan, Utkarsh Srivastava

Yahoo!, Inc.*

ABSTRACT

Increasingly, organizations capture, transform and analyze enormous data sets. Prominent examples include internet companies and e-science. The *Map-Reduce* scalable dataflow paradigm has become popular for these applications. Its simple, explicit dataflow programming model is favored by some over the traditional high-level declarative approach: SQL. On the other hand, the extreme simplicity of Map-Reduce leads to much low-level hacking to deal with the many-step, branching dataflows that arise in practice. Moreover, users must repeatedly code standard operations such as *join* by hand. These practices waste time, introduce bugs, harm readability, and impede optimizations.

Pig is a high-level dataflow system that aims at a sweet spot between SQL and Map-Reduce. Pig offers SQL-style high-level data manipulation constructs, which can be assembled in an explicit dataflow and interleaved with custom Map- and Reduce-style functions or executables. Pig programs are compiled into sequences of Map-Reduce jobs, and executed in the *Hadoop* Map-Reduce environment. Both Pig and Hadoop are open-source projects administered by the Apache Software Foundation.

This paper describes the challenges we faced in developing Pig, and reports performance comparisons between Pig execution and raw Map-Reduce execution.

1. INTRODUCTION

Organizations increasingly rely on ultra-large-scale data processing in their day-to-day operations. For example, modern internet companies routinely process petabytes of web content and usage logs to populate search indexes and perform ad-hoc mining tasks for research purposes. The data includes unstructured elements (e.g., web page text; images) as well as structured elements (e.g., web page click records; extracted entity-relationship models). The processing combines generic relational-style operations (e.g., filter; join; count) with specialized domain-specific operations (e.g., part-of-speech tagging; face detection). A similar situation arises in e-science, national intelligence, and other domains.

The popular *Map-Reduce* [8] scalable data processing framework, and its open-source realization *Hadoop* [1], cater to these workloads and offer a simple dataflow programming model that appeals to many users. However, in practice, the extreme simplicity of the Map-Reduce programming model leads to several problems. First, it does not directly support complex N-step dataflows, which often arise in practice. Map-Reduce also lacks explicit support for combined processing of multiple data sets (e.g., joins and other data matching operations), a crucial aspect of knowledge discovery. Lastly, frequently-needed data manipulation primitives like filtering, aggregation and top-k thresholding must be coded by hand.

Consequently, users end up stitching together Map-Reduce dataflows by hand, hacking multi-input flows, and repeatedly implementing standard operations inside black-box functions. These practices slow down data analysis, introduce mistakes, make data processing programs difficult to read, and impede automated optimization.

Our *Pig* system [4] offers composable high-level data manipulation constructs in the spirit of SQL, while at the same time retaining the properties of Map-Reduce systems that make them attractive for certain users, data types, and workloads. In particular, as with Map-Reduce, Pig programs encode explicit dataflow graphs, as opposed to implicit dataflow as in SQL. As one user from Adobe put it:

> "Pig seems to give the necessary parallel programming constructs (FOREACH, FLATTEN, COGROUP .. etc) and also give sufficient control back to the programmer (which a purely declarative approach like [SQL on top of Map-Reduce][1] doesn't)."

Pig dataflows can interleave built-in relational-style operations like filter and join, with user-provided executables (scripts or pre-compiled binaries) that perform custom processing. Schemas for the relational-style operations can be supplied at the last minute, which is convenient when working with temporary data for which system-managed metadata is more of a burden than a benefit. For data used

*Author email addresses: {gates, olgan, shubhamc, pradeepk, shravanm, olston, breed, sms, utkarsh}@yahoo-inc.com.

VLDB '09, August 24-28, 2009, Lyon, France

[1]Reference to specific software project removed.

exclusively in non-relational operations, schemas need not be described at all.

Pig compiles these dataflow programs, which are written in a language called *Pig Latin* [15], into sets of Hadoop Map-Reduce jobs, and coordinates their execution. By relying on Hadoop for its underlying execution engine, Pig benefits from its impressive scalability and fault-tolerance properties. On the other hand, Pig currently misses out on optimized storage structures like indexes and column groups. There are several ongoing efforts to add these features to Hadoop.

Despite leaving room for improvement on many fronts, Pig has been widely adopted in Yahoo, with hundreds of users and thousands of jobs executed daily, and is also gaining traction externally with many successful use cases reported. This paper describes the challenges we faced in developing Pig, including implementation obstacles as well as challenges in transferring the project from a research team to a development team and converting it to open-source. It also reports performance measurements comparing Pig execution and raw Hadoop execution.

1.1 Related Work

For the most part, Pig is merely a combination of known techniques that fulfills a practical need. That need appears to be widespread, as several other systems are emerging that also offer high-level languages for Map-Reduce-like environments: DryadLINQ [20], Hive [3], Jaql [5], Sawzall [16] and Scope [6]. With the exception of Sawzall, which provides a constrained filter-aggregate abstraction on top of a single Map-Reduce job, these systems appear to have been developed after or concurrently with Pig. Some of these systems adopt SQL syntax (or a close variant), whereas others intentionally depart from SQL, presumably motivated by scenarios for which SQL was not deemed the best fit.

1.2 Outline

Rather than trying to be comprehensive, this paper focuses on aspects of Pig that are somewhat non-standard compared to conventional SQL database systems. After giving an overview of the system, we describe Pig's type system (including nested types, type inference and lazy casting), generation, optimization and execution of query plans in the Map-Reduce context, and piping data through user-supplied executables ("streaming"). We then present performance numbers, comparing Pig execution against hand-coded Map-Reduce execution. At the end of the paper, we describe some of our experiences building and deploying Pig, and mention some of the ways Pig is being used (both inside and outside of Yahoo).

2. SYSTEM OVERVIEW

The Pig system takes a Pig Latin program as input, compiles it into one or more Map-Reduce jobs, and then executes those jobs on a given Hadoop cluster. We first give the reader a flavor of Pig Latin through a quick example, and then describe the various steps that are carried out by Pig to execute a given Pig Latin program.

EXAMPLE 1. *Consider a data set* `urls: (url, category, pagerank)`. *The following Pig Latin program finds, for each sufficiently large category, the top ten urls in that category by pagerank.*

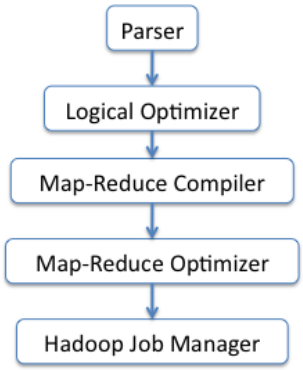

Figure 1: Pig compilation and execution stages.

```
urls = LOAD 'dataset' AS (url, category, pagerank);
groups = GROUP urls BY category;
bigGroups = FILTER groups BY COUNT(urls)>1000000;
result = FOREACH bigGroups GENERATE
            group, top10(urls);
STORE result INTO 'myOutput';
```

Some of the salient features of Pig Latin as demonstrated by the above example include (a) a step-by-step dataflow language where computation steps are chained together through the use of variables, (b) the use of high-level transformations, e.g., `GROUP`, `FILTER`, (c) the ability to specify schemas as part of issuing a program, and (d) the use of user-defined functions (e.g., `top10`) as first-class citizens. More details about Pig Latin and the motivations for its design are given in [15].

