# PigOut: Making Multiple Hadoop Clusters Work Together

Kyungho Jeon, Sharath Chandrashekhara, Feng Shen, Shikhar Mehra, Oliver Kennedy, and Steven Y. Ko
University at Buffalo, the State Univeristy of New York
Email: {kyunghoj|sc296|fengshen|shikharm|okennedy|stevko}@buffalo.edu

***Abstract*—This paper presents PigOut, a system that enables federated data processing over multiple Hadoop clusters. Using PigOut, a user (such as a data analyst) can write a single script in a high-level language to efficiently use multiple Hadoop clusters. There is no need to manually write multiple scripts and coordinate the execution for different clusters.**

**PigOut accomplishes this by automatically partitioning a single, user-supplied script into multiple scripts that run on different clusters. Additionally, PigOut generates workflow descriptions to coordinate execution across clusters. In doing so, PigOut leverages existing tools built around Hadoop, avoiding extra effort required from users or administrators. For example, PigOut uses Pig Latin, a popular query language for Hadoop MapReduce, in a (virtually) unmodified form. Through our evaluation with PigMix, the standard benchmark for Pig, we demonstrate that PigOut's automatically-generated scripts and workflow definitions have comparable performance to manual, hand-tuned ones. We also report our experience with manually writing multiple scripts for a set of federated clusters, and compare the process with PigOut's automated approach.**

## I. INTRODUCTION

Hadoop [1] has gained tremendous traction from both academia and industry since its release, having been widely adopted by industry [2] and inspiring numerous publications in academic conferences and journals in the past few years. What started with MapReduce and HDFS, has now grown into a full ecosystem with many more components (e.g., Pig [3], [4], a dataflow language and execution system, and Oozie [5], a workflow scheduler), making Hadoop as a one-stop shop for all cluster computing needs.

Due in part to Hadoop's ease of use and management, even relatively small organizations or departments are spinning up Hadoop infrastructures for internal use. This proliferation of Hadoop instances is driving a need for federated (i.e., inter-cluster) functionality to support joint processing of distributed data sets stored in different clusters. Example scenarios are numerous—multi-organization coalitions, collaborations across different administrative domains within a single organization, or even collaboration between on-site clusters and infrastructures deployed on cloud-computing backends. Workflow systems like Oozie, and others [6], [7] provide support for inter-cluster workflows. However, fixed workflows are hard to adapt to changing cloud pricing schemes, resource availability, organizational policies, and international laws. In short, creating and managing multi-cluster workflows is still quite hard.

This paper proposes PigOut, a system that enables multiple Hadoop clusters to work together. PigOut addresses two main challenges critical for multi-cluster support—*ease of programming* and *ease of management*. Ease of programming means that a program spanning multiple clusters should be easy to write, relative to a single-cluster program. Ease of management means that administrators of a Hadoop cluster should not need to expend additional effort maintaining new software infrastructure or enforcing internal organizational policies.

We make careful design decisions to address these challenges in PigOut. First, we achieve ease of programming by leveraging Pig, a popular component in Hadoop ecosystem, that already provides ease of programming for a single cluster. Pig provides a high-level language and a runtime and internally generates a series of MapReduce jobs. Writing a Pig program (called a *script*) is easier [8] than writing pure MapReduce jobs, which makes it a popular choice among Hadoop users. PigOut essentially extends this benefit of Pig to federated clusters by automatically partitioning individual Pig scripts for execution on different clusters. From the user's point of view, there is little difference between writing a script for a single cluster and multiple clusters. Second, we achieve ease of management by designing PigOut as a front-end that interfaces with multiple Hadoop clusters, i.e., we make no change to Hadoop's existing infrastructure components. This is possible because PigOut partitions a script into complete stand-alone Pig scripts, each of which is independently runnable on a Hadoop cluster. From a cluster administrator's point of view, there is no difference between supporting and running a regular Pig script and a PigOut script. Thus, different administrative domains are free to manage their own cluster in any way they like. PigOut only needs a Hadoop user account in all the Hadoop clusters it uses on behalf of its user.

Specifically, we make the following contributions:

- **PigOut:** We present the design and implementation of PigOut, a system that enables inter-cluster scripting for Hadoop. We detail how PigOut partitions a single Pig script into multiple stand-alone scripts and manages the execution of the scripts. We also detail the optimization strategy PigOut uses when deciding how to distribute script execution.
- **Evaluation of PigOut:** We evaluate PigOut with Apache PigMix [9], a benchmark for Pig. Using PigMix, we compare the execution times and behavior between manually hand-tuned scripts and PigOut-generated scripts running on multiple clusters. As a result of our evaluation, we make two observations: First, hand-tuning PigMix scripts for multiple clusters can be very time-consuming and error-prone compared to using PigOut. Second, our performance results show that PigOut-generated scripts can provide performance comparable to that of hand-tuned scripts.

The rest of the paper is organized as follows. Section II

describes a motivating scenario. Sections III, IV, and V present an overview of PigOut, the details of PigOut, and our evaluation results, respectively. Section VI discusses related work and Section VII concludes.

## II. Motivation

In this section, we motivate the goals of PigOut by first showing how a developer would approach a multi-cluster data processing task using existing Hadoop components. We start with a simple scenario that we return to throughout the paper, and show how a developer might manually partition the script. Based on this discussion, we present our goals for PigOut.

### A. Scenario

Alice is a data analyst at Business-User Flow (BUFlow), a hypothetical global web analytics company with data centers in the US, Europe, and Asia. Each data center maintains a local Hadoop cluster, where it stores access logs (`page_views`). Per-user information (`users`) is stored at BUFlow's primary datacenter in the US. Alice performs a routine reporting task: estimating per-page revenue grouped by the market segment of each user. Nominally, this is an easy task: a 2-way join, followed by a group-by aggregate. However, the data Alice needs to analyze is scattered across multiple clusters. Alice needs federated data processing.

Naively, Alice could first pull page view logs from Europe and Asia to BUFlow's US datacenter and run her analysis there. However, this approach wastes significant resources: transferring terabytes of data (or more) is slow and consumes bandwidth. Moreover, the compute resources of the Europe and Asia datacenters are not used at all by this approach. A more efficient approach would be to spread the processing over all clusters involved, i.e., writing tasks to run on different clusters and managing the execution of the tasks regarding the dependency and data transfer among them. Next, we discuss how one could do this natively using Pig.

### B. Manual Approach

In order to write tasks to run on different clusters and manage the execution, Alice could benefit from leveraging two of the components in Hadoop—Pig and Oozie. Pig is useful in writing per-cluster tasks, while Oozie is useful in controlling their execution. We introduce these two components first and show how to use them in the running example.

