Abstract—Big data processing undoubtedly represents a major challenge of this era. While several programming models and supporting systems have been proposed to deal with such data in so-called “cloud” infrastructures, they all exhibit the same limitation: all data is assumed to be located in one datacenter. This limitation results from cloud vendors promoting the abstraction of omnipresent computing and storage resources.

When dealing with data distributed across datacenters, programmers currently have two options: (1) copying all data to a single datacenter easily becomes tedious if done manually as the original dataset is updated, leads to repetitive copying if performed as part of a program, and is sometimes impossible; (2) writing multiple variants of the same program, with consolidation occurring at different points varying by characteristics of the task (e.g., input sub-dataset sizes) is laborious and does not help determining the most appropriate one for a given run.

This paper introduces geo-distributed data structures and operations for expressing data processing tasks taking place across datacenters. We describe the design and implementation of such data structures and operations for the PigLatin language. We illustrate the performance benefits of our geo-distributed data structures and operations through several benchmarks, showing up to 2× faster response times.

I. INTRODUCTION

Big data processing undoubtedly represents one of the major challenges of this era. Indeed, the current limitations of many applications in healthcare, science, commerce, government, or finance alike are dictated by the amount of data that can be analyzed or processed in a given timeframe.

While hardware dictates certain fundamental boundaries, many currently experienced limitations are due to software and programming paradigms not being aligned with the needs of big data. Several dataflow-based programming models and languages have thus been recently proposed to program data analysis tasks in a way which captures the programmer’s intuition and is amenable to efficient execution. Examples include Microsoft’s LINQ [1], Google’s FlumeJava [2], or Yahoo!’s PigLatin [3]. These represent datasets by data structures, and operations on these datasets by a specific set of operations (e.g., filter, group, join) on these data structures. Typically, these data structures are never incarnated at execution in the client’s address space, and operations are translated to execution entities such as MapReduce [4] jobs for parallelization.

A. Geo-Distributed Data

One bottleneck for current systems such as the popular Apache Hadoop implementation of MapReduce [5] or the Apache Hadoop Distributed File System (HDFS) employed for data storage is that they assume a single, homogeneously addressable, “cluster” of nodes. In reality, however, more and more often the topology exhibited by processing and storage nodes for a given application in so-called “clouds” is heterogeneous, for a combination of reasons:

• Many analysis tasks involve several datasets, which are not necessarily all stored in the same datacenter (or datacenter zone, sub-network). Most governments and large companies indeed have datacenters for individual institutions, departments, and business units. Analysis tasks often involve datasets generated and maintained by different units. Legal constraints may restrict certain datasets to specific locations, while certain analyses results may still be exported [6].

• An individual logical dataset may consist in several sub-datasets. Consider for example the US Census [7], which was conducted state-wise and thus also stored across states, whilst most analyses consider the entire dataset. Such partitioning across datacenters (zones, sub-networks) can be due to legislation, or for other reasons such as availability limitations.

In other terms, in contrast to the illusion of omni-present uniform storage and computation resources promoted by cloud vendors, clouds are implemented by concrete datacenters with specific locations; data is often geographically distributed (geo-distributed, GD [8]) across datacenters (or zones, sub-networks), and access times are non-uniform, especially but not solely across datacenters [9]. Even when governments or large corporations avoid partitioning of logical datasets and replicate these across a number of datacenters for availability and fault-tolerance, they often do so incompletely, to keep a favorable tradeoff between (a) fast reads of data in individual datacenter locations close to users
(ideally full replication), and (b) fast writes which must be synchronized across replicas (ideally only one replica) [10].

B. Processing Geo-Distributed Data

With the existing toolchain (e.g., Hadoop, HDFS) supporting distribution over datacenters poorly if at all [11], [12], [13], programmers deal with distributed data as follows:

A. Copy all data to one place before running programs locally. If at all possible (see above) this approach has many disadvantages. If performed as part of a dataflow program (in languages supporting this, e.g., FlumeJava), every re-run of the program leads to re-copying the data; if performed manually, copies need to be updated explicitly when original datasets are modified. The approach can also be highly inefficient, for instance if an early filtering phase on some remote input dataset as part of the task yields very small output which would be much faster (and possibly cheaper in the case of public clouds which charge for remote communication, e.g., Amazon EC2 [14]) to transfer instead.