Pig allows three modes of user interaction:

1. **Interactive mode**: In this mode, the user is presented with an interactive shell (called *Grunt*), which accepts Pig commands. Plan compilation and execution is triggered only when the user asks for output through the `STORE` command. (This practice enables Pig to plan over large blocks of program logic. There are no transactional consistency concerns, because Hadoop data is immutable.)

2. **Batch mode**: In this mode, a user submits a pre-written script containing a series of Pig commands, typically ending with `STORE`. The semantics are identical to interactive mode.

3. **Embedded mode**: Pig is also provided as a Java library allowing Pig Latin commands to be submitted via method invocations from a Java program. This option permits dynamic construction of Pig Latin programs, as well as dynamic control flow, e.g. looping for a non-predetermined number of iterations, which is not currently supported in Pig Latin directly.

In interactive mode, two commands are available to help the user reason about the program she is using or creating: `DESCRIBE` and `ILLUSTRATE`. The `DESCRIBE` command displays the schema of a variable (e.g. `DESCRIBE urls`, `DESCRIBE bigGroups`). The `ILLUSTRATE` command displays a small amount of example data for a variable and the variables in

its derivation tree, to give a more concrete illustration of the program semantics; the technology behind Pig's example data generator is described in [14]. These features are especially important in the Pig context, given the complexity of dealing with nested data, partially-specified schemas, and inferred types (see Section 3).

Regardless of the mode of execution used, a Pig program goes through a series of transformation steps before being executed, depicted in Figure 1.

The first step is parsing. The parser verifies that the program is syntactically correct and that all referenced variables are defined. The parser also performs type checking and schema inference (see Section 3). Other checks, such as verifying the ability to instantiate classes corresponding to user-defined functions and confirming the existence of streaming executables referenced by the user's program, also occur in this phase. The output of the parser is a canonical logical plan with a one-to-one correspondence between Pig Latin statements and logical operators, arranged in a directed acyclic graph (DAG).

The logical plan generated by the parser is passed through a *logical optimizer*. In this stage, logical optimizations such as projection pushdown are carried out. The optimized logical plan is then compiled into a series of Map-Reduce jobs (see Section 4), which then pass through another optimization phase. An example of Map-Reduce-level optimization is utilizing the Map-Reduce *combiner* stage to perform early partial aggregation, in the case of distributive or algebraic [12] aggregation functions.

The DAG of optimized Map-Reduce jobs is then topologically sorted, and jobs are submitted to Hadoop for execution in that order (opportunities for concurrent execution of independent branches are not currently exploited). Pig monitors the Hadoop execution status, and periodically reports the progress of the overall Pig program to the user. Any warnings or errors that arise during execution are logged and reported to the user.

3. TYPE SYSTEM AND TYPE INFERENCE

Pig has a nested data model, thus supporting complex, non-normalized data. Standard scalar types of `int`, `long`, `double`, and `chararray` (string) are supported. Pig also supports a `bytearray` type that represents a collection of uninterpreted bytes. The type `bytearray` is also used to facilitate unknown data types and lazy conversion of types, as described in Sections 3.1 and 3.2.

Pig supports three complex types: `map`, `tuple`, and `bag`. `map` is an associative array, where the key is a `chararray` and the value is of any type. We chose to include this type in Pig because much of Yahoo's data is stored in this way due to sparsity (many tuples only contain a subset of the possible attributes). `tuple` is an ordered list of data elements. The elements of `tuple` can be of any type, thus allowing nesting of complex types. `bag` is a collection of tuples.

When Pig loads data from a file (and conversely when it stores data into a file), it relies on *storage functions* to delimit data values and tuples. The default storage function uses ASCII character encoding. It uses tabs to delimit data values and carriage returns to delimit tuples, and left/right delimiters like { } to encode nested complex types. Pig also comes with a binary storage function called `BinStorage`. In addition, users are free to define their own storage function, e.g., to operate over files emitted by another system, to create data for consumption by another system, or to perform specialized compression. To use a storage function other than the default one, the user gives the name of the desired storage function in the `LOAD` or `STORE` command.

3.1 Type Declaration

Data stored in Hadoop may or may not have schema information stored with it. For this reason, Pig supports three options for declaring the data types of fields. The first option is that no data types are declared. In this case the default is to treat all fields as `bytearray`. For example:

```
a = LOAD 'data' USING BinStorage AS (user);
b = ORDER a BY user;
```

The sorting will be done by byte radix order in this case. Using `bytearray` as the default type avoids unnecessary casting of data that may be expensive or corrupt the data.

Note that even when types are undeclared, Pig may be able to infer certain type information from the program. If the program uses an operator that expects a certain type on a field, then Pig will coerce that field to be of that type. In the following example, even though `addr` is not declared to be a `map`, Pig will coerce it to be one since the program applies the map dereference operator `#` to it:

```
a = LOAD 'data' USING BinStorage AS (addr);
b = FOREACH a GENERATE addr#'state';
```

Another case where Pig is able to know the type of a field even when the program has not declared types is when operators or user-defined functions (UDFs) have been applied whose return type is known. In the following example, Pig will order the output data numerically since it knows that the return type of `COUNT` is `long`:

```
a = LOAD 'data' USING BinStorage AS (user);
b = GROUP a BY user;
c = FOREACH b GENERATE COUNT(a) AS cnt;
d = ORDER c BY cnt;
```

The second option for declaring types in Pig is to provide them explicitly as part of the `AS` clause during the `LOAD`:

```
a = LOAD 'data' USING BinStorage
                AS (user:chararray);
b = ORDER a BY user;
```

Pig now treats `user` as a `chararray`, and the ordering will be done lexicographically rather than in byte order.

The third option for declaring types is for the load function itself to provide the schema information, which accommodates self-describing data formats such as JSON. We are also developing a system catalog that maintains schema information (as well as physical attributes) of non-self-describing data sets stored in a Hadoop file system instance.

It is generally preferable to record schemas (via either self-description or a catalog) for data sets that persist over time or are accessed by many users. The option to forgo stored schemas and instead specify (partial) schema information through program logic (either explicitly with `AS`, or implicitly via inference) lowers the overhead for dealing with transient data sets, such as data imported from a foreign source for one-time processing in Hadoop, or a user's "scratch" data files.