*1) Pig:* Pig is a query processing engine built atop MapReduce. A pig user writes a Pig script in Pig Latin [4], a dataflow programming language. The user then submits the Pig script to Pig and Pig compiles the script into MapReduce jobs. Pig manages the execution of these jobs as well.

In Pig, users can specify a sequence of data transformation steps, as they do in procedural programming languages. Each step is a simple and high-level transformational command that resembles relational algebra primitives, such as `GROUP`, `FILTER`, etc.

As a concrete example, consider the scenario described in Section II-A. Alice wants to estimate per-page revenue grouped by the market segment of each user. So she first pulls all the data from the Europe and Asia data centers to the US data center. She then writes a Pig script to process all the data together as follows (Table I shows a brief summary of the commands used in the script):

TABLE I. Pig Latin Command Examples.

| Command | Description |
|---|---|
| LOAD | Specify an input |
| STORE | Specify an output |
| UNION | Concatenate two inputs |
| FILTER | Select which records to retain |
| FOREACH | Apply a set of expressions to every record |
| JOIN | Select records from one input to put together with records from another input |
| GROUP | Collect records of an input based on a key |
| COGROUP | Collect records of inputs based on a key |

```
 1: us_pv = LOAD 'hdfs://us.buflow.com:8093/us_in';
 2: eu_pv = LOAD 'hdfs://us.buflow.com:8093/eu_in';
 3: as_pv = LOAD 'hdfs://us.buflow.com:8093/as_in';
 4: page_views = UNION us_pv, eu_pv, as_pv;
 5: filtered = FILTER page_views BY timespent > 1;
 6: projected = FOREACH filtered
                GENERATE user, revenue;
 7: users = LOAD 'hdfs://us.buflow.com:8093/users';
 8: p_users = FOREACH users GENERATE name, segment;
 9: joined = JOIN projected BY user, p_users BY name;
10: grouped = GROUP joined BY segment;
11: result = FOREACH grouped
             GENERATE group, SUM(grouped.revenue);
12: STORE result into 'hdfs://us.buflow.com:8093/out';
```

The first three commands load the input data sets. Command `4` combines all page views from all input data sets. Command `5` filters page views by time. Command `6` projects two relevant columns (`user` and `revenue`) from the combined data set. Command `7` loads the user table. Command `8` projects two relevant columns (`name` and `segment`) from the `users` table. Commands `9` and `10` combine the two newly-generated tables and group the data by segment. Command `11` computes the revenue per segment and the last command stores the result.

This single-cluster Pig script is not efficient since the resources in other data centers are wasted. Using Oozie described below, Alice could use the resources in all three data centers more efficiently.

*2) Oozie:* Oozie is a Hadoop workflow engine that can control the execution of MapReduce jobs, Pig scripts, shell scripts, and a few other types of execution units. A workflow is described by a DAG (Directed Acyclic Graph). Each node in the DAG is an *action*, which can be an execution of a script. Each edge in the DAG is a *dependency*; a task is blocked until each of its in-edge tasks are complete. Workflow descriptions are written for Oozie with an XML based process definition language called hPDL.

As a concrete example, consider the Alice's scenario again. Instead of processing all the data in one site, Alice could make her reporting query more efficient; she could separate the projection command into a separate script which she runs independently on each cluster as follows:

```
1: page_views = LOAD '$INPUT';
2: filtered = FILTER page_views BY timespent > 1;
3: projected = FOREACH filtered
               GENERATE user, revenue;
4: STORE projected into '$OUTPUT';
```

The resulting `projected` data sets are far smaller, and can be transferred to BUFlow's primary datacenter more

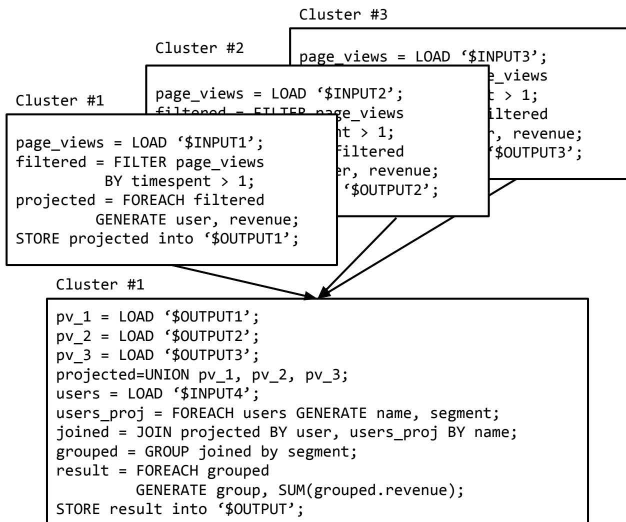



Fig. 1. Graphical representation of Pig Latin scripts in an Oozie Workflow. Variables prefixed with a '$' are parameters provided through Oozie.

efficiently. Using Oozie, Alice could coordinate the execution of these scripts through a workflow such as the one illustrated visually in Figure 1.

Although this multi-cluster workflow is far more efficient, it comes at the cost of code complexity. Alice now has to maintain four scripts across three clusters, on top of a workflow description for Oozie. Furthermore, changes in company policy, cluster resource availability, or cloud-provider pricing models can lead to further bifurcation of a once-simple codebase.

### C. *PigOut Features*

With PigOut, we aim to provide the benefit of the manual approach described above without the complexity of writing multiple scripts and managing the execution among them using Oozie. PigOut provides the following three features:

- **Automation:** Manually writing multiple scripts and maintaining them over time is difficult and error-prone. PigOut allows developers to write a single script that accesses multiple data sources spread over different clusters, which it then automatically partitions. The resulting stand-alone Pig scripts are then deployed to their respective clusters through automatically generated Oozie workflow descriptions.
- **Performance:** When partitioning a single script, PigOut aims to provide performance comparable to that of manually-written, hand-tuned scripts.
- **Pig Latin Extension:** As an additional feature, PigOut extends the syntax of Pig Latin with new commands to simplify the development of federated scripts, and to remain in keeping with Pig's philosophy of predictable script execution plans. These commands are provided to support direct manipulation of PigOut's execution strategy and to automate frequently occurring design patterns. It is not necessary for developers to use this new functionality unless desired.

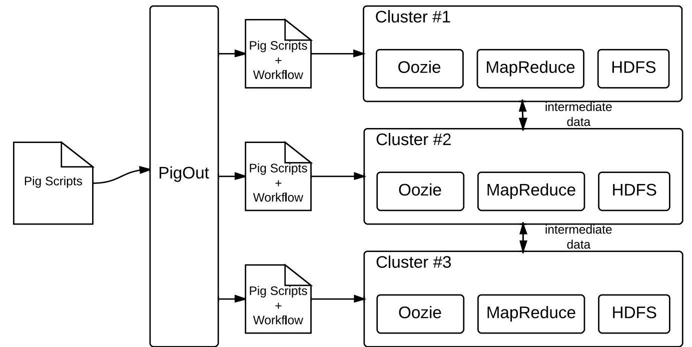



Fig. 2. High-level view of PigOut: A data analyst utilizes multiple clusters via PigOut.