B. Deal explicitly with geo-distribution by performing remote copying of original, intermediate, and result datasets as “integral” part of a dataflow program. In languages which are not full programming languages and provide no means of implementing such custom copy operations (e.g., PigLatin), this can be achieved by breaking dataflow scripts up in the most appropriate manner and invoking them as part of shell scripts with copying performed in between. This can yield a highly effective schedule (i.e., optimally copying and consolidating data with respect to operations) for given input datasets, but as soon as characteristics of the task change — e.g. data location or distribution (partition sizes/size ratios), filtering operations — the efficiency of the program may be hampered. If multiple program variants are developed for the same task, then the onus of choosing a specific one for a given task instance is still left to the user.

To illustrate the effect of distribution, Figure 1 shows a simple performance evaluation where a sequence of two MapReduce jobs, MR1 (sampling) and MR2 (determining $k$ nearest neighbors), are executed on a dataset [15] distributed across two datacenters $DC_1$ and $DC_2$ at varying ratios (x-axis), with the result expected in $DC_2$. The figure compares three schedules, varying by when copying takes place for consolidation in a single datacenter; their names represent the order of operations. As the graph illustrates, the performance of different schedules varies strongly across distributions. The first schedule which copies all data first is always slowest, but in the first distribution (20% of data in $DC_1$) schedule 2 is not much faster (observe that we focus on 600-1200s on the y-axis) and so schedule 1 could be viable. With an increasing portion of data in $DC_1$, schedules 2 and 3 come closer and their relative advantage of schedule 1 increases.

C. Distributed Data Processing with Rout

In this paper we propose a system for geo-distributed (GD) data processing. Our approach exposes only very little geo-distribution to programmers and lets the distributed runtime environment deal with the tedious aspects of geo-distribution such as orchestration of operations.

To that end we propose Rout, an extension of PigLatin, which introduces GD data structures (GD bags and maps) and GD operations to complement the original ones (regular bags, maps, and operations). The latter are retained for backward compatibility and because they are slightly simpler to use. In fact, regular bags and maps can be viewed as special cases of GD bags and maps which are pinned to specific datacenters. Rout’s distributed runtime infrastructure is implemented as a middleware system controlling Hadoop MapReduce and HDFS clusters across the individual involved datacenters, leveraging the topological information included in executed scripts to determine when and where it is most appropriate to copy data across datacenters.

Concretely, this paper makes the following contributions:

- a model of GD dataflow programming through the abstraction of GD data structures and operations and the design and implementation of this model in Rout, an extension of Pig [16], the Apache open-source implementation of PigLatin.
- heuristics for determining GD Rout schedules with efficient data copying, and a middleware system to execute such schedules across datacenters.
- an evaluation of Rout on various typical dataflow benchmarks [4]. In short, Rout performs up to 2× faster compared to corresponding PigLatin scripts.

The remainder of this paper is organized as follows. Section II presents background information on PigLatin and MapReduce. Section III details our system model and describe our GD language. Section IV details techniques and heuristics adapted for efficient GD dataflow execution. Section V evaluates our system. Section VI overviews related work. Section VII draws conclusions.

II. BACKGROUND

Below we provide more detailed background information pertinent to the contributions proposed.

A. MapReduce

MapReduce is a popular, widely used parallel processing framework, based on the paradigm originally introduced
in the Lisp programming language and popularized by Google [4]. Typical motivating scenarios for Google include word count and search, sort, and page ranking. MapReduce is used as an execution environment by several data processing tools (e.g., FlumeJava, Pig). Apache Hadoop [5] is a widely used open-source MapReduce implementation, which is usually executed on top of the integrated Apache HDFS distributed file system.