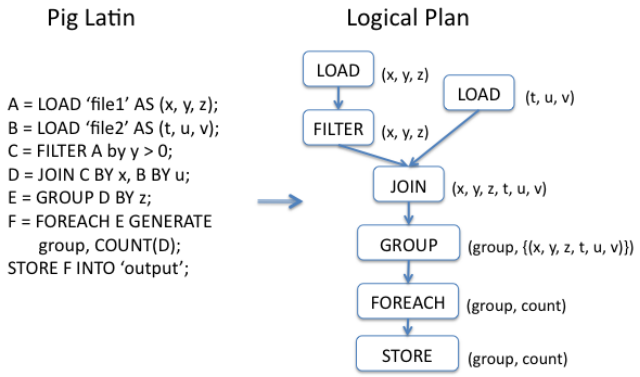

Figure 2: Pig Latin to logical plan translation.

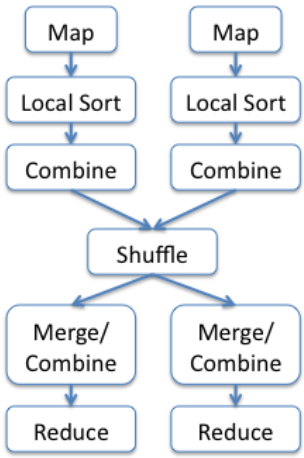

Figure 3: Map-Reduce execution stages.

3.2 Lazy Conversion of Types

When Pig does need to cast a `bytearray` to another type because the program applies a type-specific operator, it delays that cast to the point where it is actually necessary. Consider this example:

```
students = LOAD 'data' USING BinStorage
   AS (name, status, possiblePoints, earnedPoints);
paid = FILTER students BY status == 'paid';
gpa = FOREACH paid GENERATE name,
        earnedPoints / possiblePoints;
```

In this example, `status` will need to be cast to a `chararray` (since it is compared to constant of type `chararray`), and `earnedPoints` and `possiblePoints` will need to be cast to `double` since they are operands of the division operator. However, these casts will not be done when the data is loaded. Instead, they will be done as part of the comparison and division operations, which avoids casting values that are removed by the filter before the result of the cast is used.

4. COMPILATION TO MAP-REDUCE

This section describes the process of translating a logical query plan into a Map-Reduce execution plan. We describe each type of plan, and then explain how Pig translates between them and optimizes the Map-Reduce plan.

4.1 Logical Plan Structure

Recall from Section 2 that a Pig Latin program is translated in a one-to-one fashion to a *logical plan*. Figure 2 shows an example. Each operator is annotated with the schema of its output data, with braces indicating a bag of tuples.[2] With the exception of *nested plans* (Section 5.1.1) and *streaming* (Section 6), a Pig logical query plan resembles relational algebra with user-defined functions and aggregates.

Pig currently performs a limited suite of logical optimizations to transform the logical plan, before the compilation into a Map-Reduce plan. We are currently enriching the set of optimizations performed, to include standard System-R-style heuristics like filter pushdown, among others. Join ordering does not appear to be an important issue in the Pig/Hadoop context, because data is generally kept in non-normalized form (after all, it is read-only); in practice Pig programs seldom perform more than one join. On the other hand, due to the prevalence of "wide" data tables, we do expect to encounter optimization opportunities of the form studied in the column-store context (e.g. deferred stitching), once column-wise storage structures are added to Hadoop.

4.2 Map-Reduce Execution Model

A Hadoop Map-Reduce job consists of a series of execution stages, shown in Figure 3. The *map* stage processes the raw input data, one data item at a time, and produces a stream of data items annotated with keys. A subsequent *local sort* stage orders the data produced by each machine's map stage by key. The locally-ordered data is then passed to an (optional) *combiner* stage for partial aggregation by key.

The *shuffle* stage then redistributes data among machines to achieve a global organization of data by key (e.g. globally hashed or ordered). All data received at a particular machine is combined into a single ordered stream in the *merge* stage. If the number of incoming streams is large (relative to a configured threshold), a multi-pass merge operation is employed; if applicable, the *combiner* is invoked after each intermediate merge step. Lastly, a *reduce* stage processes the data associated with each key in turn, often performing some sort of aggregation.

4.3 Logical-to-Map-Reduce Compilation

Pig first translates a logical plan into a *physical plan*, and then embeds each physical operator inside a Map-Reduce stage to arrive at a Map-Reduce plan.[3]

[2]Note that the keyword "group" is used both as a command (as in "`GROUP D BY ...`") and as the automatically-assigned field name of the group key in the output of a group-by expression (as in "`FOREACH E GENERATE group, ...`").

[3]Pig can also target platforms other than Map-Reduce. For example, Pig supports a "local" execution mode in which physical plans are executed in a single JVM on one machine (the final physical-to-Map-Reduce phase is not performed in this case). A student at UMass Amherst extended Pig to execute in the Galago [19] parallel data processing environment.

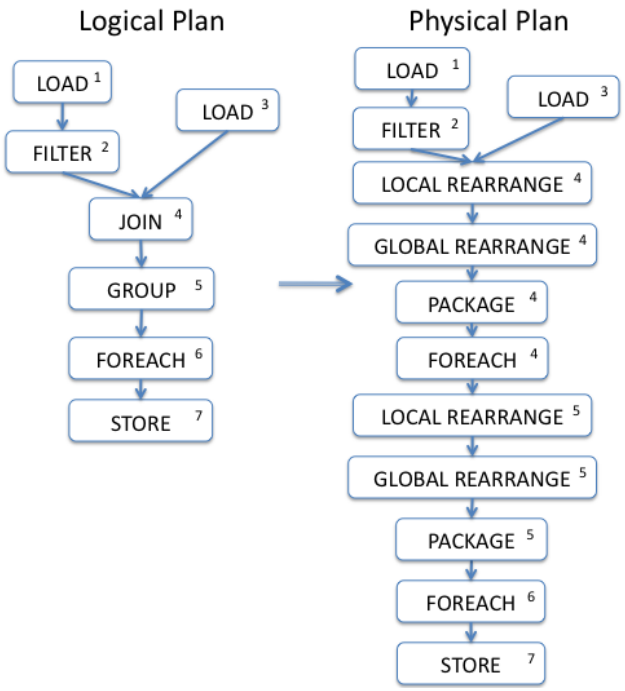

Figure 4: Logical plan to physical plan translation.

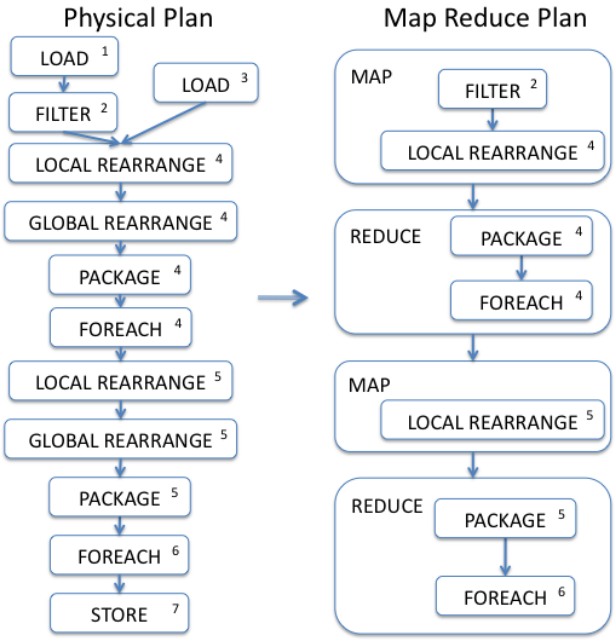

Figure 5: Physical plan to map reduce plan translation.