In the rest of the paper, we show how we provide these features in PigOut, starting with its design and implementation.

## III. PigOut Overview

We first overview PigOut from the user's point of view using our running example of Alice the BUFlow employee (Section II-A). We then present an overview of the internals of PigOut. Later in Section IV, we discuss the design summarized here in detail.

### A. *PigOut Scripting Overview*

Figure 2 depicts the high-level architecture of PigOut. PigOut is essentially a front-end that a user (such as a data analyst) uses to interact with multiple clusters. From the user's point of view, there is little difference between writing a script for a single cluster and multiple clusters.

More concretely, consider our running example from Section II-A. Using PigOut, Alice could write the following script to use three Hadoop clusters in her US, Europe, and Asia data centers:

```
 1: us_pv = LOAD 'hdfs://us.buflow.com:8093/input';
 2: eu_pv = LOAD 'hdfs://eu.buflow.com:8093/input';
 3: as_pv = LOAD 'hdfs://as.buflow.com:8093/input';
 4: page_views = UNION us_pv, eu_pv, as_pv;
 5: filtered = FILTER page_views BY timespent > 1;
 6: projected = FOREACH filtered
              GENERATE user, revenue;
 7: users = LOAD 'hdfs://us.buflow.com:8093/users';
 8: p_users = FOREACH users GENERATE name, segment;
 9: joined = JOIN projected BY user, p_users BY name;
10: grouped = GROUP joined BY segment;
11: result = FOREACH grouped
             GENERATE group, SUM(grouped.revenue)
12: STORE result into 'hdfs://us.buflow.com:8093/out';
```

As shown, the only difference between this (PigOut) script and the (Pig) script presented in Section II-B for a manual approach is `LOAD`; in the first three `LOAD` commands, Alice specifies different cluster URIs as input sources. The rest of the script is exactly the same.

From this single script that Alice writes, PigOut automatically generates multiple scripts and Oozie workflow descriptions that makes effective use of all three BUFlow clusters. PigOut extends Pig's built-in `LOAD` and `STORE` commands to

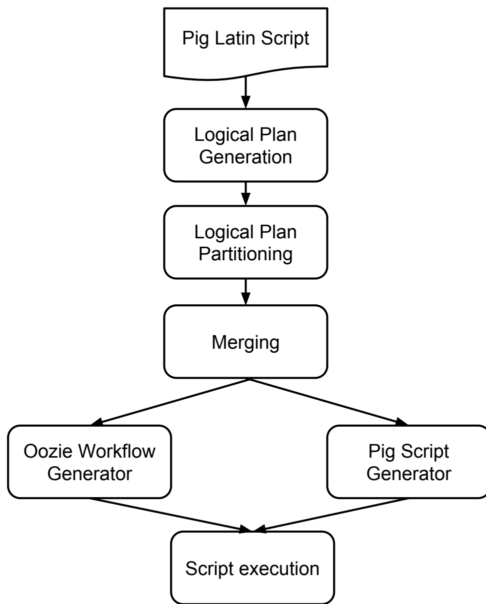



Fig. 3. PigOut System Flow

recognize full HDFS URIs[1], rather than just local file paths. This is the *only* modification required to turn single-cluster pig scripts into PigOut scripts. We will return to the further (optional) functionality that we add to PigOut in Section IV-E. `LOAD` and `STORE` commands are executed on their local clusters. PigOut partitions the remainder of Alice's script across the three clusters.

### *B. PigOut System Overview*

Figure 3 shows how PigOut generates multiple scripts and Oozie workflow descriptions from a single script. This is done in five phases. The first phase is *logical plan generation*, where PigOut parses the script and builds an internal representation of it (i.e., the logical plan described in Section IV-A), a DAG where each vertex represents a data transformation command and each edge defines a data flow between two commands. Prior to proceeding with its multi-cluster transformations, PigOut first applies Pig's native logical plan optimizations to simplify the logical plan [4].

The second phase is *logical plan partitioning*. PigOut enumerates participating clusters (sites) based on URIs provided in the script's `LOAD` and `STORE` commands, and annotates each command in the logical plan with a list of candidate sites where the command could be executed. Once annotated in this way, commands in the plan are chunked into subplans using a simple heuristic described in Section IV-B2. Each subplan is an isolated set of Pig Latin instructions that, once converted to a script, is runnable in a single Hadoop cluster.

The third phase is *merging*, where PigOut further optimizes the chunked logical plan. In this phase, PigOut merges subplans in order to reduce the number of MapReduce jobs created, and maximize the potential for single-site optimization at runtime. We detail this stage in Section IV-C.

The fourth phase is *script and workflow generation*, where PigOut uses the partitioned logical subplans to generate a Pig script for each subplan, an Oozie workflow description coordinating execution within a single cluster, and an Oozie coordination plan that captures cross-cluster workflow dependencies. PigOut also generates data transfer scripts that move intermediate data among all the clusters involved. This phase is a purely mechanical translation from our internal graph representation back to scripts and workflow descriptions, and is described in Section IV-D.

The last phase is *script execution*, where PigOut submits the generated workflow scripts to Oozie and Pig instances across different clusters. When the submissions are done, the off-the-shelf Hadoop components (i.e., Oozie, Pig, HDFS, and MapReduce) take over and execute the scripts according to the workflow submitted.

## IV. PigOut Design and Implementation

In this section, we describe six phases that PigOut uses to generate partitioned scripts and workflow descriptions.

### *A. Logical Plan Generation*

The first phase of PigOut involves two steps—initial logical plan generation and logical plan optimization.