As the name suggests, a MapReduce operation is performed over a job, consists of two phases. The first one is a map phase where mapper components process input data in the form of key-value pairs and output new sets of key-value pairs. The signature of the Java method for mapping in Apache Hadoop [5] is as follows:

```java
void map(KEYIN key, VALUEIN val, Context ctxt)
```

Output pairs are generated by calling `write()` on the `Context` argument, and are referred to as intermediate data.

The second phase named reduce phase processes output from the map phase via a number of reducer components. Each reducer handles a number of keys from the intermediate data, each key with all values emitted for that key. From this data the reduce phase generates a set of values. The signature of the Java method for mapping is as follows, without output data again emitted via the `Context`:

```java
void reduce(KEYIN key, Iterable<VALUEIN> vals, Context ctxt)
```

B. PigLatin

PigLatin [3] is a dataflow processing language proposed by Yahoo! as part of the Pig data processing system and later moved to Apache Software Foundation to be developed as an open source project [16]. We provide a short overview of the salient features of PigLatin. More details can be found at [17].

Types: PigLatin introduces simple types and complex types. The former include signed 32-bit and 64-bit integers (int and long respectively), 32-bit and 64-bit floating point values (float, double), arrays of characters and bytes (chararray, bytarray), and booleans. PigLatin also pre-defines certain values for these types (e.g., null).

Complex types include the following:

- **bags** {...} are collections of **tuples**.
- **tuples** {...} are ordered sets of fields.
- **maps** [...] are sets of key-value pairs **key#value**.

A field is a data item, which can be a bag, tuple, or map. PigLatin statements work with relations; a relation is simply a (outermost) bag of tuples. Relations are referred to by named variables called **aliases**. PigLatin supports assignments.

Operators and expressions: Relations are also created by applying operators to other relations. The main relational operators include:

- **PROJECT** Extracts specified fields from tuples.
- **SELECT** Filters tuples that satisfy a specified condition.
- **GROUP** Groups tuples based on a specified key.
- **ORDER** Sorts tuples by a specified key.
- **UNION** Creates the union of two or more relations; does not require their schemas to be identical.
- **CROSS** Creates the cross product of two or more relations. This expensive operator is implemented via a MapReduce job, and allows for the number of reducers to be specified, as well as a partitioner which guides the distribution of mapper keys to reducers. (Several operators allow parameters of resulting MapReduce jobs to be set.)
- **JOIN** The same operator is used with various parameters to distinguish between inner and outer joins. The syntax closely adheres to the SQL standard.
- **GROUP** Elements of several relations can be grouped according to various criteria. Note that GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples.
- **FOREACH** Generates data transformations based on columns of data.
- **FILTER** This operator is used with tuples or rows of data, rather than with columns of data as FOREACH...GENERATE.

Operators also include arithmetic operators (e.g., +, -, /, *), comparisons, casts, as well as STORE and LOAD operators which we discuss in Section III-C3. PigLatin is an expression-oriented language. Expressions are written in conventional mathematical infix notation, and can include operators and functions described next.

Functions: PigLatin includes built-in and user-defined functions. The former include several different categories:

- **Eval** functions operate on tuples. Examples include AVG, COUNT, CONCAT, SUM, TOKENIZE.
- **Math** functions are self-explanatory. Examples include ABS or COS.
- **String** functions operate on character strings. Examples include SUBSTRING or TRIM.

User-defined functions can be defined in Java, Python, JavaScript and Ruby, with the support for Java being most matured. Java user-defined functions are expressed by subclassing EvalFunc<T>, with T corresponding to the respective return type, and implementing its only method with signature

```java
String exec(Tuple input) throws IOException
```

Implementation: The Pig compiler analyzes a PigLatin script and determines an optimal execution plan for executing the operations defined by the script using a series of MapReduce jobs for executing the dataflow. Since the compiler analyzes the complete script prior to execution, it can optimize the execution by performing steps such as reordering and merging of operations.