Figure 4 shows our example logical plan translated to a physical plan. For clarity each logical operator is shown with an id. Physical operators that are produced by the translation of a logical operator are shown with the same id. For the most part, each logical operator becomes a corresponding physical operator.

The logical `(CO)GROUP` operator becomes a series of three physical operators: *local rearrange*, *global rearrange*, and *package*. Rearrange is a term that stands for either hashing or sorting by key. The combination of local and global rearrange results in the data being arranged such that all tuples having the same group-by key wind up on the same machine and adjacent in the data stream. In the case of cogrouping multiple incoming streams, the local rearrange operator first annotates each tuple in a way that indicates its stream of origin. The package operator places adjacent same-key tuples into a single-tuple "package," which consists of the key followed by one bag of tuples per stream of origin.

The `JOIN` operator is handled in one of two ways: (1) rewrite into `COGROUP` followed by a `FOREACH` operator to perform "flattening" (see [15]), as shown in Figure 4, which yields a parallel hash-join or sort-merge join, or (2) fragment-replicate join [10], which executes entirely in the map stage or entirely in the reduce stage (depending on the surrounding operations). The choice of join strategy is controlled via syntax (a future version of Pig may offer the option to automate this choice).

Having constructed a physical plan, Pig assigns physical operators to Hadoop stages (Section 4.2), with the goal of minimizing the number of reduce stages employed. Figure 5 shows the assignment of physical operators to Hadoop stages for our running example (only the map and reduce stages are shown). In the Map-Reduce plan, the local rearrange operator simply annotates tuples with keys and stream identifiers, and lets the Hadoop local sort stage do the work. Global rearrange operators are removed because their logic is implemented by the Hadoop shuffle and merge stages. Load and store operators are also removed, because the Hadoop framework takes care of reading and writing data.

4.3.1 *Branching Plans*

If a Pig Latin program contains more than one `STORE` command, the generated physical plan contains a `SPLIT` physical operator. The following program contains a logical `SPLIT` command and ends with two `STORE` commands, one for each branch of the split:

```
clicks = LOAD 'clicks'
          AS (userid, pageid, linkid, viewedat);
SPLIT clicks INTO
          pages IF pageid IS NOT NULL,
          links IF linkid IS NOT NULL;
cpages = FOREACH pages GENERATE userid,
          CanonicalizePage(pageid) AS cpage,
          viewedat;
clinks = FOREACH links GENERATE userid,
          CanonicalizeLink(linkid) AS clink,
          viewedat;
STORE cpages INTO 'pages';
STORE clinks INTO 'links';
```

The Map-Reduce plan for this program is shown in Figure 6 (in this case, we have a "Map-only" plan, in which the Reduce step is disabled). Pig physical plans may contain nested sub-plans, as this example illustrates. Here, the split operator feeds a copy of its input to two nested sub-plans, one for each branch of the logical split operation. (The reason for using a nested operator model for split has to do with flow control during execution, as discussed later in Section 5.1.)

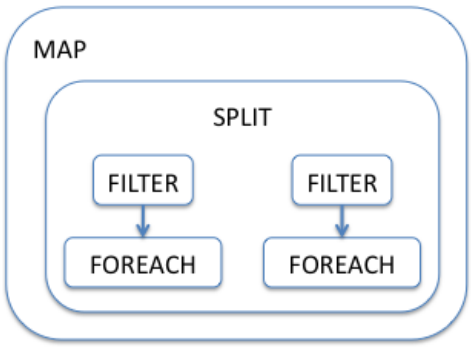

Figure 6: Split operator with nested sub-plans.

The situation becomes tricker if the split propagates across a map/reduce boundary, which occurs in the following example:

```
clicks = LOAD 'clicks'
          AS (userid, pageid, linkid, viewedat);
goodclicks = FILTER clicks BY
          viewedat IS NOT NULL;
bypage = GROUP goodclicks BY pageid;
cntbypage = FOREACH bypage GENERATE group,
          COUNT(goodclicks);
STORE cntbypage INTO 'bypage';
bylink = GROUP goodclicks BY linkid;
cntbylink = FOREACH bylink GENERATE group,
          COUNT(goodclicks);
STORE cntbylink INTO 'bylink';
```

Here there is no logical `SPLIT` operator, but the Pig compiler inserts a physical `SPLIT` operator to duplicate the `goodclicks` data flow. The Map-Reduce plan is shown in Figure 7. Here, the `SPLIT` operator tags each output tuple with the sub-plan to which it belongs. The `MULTIPLEX` operator in the Reduce stage routes tuples to the correct sub-plan, which resume where they left off.

The `SPLIT`/`MULTIPLEX` physical operator pair is a recent addition to Pig motivated by the fact that users often wish to process a data set in multiple ways, but do not want to pay the cost of reading it multiple times. Of course, this feature has a downside: Adding concurrent data processing pipelines reduces the amount of memory available for each pipeline. For memory-intensive computations, this approach may lead to spilling of data to disk (Section 5.2), which can outweigh the savings from reading the input data only once. Also, pipeline multiplexing reduces the effectiveness of the combiner (discussed next, in Section 4.4), since each run of the combiner only operates on as much data as it can fit into memory. By multiplexing pipelines, a smaller portion of a given pipeline's data is held in memory, and thus less data reduction is achieved from each run of the combiner.

Pig does not currently have an optimizer sophisticated enough to reason about this tradeoff. Thus, Pig leaves the decision to the user. The unit of compilation and execution is a Pig program submitted by the user, and Pig assumes that if the program contains multiple `STORE` commands they should be multiplexed. If the user does not wish them to be multiplexed, she can submit the pipelines independently as separate Pig programs.

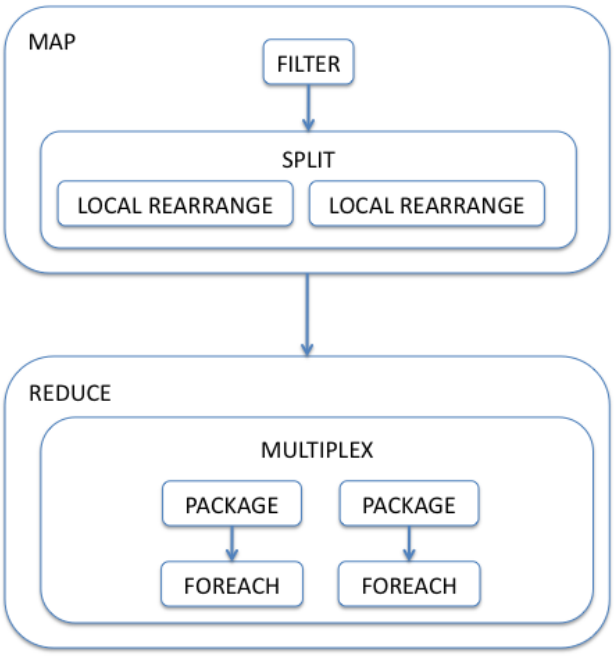

Figure 7: Split and multiplex operators.