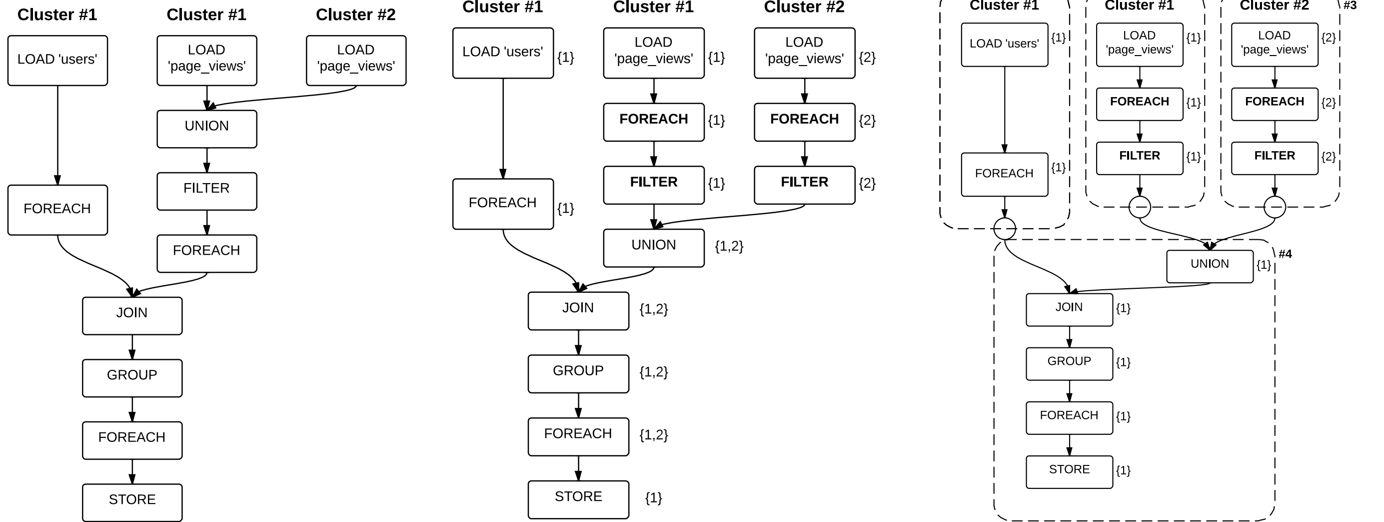
*1) Initial Logical Plan Generation:* When a user-provided script is submitted for execution, PigOut first parses the script and transforms it into an internal representation, i.e., a logical plan. As mentioned earlier, a logical plan is a DAG representation of a script. Figure 4(a) shows an example logical plan from the script we discuss in Section II (for the simplicity of presentation, we only show two clusters, not three as in the scenario). This initial logical plan is optimized and partitioned into subplans for multi-cluster execution.

*2) Logical Plan Optimization:* Once a script is converted to an initial logical plan, PigOut applies rule-based optimization techniques. In doing so, we adapt some of the existing optimization rules from the original Pig Latin compiler to our multi-cluster context. The main purpose of this initial logical plan optimization is to reduce the size of intermediate data that flows from one command to the next. This is an important step of optimization since less data means less execution time in Pig and MapReduce. For example, consider our scenario in Section II-A. Since only the `user` and `revenue` columns are used throughout the script from the data sets, the rest of the columns can be filtered out right after each `LOAD`.

More precisely, there are two principles in our optimization—(i) we filter out unnecessary data as early as possible, and (ii) we generate additional data as late as possible. To accomplish this, we adapt the following optimization rules from the original Pig Latin compiler:

- **PushUpFilter**: This rule pushes `FILTER` up as early as possible to reduce the size of intermediate data.
- **PushDownForEachFlatten**: This rule pushes down `FLATTEN`, which (roughly) generates a list of cross

[1]Currently, we assume that when there is an HDFS instance running in a cluster, there is a corresponding Pig and MapReduce instances running in the same cluster. This is indeed the case for most of the Hadoop installations; however, we can relax this assumption by having additional configuration parameters that provide Pig and MapReduce locations.




(a) The logical plan of the script presented in Section II for finding estimated revenues.

(b) The logical plan after *ColumnPruner* and *PushFilter* optimization rules applied and site assignment annotations are done. Operators added and/or moved by optimization rules are noted by **bold** letters.

(c) The logical plan after partitioning. Small circles represent breakers.

Fig. 4. PigOut's logical plan processing. The third cluster is omitted for the simplicity of presentation.

products between data items. By pushing this down to a later stage in the plan, we can keep the size of intermediate data in the flow low.

- **ColumnPruner**: Removes columns that are not used or no longer needed.
- **MapKeyPruner**: Removes map keys that are not used, so reduces the size of the record.

Figure 4(b) shows the logical plan after ColumnPruner and PushUpFilter rules are applied (in addition to the site assignments, which we explain in Section IV-B1). As shown, the logical plan now filters out unused columns right after each `LOAD` in order to reduce the amount of intermediate data.

### B. Logical Plan Partitioning

Once PigOut finishes the logical plan generation phase, it partitions the plan into subplans. This phase involves two steps—site assignment and partitioning.

*1) Site Assignment:* The first step for logical plan partitioning is site assignment, where PigOut chooses one execution site (i.e., cluster) for each vertex (i.e., command). This site assignment is done with two sub-steps. In the first sub-step, PigOut annotates each vertex with a list of potential execution sites. In the second sub-step, PigOut chooses the best site for each command based on our cost calculation.

PigOut uses the following three rules in the first sub-step:

- **Rule 1:** `LOAD` and `STORE` get their site assignment from the URI specified in the original script.
- **Rule 2:** The site assigned for a `LOAD` command gets propagated *down* all the way through the logical plan until a `STORE` is encountered. The encountered `STORE` does *not* inherit the site assignment.
- **Rule 3:** The site assigned for a `STORE` command gets propagated *up* all the way through the logical plan until it encounters a command that combines multiple data flows, such as `UNION` and `JOIN` in Figure 4(a). The encountered command *does* inherit the site assignment.

These three rules naturally capture potential execution sites for each command based on where to load input data and where to store output data.

Figure 4(b) illustrates the site assignments for our Alice the BUFlow employee example. Each `LOAD` and `STORE` command has a URI given to it; applying rule #1, PigOut assigns one site to each `LOAD` and `STORE` accordingly. PigOut then applies rule #2 and propagates the site assignment down the logical plan from each `LOAD`. As shown, this results in multiple potential execution sites for some of the commands. Lastly, PigOut applies rule #3 and propagates the site assignment for `STORE` up the logical plan. This propagation stops at `JOIN` as it is a command that combines multiple data flows.

Once the candidate sites for each command are determined, PigOut proceeds to the second sub-step of site assignment, where it chooses the best execution site for each command. In doing so, the primary goal is to minimize overall execution time. Pruning techniques for enumeration-based optimization strategies (e.g., left-deep joins, gradient descent, and A* search) are well studied. For evaluation purposes, we adopt a simple approach: PigOut enumerates all possible allocations and selects the plan with the lowest cost. In the spirit of the textbook System R cost estimation model [10], we use rough arity estimates as an estimator for execution time, as discussed below. PigOut does not preclude more sophisticated cost estimation and optimization strategies. However, our evaluation shows that the heuristics we use are practical and provide comparable performance to that of a manual, hand-

TABLE II. SELECTIVITIES FOR COMMANDS

| Command | Selectivity Factor |
|---|---|
| `JOIN` | Sum of the predecessors |
| `UNION` | Sum of the predecessors |
| `GROUP` | 0.24 |
| `FILTER` | 0.5 |
| `DISTINCT` | 0.25 |
| User-defined Func | 0.10 |

tuned approach.