Geo-distribution with PigLatin?: To illustrate the difficulty of currently dealing with geo-distribution manually in a language like PigLatin, consider a simple word count [4] that counts the total number of occurrences of words in a large text dataset. Figure 2 outlines a vanilla version of

---

472
input_lines = LOAD 'input_file' AS (line:chararray);
words = FOREACH input_lines GENERATE FLATTEN (TOKENIZE(line)) AS word;
word_groups = GROUP words BY word;
word_count = FOREACH word_groups GENERATE group, COUNT(words);
STORE word_count INTO 'output_file';

Figure 2. WordCount - PigLatin (single datacenter)

// Part 1 : Executed in both datacenters.
input_lines = LOAD 'input_file' AS (line:chararray);
words = FOREACH input_lines GENERATE FLATTEN (TOKENIZE(line)) AS word;
word_groups = GROUP words BY word;
word_count = FOREACH word_groups GENERATE group, COUNT(words);
STORE word_count INTO 'output_file';

// Part 2 : Executed in datacenter DC1 only
records1 = LOAD 'file1';
records2 = LOAD 'file2';
STORE record1, record2 INTO 'file2';

// Part 3: Executed in datacenter DC1 only
all_records = UNION records1, records2;
grouped = GROUP all_records BY word;
word_count = FOREACH grouped GENERATE group, SUM(all_records.count);
STORE word_count INTO 'output_file';

Figure 3. WordCount - PigLatin (multi-datacenter)

such a script in PigLatin, with everything happening in one datacenter.

In short, a set is first generated with elements called line of type chararray read from an input file. Next, the script breaks these lines into tokens, representing individual words. Then, occurrences of a same word are grouped, and their respective number of occurrences counted. Finally, the data in this table is stored in a file output_file in the distributed file system.

Now assume that the dataset consists of two sub-datasets that are located in two datacenters DC1 and DC2 respectively. Figure 3 shows one naive way to implement this scenario.

This variant first determines the word counts individually for each sub-dataset. The first part is thus identical to Figure 2. The second part starting at the copy operation which copies results of DC2 to DC1 is only performed in the prior datacenter. It completes by aggregating the results from the two datacenters to determine the final word counts by taking the UNION of the intermediate results, GROUPing them by word, and SUMming the corresponding counts.

An alternative (cf. A. in Section I), consists in performing the copying of one dataset to the other involved datacenter and combining the input sub-datasets before performing any word counting. In most cases, the version outlined in Figure 3 will be more effective, as the amount of data representing the word counts from an individual sub-dataset will be smaller than the corresponding original input data. Moreover, as mentioned in (B. in Section I), many more distributed "schedules" are envisionable, and, as demonstrated by Figure 1, performance differences can be substantial especially when considering that the two initial counts in Figure 3 can be done in parallel.

III. A DISTRIBUTED DATA PROCESSING LANGUAGE

This section describes our proposed language Rout for processing GD data. The language assumes the system model first described in Section III-A. The Rout runtime infrastructure, Rrun for short, will be detailed in Section IV.

A. System Model

We consider n datacenters, DC1, DC2, ..., DCn. In these datacenters, d original input datasets, denoted by DS1, DS2, ..., DSd, are stored. Each dataset DSi consists of si sub-datasets where 1 ≤ si ≤ n.

A GD dataflow task has to be executed on these datasets. A dataflow task consists of (possibly) GD operations. Each operation takes one or more GD datasets as input and generates one or more GD datasets as outputs. A pinned operation is an operation which executes in a single datacenter that is specified by the dataflow program.

Each input to a GD operation is a GD dataset. Each input to a given pinned operation is a sub-dataset located in a single datacenter. All inputs given to a pinned operation must be located in the same datacenter and all corresponding outputs are also generated in the same datacenter. In the following we describe how this model can be supported in PigLatin.

B. Data Types

As mentioned previously, the main data type in PigLatin scripts is the relation, which is defined as a bag of tuples. PigLatin also defines a map data structure (a.k.a. complex data type) which represents a set of unique key-value pairs.

We keep PigLatin's definition of the tuple and require a given tuple to be pinned to a datacenter. So the data of all fields in a given tuple has to be located in the same datacenter. For backward compatibility, Rout still supports the complex data types bag and map. But, once defined, each bag or map is pinned to a single datacenter. Operations performed on a bag or a map are executed in the datacenter which the corresponding bag or map is pinned to, and results
retrieved after executing the operation are stored in the same datacenter.