4.4 Map-Reduce Optimization and Job Generation

Once a Map-Reduce plan has been generated, there may be room for additional optimizations. Currently only one optimization is performed at this level: Pig breaks *distributive* and *algebraic* [12] aggregation functions (such as `AVERAGE`) into a series of three steps: *initial* (e.g. generate (sum, count) pairs), *intermediate* (e.g. combine n (sum, count) pairs into a single pair), *final* (e.g. combine n (sum, count) pairs and take the quotient).[4] These steps are assigned to the map, combine, and reduce stages respectively.[5]

We have found that using the combiner as aggressively as possible has two benefits. The first is obvious: it typically reduces the volume of data handled by the shuffle and merge phases, which often consume a significant portion of job execution time. Second, combining tends to equalize the amount of data associated with each key, which lessens skew in the reduce phase.

The final compilation step is to convert each Map-Reduce combination (or just Map, if there is no final Reduce) into

[4]The separation of initial from intermediate is necessary because recent versions of Hadoop do not guarantee that every tuple will pass through the combiner (these semantics lead to greater flexibility in the Hadoop layer, e.g. bypassing the combiner in cases of a single data item for a given key).

[5]Although we could have included this optimization as part of the main plan generation process (i.e. create separate physical operators for the three aggregation steps), we opted to implement it as a separate transformation to make it more modular and hence easier to interleave with other (future) optimizations, and also easy to disable for testing and microbenchmarking purposes.

a Hadoop job description that can be passed to Hadoop for execution. This step involves generating a Java jar file that contains the Map and Reduce implementation classes, as well as any user-defined functions that will be invoked as part of the job. Currently, the Map and Reduce classes contain general-purpose dataflow execution engines (described next in Section 5), which are configured at runtime to implement the designated plan. In the future we plan to consider code generation (see Section 10).

5. PLAN EXECUTION

This section describes the way Pig executes the portion of a physical plan that lies inside a Map or Reduce stage.

5.1 Flow Control

To control movement of tuples through the execution pipeline, we considered both a push model and a pull (iterator) model [11]. We were attracted to the iterator model, which has a simple single-threaded[6] implementation that avoids context-switching overhead. Another advantage of the iterator model is that it leads to simple APIs for user-defined functions, which are especially important in the Pig/Hadoop context because of the prevalence of custom processing needs. A push model especially complicates the API (and implementation) of a UDF with multiple inputs. For the same reason, implementing binary operators like fragment-replicate join can be more difficult in the push case.

On the other hand, two drawbacks of the iterator model caused some concern. One drawback in the Pig context is that for operations over a bag nested inside a tuple (e.g. duplicate elimination on a nested bag, aggregation following group-by), the entire bag must be constituted before being passed to the operation. If the bag overflows memory, the cost of spilling to disk must be paid. In practice most operations over bags can make use of the combiner, such that memory overflows are handled by combining tuples rather than spilling the raw data to disk. Indeed, if an operation is not amenable to a combiner (e.g. a holistic UDF) then materializing the entire bag is generally an intrinsic requirement; a push-based implementation would lead to the operation performing its own bag materialization internally. It is preferable for spilling to be managed as part of the built-in infrastructure, for obvious reasons.

Another potential drawback of the iterator model is that if the data flow graph has multiple sinks, then depending on the mechanism used to "pull" data from the sinks, operators at branch points may be required to buffer an unbounded number of tuples. We sidestepped this problem by extending the iterator model as follows. When an operator is asked to produce a tuple, it can respond in one of three ways: (1) return a tuple, (2) declare itself finished, or (3) return a *pause signal* to indicate that it is not finished but also not able to produce an output tuple at this time. Pause signals propagate down the chain of operators. Their purpose is to facilitate synchronization of multiple branches of a data flow graph, and thereby minimizing buffering of tuples at branch points.

In a Pig physical plan, the outermost data flow graph always has exactly one sink and makes no use of pause signals. Our `SPLIT` and `MULTIPLEX` operators (Section 4.3.1) are special operators that behave the same as regular operators from the point of view of the outermost data flow graph, but internally they manage nested, branching data flows with buffering and pause signals. We describe the mechanics of the `SPLIT` operator (the `MULTIPLEX` case is similar).

During execution, `SPLIT` maintains a one-tuple input buffer for each branch of its nested data flow graph (called a *sub-flow*). If a sub-flow issues a read request to an empty buffer, it receives a pause signal. Each time the `SPLIT` operator is asked to produce a tuple, it selects a sub-flow and asks the sub-flow's sink operator to produce a tuple. If this operator returns a pause signal, the `SPLIT` operator moves on to another sub-flow that is not currently paused. If all sub-flows are paused, the `SPLIT` operator requests a tuple from its parent in the outer data flow graph, places a copy of the returned tuple in each sub-flow's input buffer,[7] sets the status of all sub-flows to "not paused," and proceeds.

5.1.1 Nested Programs

Pig permits a limited form of nested programming, whereby certain Pig operators can be invoked over bags nested within tuples. For example:

```
clicks = LOAD 'clicks'
          AS (userid, pageid, linkid, viewedat);
byuser = GROUP clicks BY userid;
result = FOREACH byuser {
          uniqPages = DISTINCT clicks.pageid;
          uniqLinks = DISTINCT clicks.linkid;
          GENERATE group, COUNT(uniqPages),
             COUNT(uniqLinks);
};
```

This program computes the number of distinct pages and links visited by each user. The `FOREACH` operator considers one "user tuple" at a time, and the `DISTINCT` operators nested inside it operate over the bag of "click tuples" that have been placed into the "user tuple" by the preceding `GROUP` operator.

In the physical plan Pig generates for the above Pig Latin program, there is an outer operator graph containing a `FOREACH` operator, which contains a nested operator graph consisting of two pipelines, each with its own `DISTINCT` and `COUNT` operator. Execution occurs as follows. When the outer `FOREACH` operator is asked to produce a tuple, it first requests a tuple T from its parent (in this case it will be a `PACKAGE` operator compiled from the `GROUP` logical operator). The `FOREACH` operator then initializes a cursor over the bag of click tuples inside T for the first `DISTINCT`–`COUNT` pipeline and requests a tuple from the bottom of the pipeline (the `COUNT` operator). This process is repeated for the second `DISTINCT`–`COUNT` pipeline, at which point the `FOREACH` operator constructs its output tuple (the group key plus the two counts) and returns it.