PigOut uses arity estimates as a basis for comparing different execution plans; in the common case, a ballpark estimate is sufficient for this purpose [10]. With this in mind, PigOut's arity estimator operates as follows: For cross-site data transfer cost estimation, we first estimate how much (intermediate) output a command would generate. We assign a *selectivity* for this. A selectivity is an input-to-output ratio estimation for each command. For example, $0.5$ for `FILTER` means that PigOut estimates the output size of `FILTER` would be ($0.7 \times$ the input size). Table II shows the selectivities for various commands we use in this paper. The number can be adjusted according to data characteristics and workloads.

With a selectivity assigned for each command, we can estimate how much intermediate data would flow for each edge in the logical plan.

*2) Partitioning:* Once PigOut determines the final site assignment for each command, it identifies all possible break points to partition the logical plan. PigOut accomplishes this by depth-first search. It starts from each of the top-most `LOAD` commands and traverses the logical plan. For each vertex it visits, it checks whether or not the successor of the vertex is assigned the same site. If that is not the case, PigOut inserts a *breaker* vertex, which we explain below. Otherwise, PigOut continues its depth-first traversal.

A *breaker* is a special type of vertex that marks a potential partition point. An example is depicted in Figure 4(c). Each breaker vertex indicates that the commands before and after the breaker can run on two different clusters. This also means that transferring intermediate data from one cluster to another cluster will be required at the breaker. Thus, PigOut later converts each of these breakers to a sequence of `STORE` in one cluster, data transfer from one cluster to the other, and `LOAD` in the other cluster, appropriately.

### C. Merging

When PigOut finishes the partitioning phase, the logical plan is divided into multiple subplans as indicated by breakers and each subplan is assigned to one site for execution. At this stage, it is possible to convert each subplan to a Pig script and execute them. However, doing it right after partitioning runs the risk of generating too many scripts unnecessarily.

For correctness, generating and running many scripts does not have any issue. However, performance might suffer as there is a start-up cost associated with each script execution, e.g., script submission cost, Pig's script compile cost, MapReduce job startup cost, etc., that all add up to be non-negligible. Thus, it is important to reduce the number of scripts without sacrificing any other aspects of performance.

Consequently, PigOut has an additional merging phase, where it combines multiple subplans if possible. This requires a balance between parallelism and the number of scripts generated. For example, we could combine the three subplans assigned to cluster #1 (subplans #1, #2, and #4) in Figure 4(c). This will reduce the number of scripts for cluster #1 from three to one, potentially saving much of the additional script-running overhead. However, this reduces opportunities for parallelism. As shown in Figure 4(c), the first three subplans (subplans #1, #2, and #3) are independent of each other. This means that we can run them in parallel in clusters #1 and #2. However, if we combined all the subplans for cluster #1 (subplans #1, #2, and #4), we would lose the parallelism. This is because we can start executing a script only when all the input data is available. The script generated from the combined subplan for cluster #1 would need to wait until cluster #2 finishes its processing completely.

PigOut strikes the balance between parallelism and the number of scripts by only combining subplans that have the exact same dependency with respect to other subplans. For example, subplans #1 and #2 in Figure 4(c) have the exact same dependency, which is to feed the output to subplan #4. Thus, PigOut combines them into a single subplan. We would not lose much parallelism since they can still run alongside subplan #3. Note that we do not lose the parallelism between subplans #1 and #2 even if we combine them into a single script. This is because the original Pig identifies parallel tasks and actually runs them in parallel. Thus, PigOut relies on Pig to run combined subplans in parallel.

### D. Script and Workflow Generation

Once the initial logical plan is partitioned and merged into subplans, PigOut converts them into Pig scripts. This phase is purely mechanical and just involves translation of logical subplans to scripts and workflow descriptions.

To generate the scripts, PigOut scans a plan and creates a semantically equivalent command from each vertex. As mentioned earlier, PigOut also transforms each breaker into three steps, a `STORE`, a data transfer, and a `LOAD`. In addition, PigOut generates two workflow descriptions for each subplan, one for single cluster execution, and the other for cross-cluster dependency resolution.

PigOut transfers data between clusters by `scp`. Transferring large data sets efficiently is well studied, and beyond the scope of this paper. Hence, we chose `scp`, an off-the-shelf tool, that is readily available. By leveraging off-the-shelf data transfer tools, administrators do not need to worry about supporting yet another data transfer tool. Moreover, Hadoop clusters typically sit behind a firewall, which makes it difficult to use custom data transfer tools that might require additional configuration. For example, Hadoop provides `distcp` as a data copy tool across machines, but it incurs all-to-all communication between different Hadoop cluster nodes. This requires much manual firewall configuration across different administrative domains, just to support `distcp`. PigOut's transfer process is modular, allowing us to easily leverage other off-the-shelf tools such as `rsync` easily.

### *E. PigOut Extension to Pig Latin*

Even though PigOut automatically generates distributed data flows from Pig scripts, PigOut extends the syntax minimally in order to provide a way for users to describe data flows explicitly. There are two additional commands PigOut adds to Pig Latin. First, `LOADX` specifies explicitly that the input data set will be shipped to another site without processed at or near the data source. Second, `STOREX` specifies that a data set will be shipped to the destination, after the data set is generated at a site. Therefore, `LOADX` is translated into a data transfer script followed by a `LOAD` command. Correspondingly, `STOREX` is translated into a `STORE` command and a data transfer script.

## V. EVALUATION

We evaluate PigOut's performance using PigMix, a benchmark for Pig [9]. PigMix contains 17 Pig scripts designed to stress-test and benchmark various Pig commands. We first present background necessary to understand our results, and later present the results.

### *A. Preliminary*

Since PigMix scripts are written for a single cluster setting, we adapt all 17 scripts for a two-cluster setting used in our evaluation. We first present this two-cluster setting. We then describe how we adapt PigMix scripts and manually partition them in order to compare manual approaches to PigOut.

*1) Evaluation Setting:* We use two heterogeneous Hadoop clusters in our experiments. One cluster (Cluster #1) consists of 5 machines, each with 4 CPU cores (one Intel Xeon X3430 2.40GHz), 16 GB of RAM, and one 160 GB disk for OS/applications and one 1 TB disk for data. The other cluster (Cluster #2) consists of 16 machines, each with 4 CPU cores (one Intel Xeon E5-2403 1.80 GHz), 48 GB of RAM, four 1 TB disks. A node in Cluster #1 communicates to each other through 1 Gb/s Ethernet, but Cluster #2 has 10Gb/s networks for internal communication. The two clusters communicate over 1 Gb/s link. Each cluster runs on Ubuntu Linux 10.04 (Cluster #1) and Red Hat Enterprise Linux 6 (Cluster #2). Both clusters run Hadoop MapReduce 1.0.4 and Apache Oozie 4.0.1.