A more complex situation arises when the nested plan is not two independent pipelines (as above), but rather a single branching pipeline, as in the following example:

[6] "Streaming" constitutes an exception to our single-threaded execution model; see Section 6.

[7] The semantics of the pause signal ensure that if all sub-flows are paused, then all input buffers are empty and ready to be populated with a new tuple.

```
clicks = LOAD 'clicks'
          AS (userid, pageid, linkid, viewedat);
byuser = GROUP clicks BY userid;
result = FOREACH byuser {
          fltrd = FILTER clicks BY
             viewedat IS NOT NULL;
          uniqPages = DISTINCT fltrd.pageid;
          uniqLinks = DISTINCT fltrd.linkid;
          GENERATE group, COUNT(uniqPages),
             COUNT(uniqLinks);
};
```

Pig currently handles this case by duplicating the `FILTER` operator and producing two independent pipelines, to be executed as explained above. Although duplicating an operator obviously leads to wasted CPU effort, it does allow us to conserve memory by executing only one pipeline at a time and thus reduces the danger of triggering a spill. (Indeed, in the above examples, each pipeline may have a substantial memory footprint due to the `DISTINCT` operator, which buffers tuples internally.) This approach seems to work well for the use cases we have encountered so far.

5.2 Memory Management

Following Hadoop, Pig is implemented in Java. We have encountered many of the same difficulties related to Java memory management during query processing as [18]. The difficulties stem from the fact that Java does not allow the developer to control memory allocation and deallocation directly, whereas traditional database memory management techniques rely on the ability to control memory allocation to individual tasks, control when memory is deallocated, and accurately track memory usage.

A naive option is to increase the JVM memory size limit beyond the physical memory size, and let the virtual memory manager take care of staging data between memory and disk. Our experiments have confirmed the common wisdom that doing so leads to very severe performance degradation.

Even the appealing option of specifying a large JVM size but attempting to spill large bags before the physical memory size is reached turns out to be problematic. We have found that in practice it is generally better to return an "out-of-memory" error so that an administrator can adjust the memory management parameters and re-submit the program, compared to getting mired in long periods of thrashing that may go undetected. This trial-and-error approach to memory management is rare but does occur, and appears difficult to avoid altogether when running in a Java environment.

Most memory overflow situations in Pig arise due to materialization of large bags of tuples between and inside operators, and our memory management approach focuses on dealing with large bags. As alluded to in Section 5.1.1, Pig sometimes needs to materialize large bags inside its execution pipeline for holistic bag-wise computations. (Examples include the median function, and production of the cross-product of two bags of tuples with a common key for the inner loop of sort-merge join.) The collective size of bags in the system at a given time may exceed available memory.

Pig uses Java's `MemoryPoolMXBean` class to be notified of a low memory situation. This class allows the program to register a handler which gets notified when a configurable memory threshold is exceeded. In the event of a notification, Pig scans the list of registered bags (sorted in descending order of their estimated memory size) and spills the bags in sequence until it has recovered a configurable fraction of memory.

In the general context there is no efficient way to determine the number of bytes used by a particular bag. Hence Pig estimates bag sizes by sampling a few tuples, determining their size (by walking the nested data structures until primitive types of known size are encountered), and scaling up by the bag's cardinality.

Pig's memory manager maintains a list of all Pig bags created in the same JVM, using a linked list of Java `WeakReferences`. (The use of `WeakReference` ensures that the memory management list does not prevent garbage collection of bags no longer in use (*dead bags*), and also permits the memory manager to discover that a bag is dead so that it can be removed from the list.) The list of registered bags can grow very large and can itself consume significant memory resources. (Most bags turn out to be small and do not require spilling, but it is difficult to differentiate them a priori.) The size of the bag list is controlled in two ways: (1) when the memory manager adds a bag to the list, it looks for dead bags at the head of the list and removes them, (2) when the low-memory handler is activated and the list is sorted by bag size for spilling, all dead bags are removed. The process of sorting the bag list can be time-consuming, and blocks concurrent access to the list. New bags cannot be created during the sort and spill operations.

6. STREAMING

One of the goals of Pig is to allow users to incorporate custom code wherever necessary in the data processing pipeline. User-defined functions (UDFs) provide one avenue for including user code. But UDFs must be written in Java[8] and must conform to Pig's UDF interface. *Streaming*[9] allows data to be pushed through external executables as part of a Pig data processing pipeline. Thus users are able to intermix relational operations like grouping and filtering with custom or legacy executables.

Streaming executables receive their input from standard input or a file, and write output either to standard output or to a file. Special Pig Latin syntax is used to designate a streaming executable, specify the input and output pathways (including filenames, if applicable), and specify the input and output data formats used by the executable. In Pig Latin, inputs and outputs of streaming steps are semantically and syntactically the same as regular Pig data sets, and hence streaming operations can be composed with other Pig operations like `FILTER` and `JOIN`.

6.1 Flow Control

A UDF has synchronous behavior: when invoked on a given input data item, it produces an output data item (the output may be of a complex type, e.g. bags of tuples) and then returns control to the caller. In contrast, a streaming executable behaves asynchronously. It may consume many tuples before outputting any (blocking behavior), or conversely it may generate many tuples while consuming few.

[8]We would like to expand UDFs to allow languages beyond Java; see Section 10.

[9]Pig Streaming is inspired by Hadoop Streaming, which permits an external executable to serve as a Map or Reduce function.

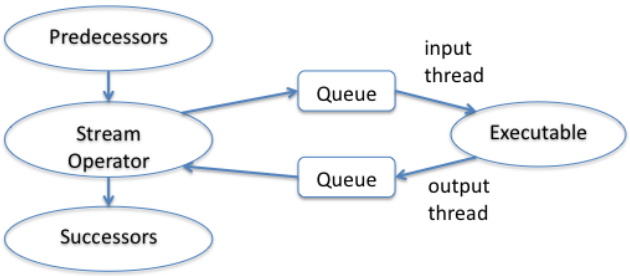

Figure 8: Data flow between a stream operator and its external executable.

One of the challenges in implementing streaming in Pig was fitting it into the iterator model of Pig's execution pipeline (Section 5.1). Due to the asynchronous behavior of the user's executable, a `STREAM` operator that wraps the executable cannot simply pull tuples synchronously as it does with other operators because it does not know what state the executable is in. There may be no output available at the moment, and the cause may be either: (1) the executable is waiting to receive more input, or (2) the executable is still busy processing prior inputs. In the former case, the stream operator needs to push new data to the operator in order to induce it to produce data. In the latter case, the stream operator should wait.