Although two clusters are located within the general domain of our university, they are separated from each other since each sits behind its own firewall. In addition, they are managed separately; Cluster #2 is a shared Hadoop cluster for our department, and Cluster #1 is a cluster managed by our research group. Thus, this mimics a real-world setting where two Hadoop clusters belong to different administrative domains.

*2) PigMix Adaptation:* PigMix scripts are written for a single-cluster setting, and we need to adapt them for our two-cluster setting. In order to understand how we adapt PigMix scripts, consider our running example in Section II-A, which in fact represents the base scenario for all PigMix scripts as well as our adaptation process. In our running example, there are two tables processed together, `page_views` and `users`. Similarly, every PigMix script processes either a single `page_views` table or both `page_views` and `users` tables together. The only difference from script to script is which columns from these tables are processed and how.

Starting from these scripts, we make the following three changes. First, we spread `page_views` across two clusters and place `users` in one cluster. Second, for each `LOAD` command that reads `page_views` in a PigMix script, we replace it with two `LOAD` commands, one reading from one cluster, the other from the other cluster. Third, for each `STORE` command, we provide the URI of the cluster that has the `users` table. Since other parts are untouched, the resulting scripts are nearly identical to the originals except `LOAD` and `STORE` commands. We have put all our adapted PigMix scripts on our website at: `http://pigout.cse.buffalo.edu`.

In our experiments, we directly feed these adapted PigMix scripts to PigOut, so that PigOut can automatically process them. In addition, we manually partition each script and write workflow descriptions for comparison, as described next.

*3) Manual Partitioning:* In order to establish the baseline for comparison, we have manually partitioned all 17 PigMix scripts and compared their performance to that of PigOut's automatically partitioned scripts. However, since manual partitioning for a script is essentially hand-optimization, there is a very high degree of freedom available. This makes it difficult to quantify how much optimization has gone into a script for manual partitioning. For this reason, we also discuss our strategy for manual partitioning below. In addition, we have put all our manually partitioned scripts on our website for further perusal at: `http://pigout.cse.buffalo.edu`.

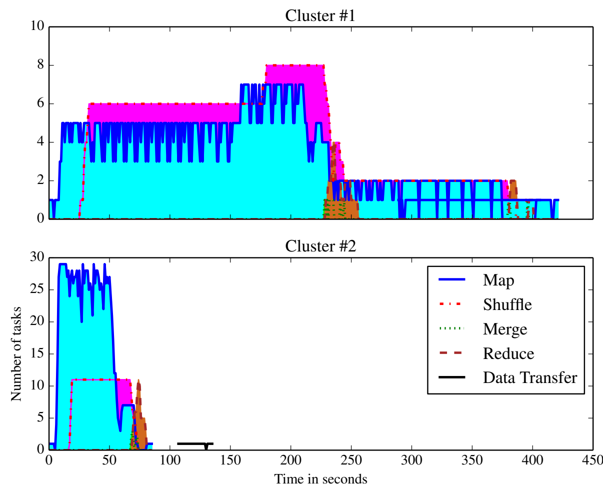
Our manual partitioning involves two passes. First, we perform mostly-mechanical manual partitioning for each adapted PigMix script. This process resembles what Alice does in our running example—(i) we first write a script for each cluster that reads the cluster's own data set (`page_view`) and projects columns to be processed, and (ii) we write another script that processes the projected data sets together.

After this, we proceed to our second pass where we analyze the script at hand to perform deeper optimization. Mainly, we look for opportunities for distributed processing; for example, if a total summation needs to be done over two data sets, then we perform a partial summation of each data set in each cluster first, then ship the results to one cluster for the final summation.

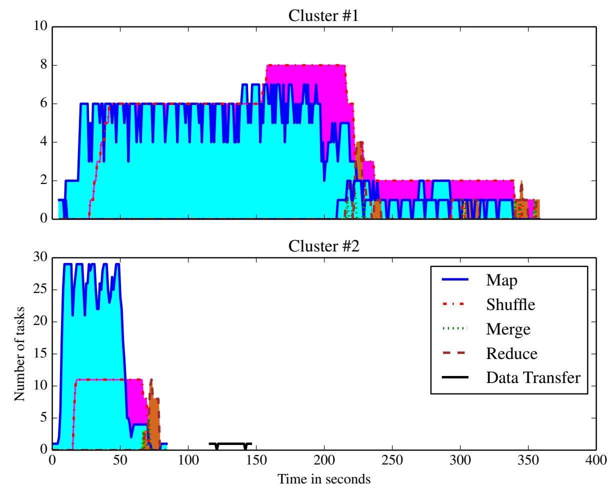
Once we are done with our hand-optimization, we then generate Oozie workflow descriptions for all clusters that coordinate the overall execution of the scripts.

### *B. Discussion: Manual Partitioning Experience*

While manually partitioning adapted PigMix scripts, we have found out that the manual approach is time-consuming and error-prone. At first, it took 3 to 4 hours for us to correctly craft a series of workflows over federated clusters from a single script. Later, we were able to use some of the already-written scripts and workflow descriptions as templates, since several PigMix scripts are structured in a similar way. Even then, it still took several dozen minutes to correctly partition a single script and generate Oozie workflows. We expect that manually partitioning scripts with a new structure (e.g., adding a new cluster, or adapting to a policy change) would again require several hours of effort.

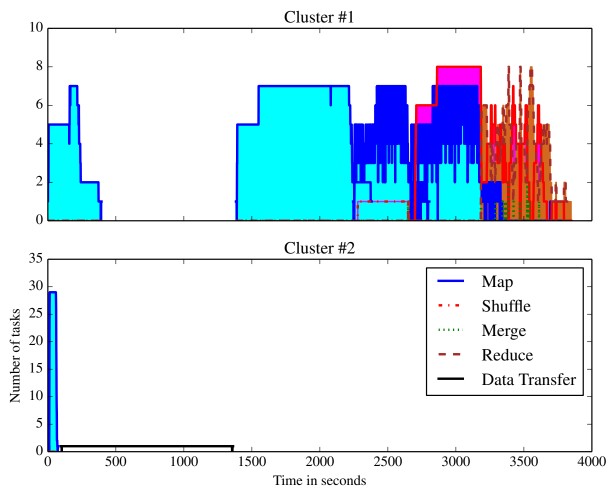


(a) L3: Manually partitioned

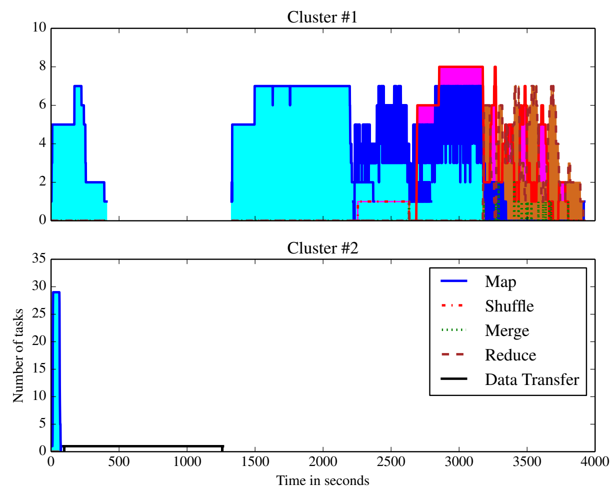


(b) L3: PigOut partitioned

Fig. 5. L3 execution behavior



(a) L9: Manually partitioned



(b) L9: PigOut partitioned

Fig. 6. L9 execution behavior

There were two main sources of difficulty in manual partitioning. The initial difficulty was that we had to reason about the nature of data and how the data was spread over multiple clusters, in order to come up with a good optimization strategy. A simple example is a filter operation; without understanding the data, there is no way to tell how much a filter operation would reduce the data set.

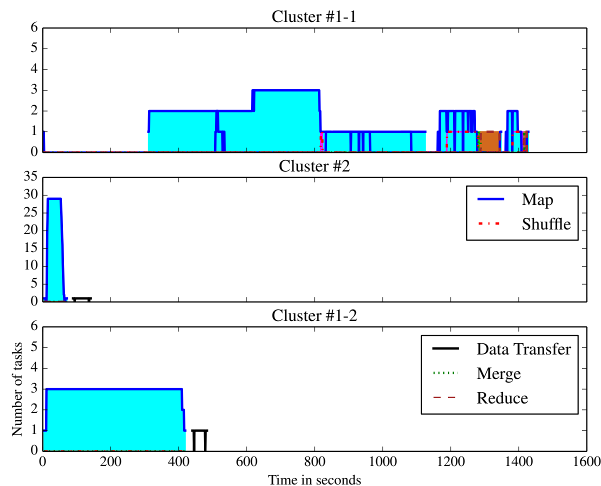
The other main source of difficulty was that the debug/rerun cycle in manual partitioning process involved an extremely heavy-weight trial and error process that spans multiple clusters. Even on a single cluster, MapReduce startup costs can lead to multi-minute trial runs. This phenomenon is amplified further still when using multiple scripts and workflow descriptions. Not only is the execution latency higher, but errors can trigger a need for *coordinated fixes across multiple scripts*, leading to further errors and consequent debug/rerun cycles.

Using PigOut had a clear benefit over this, since there was no manual labor involved. Once a PigMix script was adapted, we were able to run it right away through PigOut. For example, we were able to get the results of all adapted PigMix scripts within a matter of a few hours with PigOut.

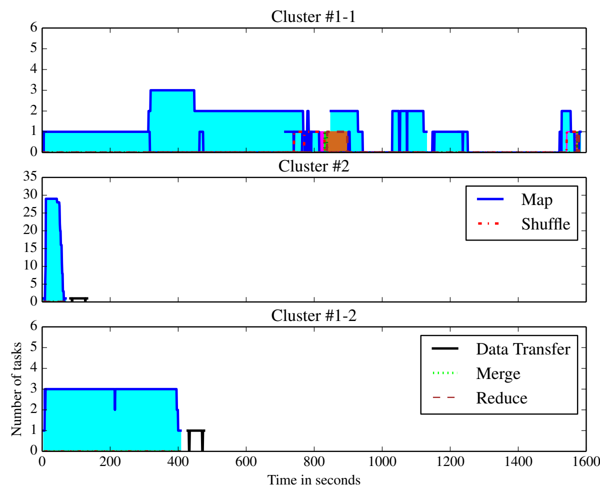
## C. Performance Comparison

The main purpose of our performance evaluation is to demonstrate that PigOut's automated approach provides comparable run-time performance to that of a manual approach. In all our experiments, each of the two clusters contains 10GB of `page_views` and Cluster #1 contains 128MB of `users` as well. With these input data sets, processing of a script mostly finishes within several minutes. However, there are some scripts that take longer as well, e.g., several tens of minutes. We believe that this skewed workload with more small jobs reflects the reality—a recent measurement study [11] analyzed seven Hadoop cluster traces from Facebook and Cloudera and reports that MapReduce jobs vary much in size but mostly small. For example, over 90% of all the jobs in the traces took less than several minutes.

Table III compares the execution times for all the scripts.




(a) L3: Manually partitioned with 3 clusters




(b) L3: PigOut partitioned with 3 clusters

Fig. 7. L3 execution behavior with 3 Hadoop clusters

TABLE III. EXECUTION TIME COMPARISON IN SECONDS.

| Script | Manual | PigOut | (PigOut − Manual) | PigOut Compile |
|---|---|---|---|---|
| L1 | 1999 | 2048 | 49 | 5.491 |
| L2 | 303 | 380 | 77 | 5.160 |
| L3 | 568 | 532 | -36 | 5.489 |
| L4 | 406 | 373 | -33 | 4.661 |
| L5 | 429 | 415 | -14 | 4.806 |
| L6 | 481 | 437 | -44 | 4.712 |
| L7 | 488 | 436 | -52 | 5.094 |
| L8 | 1205 | 1558 | 358 | 5.091 |
| L9 | 3849 | 3922 | 73 | 4.670 |
| L10 | 4034 | 4024 | -10 | 5.096 |
| L11 | 663 | 590 | -73 | 5.263 |
| L12 | 516 | 480 | -36 | 6.355 |
| L13 | 426 | 481 | 55 | 4.706 |
| L14 | 624 | 605 | -19 | 5.404 |
| L15 | 596 | 528 | -58 | 5.780 |
| L16 | 431 | 422 | -9 | 5.094 |
| L17 | 4542 | 4679 | 137 | 5.389 |

Figures 5 and 6 provide an in-depth look into the run-time behavior of each case. In all our results, we follow PigMix's naming scheme, even though we have adapted all the scripts. In addition to execution times, "PigOut Compile" shows how long it takes to generate partitioned scripts and Oozie descriptions for PigOut.

From Table III, we make two observations. First, the execution times are similar, i.e., the difference is typically within a couple of minutes. The reason is generally because PigOut generates a similar set of scripts and workflow descriptions as those we generate by hand. Second, PigOut can be either faster or slower than the manual approach, depending on the script. The largest difference is for L8, where PigOut takes around 350 seconds longer than the manual. Part of this difference is due to Oozie; before executing a script, Oozie verifies that all its input data is available by polling HDFS for the existence of the input data every minute. Due to this polling interval, there can be a delay in executing a script.