In a single-threaded operator execution model, a deadlock can occur if the Pig operator is waiting for the external executable to consume a new input tuple, while at the same time the executable is waiting for its output to be consumed (to control its output buffering). Hence we made an exception in our single-threaded execution model to handle `STREAM` operators. Each `STREAM` operator creates two additional threads: one for feeding data to the external executable, and one for consuming data from the executable. Incoming and outgoing data is kept in queues, as shown in Figure 8. When a `STREAM` operator is asked to produce an output tuple, it blocks until a tuple is available on the executable's output queue (or until the executable terminates). Meanwhile, the `STREAM` operator monitors the executable's input queue; if there is space it requests a tuple from the parent operator and places it in the queue.

7. PERFORMANCE

In the initial implementation of Pig, functionality and proof of concept were considered more important than performance. As Pig was adopted within Yahoo, better performance quickly became a priority. We designed a publicly-available benchmark called *Pig Mix* [2] to measure performance on a regular basis so that the effects of individual code changes on performance could be understood. Pig Mix exercises a wide range of Pig's functionality, and is roughly representative of the way Pig is used in Yahoo. (Actual programs and data could not be used, for confidentiality reasons.) Pig Mix consists of a suite of Pig programs and associated data sets, and also includes a raw Map-Reduce-level implementation of each of the programs.

7.1 Pig Features Exercised

The Pig features included in the Pig Mix benchmark were selected by developers, based on inspecting Pig programs collected from users. The aim was to include features that are representative of typical use cases. Some key features included in Pig Mix at present include:

- Joins (distributed hash-sort-merge join as well as fragment-replicate join).
- Grouping and cogrouping, including `GROUP ALL` in which all tuples are collected into a single group (e.g. to enable summing of values across all tuples).
- Distinct aggregation.
- Nested statements inside a `FOREACH` expression (see Section 5.1.1).
- `ORDER BY`, both on single and multiple fields.
- `UNION`, `DISTINCT` and `SPLIT`.
- Data containing nested types.

The Pig Mix benchmark consists of twelve Pig programs designed to collectively test the above features. For each Pig program, the Pig Mix has a corresponding group of Hadoop Map-Reduce programs (with the Map and Reduce functions written in Java). These Java Map-Reduce programs were written assuming a basic level of Map-Reduce programming expertise. In most of the cases, it would be possible to write a more performant Java program given additional time and effort. The decision to use a "basic" Map-Reduce programming style is based on the assumption that this is the typical alternative to writing Pig Latin programs. For example, many users do not have knowledge of advanced sorting and join algorithms. Even users who have this knowledge often are unwilling to spend additional time and effort to optimize their programs and debug the resulting complex designs.

7.2 Data Generator

Pig Mix includes a data generator that produces data with similar properties to Yahoo's proprietary data sets that are commonly processed using Pig. Some salient features of real data used by Pig, which the data generator attempts to emulate, are:

- A mixture of string, numeric (mostly integer), and map types.
- The use of doubly-nested types, particularly bags of maps.
- While 10% of the fields in user data are complex types such as maps and bags, the data in these fields often accounts for 60-80% of the data in the tuple.
- Fields used as group and join keys typically have a heavy-tailed distribution. (This aspect is especially important to capture, as it poses a particular challenge for parallel group-by and join algorithms due to skew.)

7.3 Benchmark Results

Figure 9 shows the result of running the Pig Mix benchmark on different versions of Pig developed over time. These benchmarks were run on a cluster of 26 machines; each machine has two dual-core Xeon processors, 4GB RAM, and was running Linux RHEL4. The key improvements associated with each version are:

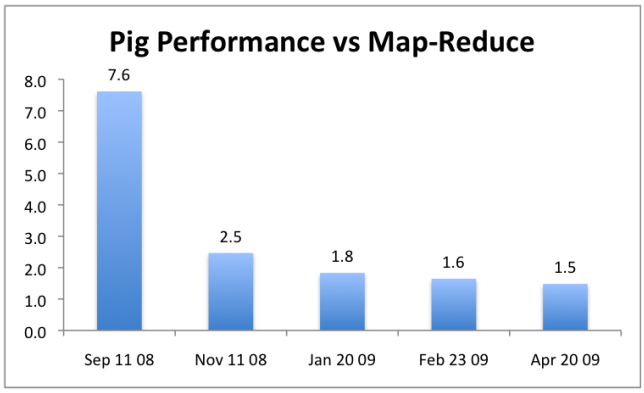

Figure 9: Pig Mix benchmark results, across several versions of the Pig software.

- September 11, 2008: Initial Apache open-source release.
- November 11, 2008: Enhanced type system, rewrote execution pipeline, enhanced use of combiner.
- January 20, 2009: Rework of buffering during data parsing, fragment-replicate join algorithm.
- February 23, 2009: Rework of partitioning function used in `ORDER BY` to ensure more balanced distribution of keys to reducers.
- April 20, 2009: Branching execution plans (Section 4.3.1).

The vertical axis of Figure 9 shows the ratio of the total running time for the twelve Pig programs (run in serial fashion), to the total running time for the corresponding Map-Reduce programs, on the same hardware. A value of 1.0 would represent parity between the Pig execution times and the raw Map-Reduce times, and our goal is to reach and then exceed[10] that mark. The current performance ratio of 1.5 represents a reasonable tradeoff point between execution time and code development/maintenance effort that many users have found acceptable, as our adoption numbers show.

8. ADOPTION

Pig has reached wide adoption in Yahoo. At the time of this writing (June 2009), roughly 60% of ad-hoc Hadoop jobs are submitted via Pig. Moreover, several production system projects have adopted Pig for their data processing pipelines, with 40% of production Hadoop jobs now coming through Pig. We expect adoption to increase further with many new users moving to Hadoop, and Pig becoming more robust and performant.

Projects that construct data processing pipelines for production systems are attracted to Pig by the fast development cycle, extensibility via custom code, protection against Hadoop changes, and ease of debugging. Ad-hoc data analysis projects such as user intent analysis are attracted to Pig because it is easy to learn, results in compact readable code, provides fast iteration through different versions of algorithms, and allows for easy collaboration among researchers.

We have also seen steady user growth outside of Yahoo. Uses of Pig we have head about include log processing and aggregation for data warehousing, building text indexes (a.k.a. inverted files), and training collaborative filtering models for image and video recommendation systems. Quoting from a user who implemented the PLSA E/M collaborative filtering algorithm [13] in Pig Latin:

> "The E/M algorithm was implemented in pig in 30-35 lines of pig-latin statements. Took a lot less compared to what it took in implementing the algorithm in Map-Reduce Java. Exactly that's the reason I wanted to try it out in Pig. It took 3-4 days for me to write it, starting from learning pig."

9. PROJECT EXPERIENCE

The development of Pig is different from projects worked on by many in the Pig team in that it is a collaboration between research and development engineering teams, and also that it is an open-source project.