We use L3 and L9 as representatives to highlight some of the execution behavior of PigOut. In Figure 5, we can observe that both the manual approach and PigOut show execution times close to each other for map tasks, reduce tasks, and data transfers. This is due to the fact that both approaches generate similar scripts for execution. In Figure 6, we can observe that the beginning part of the execution only has Map tasks. In fact, these map tasks do not do any processing except reading of the input data sets. PigOut could replace these map tasks with a shell script that simply reads the input data from HDFS to optimize the performance further.

In Figure 7, we further examine the behavior of PigOut by running L3 over 3 clusters. For this experiment, we split Cluster #1 into two clusters, Cluster #1-1 (3 nodes with one master node and two worker nodes) and Cluster #1-2 (2 nodes with one master/worker node and one worker node). Similar to the experiments discussed previously, each cluster has a 10 GB `page_view` table, and Cluster #1-1 has 180 MB of `users` table. As shown, the execution of PigOut behaves in a similar way to the manually partitioned one. Only notable difference is the first 400 seconds in Cluster #1-1, where there is some extra processing done by PigOut. The reason for this behavior is that PigOut tries to maximize the opportunities of parallelism as discussed in Section IV-C; thus, we can observe in Figure 7 that PigOut utilizes all three clusters in the beginning intead of letting one cluster stay idle for a few hundred seconds. However, it does not translate into any performance again in the case shown in Figure 7. The reason is the same as before; due to the Oozie polling interval for checking data availability, there can be a delay in between executing jobs.

## VI. RELATED WORK

The idea of processing data sets over distributed sites is not new. Federated database systems (FDBS) have been developed to use and manage a set of autonomous and possibly heterogeneous but cooperating database systems [12]. In scientific computing, there are many workflow systems in the literature such as Pegasus [6] and Kepler [7] that allow users to express multi-step computational/data tasks. These existing systems evidence that federated data processing is necessary in many domains. PigOut applies the same concept to Hadoop in order to offer the same benefits in Hadoop clusters.

The domain of scientific computing has also widely adapted Hadoop MapReduce. As a result, several systems

integrate Hadoop into existing workflow systems, albeit limited to a single Hadoop instance setting. Wang et al. [13] presents how to integrate Hadoop MapReduce with Kepler [7], a popular scientific workflow system. Recent work along the same line includes G-Hadoop [14] and Kondikoppa et al. [15].

Practitioners have also discussed the idea of a hybrid cloud, which deploys a computational infrastructure on private, on-premise systems as well as public cloud services, due to concerns over data and computation privacy. HybrEx [16], Sedic [17], Tagged MapReduce [18] propose a model of utilizing multiple clouds via MapReduce programming model while addressing concerns in processing private and sensitive data in public clouds.

## VII. Conclusions and Future Work

This paper has presented PigOut, a federated data processing system over multiple Hadoop clusters. PigOut takes care of all aspects of automation, including script and workflow generation, data transfer, and optimization suitable for cross-cluster execution. PigOut does not require any additional effort from existing Hadoop users or cluster administrators since it supports Pig Latin's syntax out of the box and leverages standard, off-the-shelf Hadoop components without any modification. Our experiments show that PigOut produces scripts and execution plans that can provide comparable performance to that of manual, hand-tuned scripts and plans.

Our future work mainly includes incorporating other strategies in various phases. Currently, PigOut uses heuristics for script optimization and a simple tool for data transfer. Although our experiments show that these are practically enough to use as-is, we plan to look into other optimization strategies and data transfer tools that might give us even better performance. Another possible enhancement is to support customizable error recovery schemes. While generating workflow definitions, PigOut can specify how to react to failures. If a user gives hints in scripts, PigOut could specify a proper error recovery routine for each generated script. For example, if one of the data sources is not considered critical in the final result, PigOut can specify Oozie to proceed even in a case that the fetching and filtering part of the workflow fails or cannot be run because the cluster is unavailable.

## References


[1] Apache, "Apache Hadoop," http://hadoop.apache.org, accessed May 27, 2014.

[2] "Altior's AltraSTAR - Hadoop storage accelerator and optimizer now certified on CDH4 (Cloudera's distribution including Apache Hadoop version 4)," http://goo.gl/QIUeoX, 2012, retrieved on June 26, 2014.

[3] Apache, "Apache Pig: high-level dataflow system for Hadoop," http://pig.apache.org, accessed May 27, 2014.

[4] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD '08*.

[5] Apache, "Apache Oozie workflow scheduler for Hadoop," http://oozie.apache.org, accessed May 27, 2014.

[6] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: a framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, Jul. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1239649.1239653

[7] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1039–1065, Aug. 2006. [Online]. Available: http://dx.doi.org/10.1002/cpe.v18:10

[8] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high level dataflow system on top of mapreduce: The pig experience," *PVLDB*, 2009.

[9] Apache, "PigMix," https://cwiki.apache.org/confluence/display/PIG/PigMix, accessed June 24, 2014.

[10] M. Blasgen, M. Astrahan, D. Chamberlin, J. Gray, W. F. King, B. Lindsay, R. Lorie, J. W. Mehl, T. Price, G. R. Putzolu, M. Schkolnick, P. Selinger, D. R. Slutz, H. R. Strong, I. L. Traiger, B. Wade, and R. Yost, "System R: An architectural overview," *IBM Systems Journal*, vol. 20, no. 1, pp. 41–62, 1981.

[11] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, Aug. 2012. [Online]. Available: http://dx.doi.org/10.14778/2367502.2367519

[12] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Comput. Surv.*, vol. 22, no. 3, pp. 183–236, Sep. 1990. [Online]. Available: http://doi.acm.org/10.1145/96602.96604

[13] J. Wang, D. Crawl, and I. Altintas, "Kepler + hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems," in *WORKS*. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1645164.1645176

[14] L. Wang, J. Tao, H. Marten, A. Streit, S. Khan, J. Kolodziej, and D. Chen, "Mapreduce across distributed clusters for data-intensive applications," in *IPDPSW PhD Forum*, May 2012.

[15] P. Kondikoppa, C.-H. Chiu, C. Cui, L. Xue, and S.-J. Park, "Network-aware scheduling of mapreduce framework on distributed clusters over high speed networks," in *Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit*, ser. FederatedClouds '12. New York, NY, USA: ACM, 2012, pp. 39–44. [Online]. Available: http://doi.acm.org/10.1145/2378975.2378985

[16] S. Y. Ko, K. Jeon, and R. Morales, "The hybrex model for confidentiality and privacy in cloud computing," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2170444.2170452

[17] K. Zhang, X. Zhou, Y. Chen, X. Wang, and Y. Ruan, "Sedic: Privacy-aware data intensive computing on hybrid clouds," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 515–526. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046767

[18] C. Zhang, E.-C. Chang, and R. H. Yap, "Tagged-mapreduce: A general framework for secure computing with mixed-sensitivity data on hybrid clouds," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, May 2014, pp. X–X.