Pig started as a research project at Yahoo. The project attracted a couple of high-profile customers that were able to accomplish real work using the system. This convinced management to convert the prototype into a production system and build a dedicated development engineering team. While transitioning a project from one team to another can be a tricky process, in this case it went well. The teams were able to clearly define their respective roles, establish good bi-directional communication, and gain mutual respect. Sharing a common background helped. Both teams spoke the same language of relational databases. While an asset for the transition, this proved to be a mild limitation for further work as neither team had a good understanding of non-database workloads such as building search indices or doing web crawls. During the transition period that lasted for about six months, both teams were actively involved in the process. The research team transitioned the project knowledge while still supporting the customers and the development team gradually took over the development and support. The research team is still actively involved in the project by building systems that rely on Pig, providing suggestions for future work, and promoting Pig both within Yahoo and outside.

Being open-source was a stated goal of the project from the beginning. In September 2007, Pig was accepted as a project by the Apache Incubator. This required Pig developers to learn the Apache methodology and to adapt to the open-source collaborative model of decision making. This resulted in the Pig committers being slow to integrate new users, review patches, and answer user and developer questions. As a result, Pig lost some potential developers and users in the process. With time and some frank and helpful feedback from users, the situation improved and we developed a small but fairly active user and development community. The project's first non-Yahoo committer joined after eight months. He contributed several significant modules to Pig including the initial implementation of the typechecker and a framework for comparing and validating execution

[10] In the future Pig may incorporate sophisticated database-style optimizations that a typical Map-Reduce programmer would not undertake due to lack of time or expertise. Additionally, Pig removes the requirement of hard-coded query execution strategies that raw Map-Reduce programming imposes, raising the possibility of adjusting query execution plans over time to suit shifting data and system properties.

plans. Our progress allowed the project to successfully graduate from the Apache Incubator and in October 2008 Pig became a Hadoop subproject. As noted in Section 8, we have an active user community that is doing interesting work with the system and providing mostly positive feedback. The development community, on the other hand, consists mostly of Yahoo engineers—a trend we have not been able to change so far. Pig is a very young project and we are hoping that being part of Hadoop will help us to attract new users as well as engineering talent.

We have found that being an open-source project has several benefits. First, being out in the open means that Pig is known both in the academic and development communities. Universities are teaching courses and sponsoring projects related to Pig. This helps our recruiting efforts as more potential developers are familiar with the project, think it is exciting, and want to be part of it.

A second, related benefit to being open is that we are able to collaborate across various companies. An example of this collaboration is a joint undertaking by Cloudera [7] and Yahoo to provide training for Pig. We have also been able to work with other Apache projects, such as Hive, sharing experiences and ideas.

Choosing Apache in particular as an open-source community has benefits. Apache provides firm guidance on an open-source development methodology (such as requiring code reviews, making decisions by consensus, and rigorous use of unit tests) that helped smooth our transition from a corporate development methodology to an open-source community based methodology. We feel that, as a team, we have taken the first steps in this conversion, though we need to become better at making sure all decisions are made in an open, community based manner.

10. FUTURE WORK

There are a number of areas under consideration for future development:

- **Query optimization.** We have begun work on a rule-based query optimizer that allows simple plan rearrangement optimizations (such as pushing inexpensive filters below joins) and very basic join selection. Research is being done on a more sophisticated optimizer that would enable both rule- and cost-based optimizations.
- **Non-Java UDFs.** We have begun work on removing the restriction that UDFs must be written in Java. We plan to write this in such a way that adding bindings for new languages will be minimal.
- **SQL interface.** SQL is a good fit for many scenarios, and is an established standard. We plan to add a SQL parser to Pig so that it will be able to accept programs in either Pig Latin or SQL.
- **Grouping and joining of pre-partitioned/sorted data.** Many data sets are kept partitioned and/or sorted in a way that can avoid having to shuffle data to perform grouping and joining. We are developing a metadata facility for Hadoop to keep track of such physical data attributes (as well as logical schema information), as a first step for Pig to be able to exploit physical data layouts.
- **Skew handling.** Parallel query processing algorithms like group-by and join are susceptible to skew, and we have experienced performance problems due to data skew with Pig at Yahoo. One particularly challenging scenario occurs when a join is performed, and a few of the join keys have a very large number of matching tuples, which must be handled by a single node in our current implementation, sometimes causing Pig to spill to disk. We are studying skew handling options, including ones that divide the responsibility for a single join key among multiple nodes [9].
- **Code generation.** Recall from Section 4.4 that Pig currently relies on a generic dataflow engine inside the Map and Reduce portions of a Hadoop job. We are considering the option of code generation [17], which might yield substantial performance benefits and enable Pig to close the gap with raw Map-Reduce.

Acknowledgements

Pig is an open-source project, and thus has numerous users and contributors from many organizations, all of whom we would like to thank for their contributions and feedback. In particular, we would like to acknowledge Arun Murthy for his work doing the initial implementation of Pig Streaming, Pi Song for contributing the type checker to Pig 0.2.0, Paolo D'alberto for his many suggestions on how to improve Pig's parsing and semantic checking, Daniel Dai for his work on `LIMIT` and his tireless support for running Pig on Windows, and Gunther Hagleitner and Richard Ding for their work on branching plans. We would also like to thank Prasenjit Mukherjee for sharing his experience implementing Hoffmann's PLSI algorithm in Pig Latin.

We are particularly grateful to our early-adopter user base at Yahoo, especially David "Ciemo" Ciemiewicz for his willingness to try early prototypes and provide a great deal of useful feedback, and for being a tireless advocate of Pig.

Lastly, we thank Sihem Amer-Yahia and Mike Carey for valuable feedback on an earlier draft of this paper.

11. REFERENCES

[1] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[2] Pig Mix Benchmark. http://wiki.apache.org/pig/PigMix.

[3] The Hive Project. http://hadoop.apache.org/hive/.

[4] The Pig Project. http://hadoop.apache.org/pig.

[5] K. Beyer, V. Ercegovac, and E. Shekita. Jaql: A JSON query language. http://www.jaql.org/.

[6] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proc. VLDB*, 2008.

[7] Cloudera. http://www.cloudera.com.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[9] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. VLDB*, 1992.

[10] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *Proc. ACM SIGMOD*, 1978.

[11] G. Graefe. Volcano – an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1), 1994.

[12] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and

H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1), 1997.

[13] T. Hofmann. Latent semantic models for collaborative filtering. *ACM Trans. Information Systems*, 22(1), 2004.

[14] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *Proc. ACM SIGMOD*, 2009.

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

[16] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 13(4):227–298, 2005.

[17] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using JVM. In *Proc. ICDE*, 2006.

[18] M. A. Shah, M. J. Franklin, S. Madden, and J. M. Hellerstein. Java support for data-intensive systems: Experiences building the telegraph dataflow system. *ACM SIGMOD Record*, 30(4), 2001.

[19] T. Strohman. *Efficient Processing of Complex Features for Information Retrieval*. PhD thesis, University of Massachusetts Amherst, 2007.

[20] Y. Yu, M. Isard, D. Fetterly, M. Badiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. OSDI*, 2008.