To support efficient processing of GD data, we define two new complex data structures, GD bags (gdbags) and GD maps (gdmaps). Similar to the bag data structures in PigLatin, gdbag is a collection of tuples, but unlike a bag, a gdbag may represent tuples from several datacenters. Operations on a gdbag will be performed on its data in all respective datacenters hence the operation will be executed in multiple datacenters. In, Rout the definition of a relation is either a bag or a gdbag.

Analogously, gdmaps may represent key-value pairs from multiple datacenters. But for efficiency, the uniqueness of keys is not enforced across datacenters until, if occurs, all data represented by a gdmap is copied to a single datacenter.

C. Built-in Operators and Functions

We modified implementations of a number of built-in functions to efficiently operate on GD data. However, for backward compatibility, we did not alter the syntax of any built-in-functions, but defined new ones for processing GD data when required.

1) Eval, string and math functions: As mentioned previously PigLatin defines a number of built-in eval, math and string functions that can be used to evaluate values of one or more tuples.

All string, math, and eval functions that are applied tuple-wise, i.e., functions that are applied to each tuple individually, are supported by Rout without modifications. This includes functions such as ABS, TRIM, CONCAT, DIFF, and TOKENIZE. When executing these functions on a relation, Rrun performs them in every datacenter where data represented by that relation is stored. The results from these executions are stored in the respective datacenters as well. In other words these functions are associative.

Many other eval functions of PigLatin are applied to groups of tuples instead of being applied to individual tuples separately. Examples are COUNT, AVG, MAX, and SUM. Some of these functions such as SUM, MAX, and MIN are associative by definition since results from multiple datacenters can be merged using the same function. But other operations, require a separate function for merging results from different datacenters. We define the original function as the work function and the corresponding function that should be used to merge results from different datacenters as the merge function. Some functions even though non-associative by nature, can be applied in a GD manner by using a merge function to consolidate results across datacenters. For example the non-associative COUNT function can use SUM as merge function.

If for a given work function a merge function has been provided, Rrun can apply the work function while keeping data GD; it will apply the provided merge function when derived data is eventually copied to a single datacenter.

If a user cannot or does not provide a merge function, and if the provided work function is non-associative, all data represented by the corresponding data structure will be automatically copied to a single datacenter in a transparent manner prior to applying the work function.

2) Relational operators: Many of the relational operators of PigLatin operate on multiple relations. Examples are JOIN, UNION, and CROSS. These operators are non-associative, since for many scripts, to guarantee the safe application of subsequent operations, all data has to be located in a single datacenter prior to the application of the multi-relation relational operator. Some other relational operators are applied to single relations but affect groups of tuples. Examples include ORDER and DISTINCT. These functions are non-associative by definition as well. Other relational operators such as FOREACH and FILTER are applied to individual tuples and are hence associative.

For some of the non-associative relational operators, we define versions that can be applied to GD data. For example we define a GDJOIN operator that has a syntax similar to the JOIN operator but can be applied on a gdbag without copying all data to a single datacenter. When executed the operation is applied separately to each datacenter with respective data of the target gdbag. So the GDJOIN operator results in a partial join of data instead of a full join. This can be useful in many scenarios. An example is provided as a part of the empirical evaluation in Section V. Similarly we define GDUNION and GDCROSS operators that can be applied to perform partial computations while keeping data GD. But in many scenarios, to guarantee the safe execution of subsequent operations, regular operators like JOIN and GROUP will have to be applied to gdbag and gdmap data structures instead, which will result in data being copied to a single datacenter before the function defined by the operator is executed.

3) Loading and storing data: Rout supports PigLatin’s LOAD operator for loading data with a slight modification. The operator syntax is as follows:

```
LOAD alias [USING function] AS schema 
IN location
```

This yields a bag or a map depending on the schema. As previously mentioned, a bag or map in Rout is always pinned to a given datacenter. The datacenter to which the bag or map is pinned to is defined by the location parameter. This location is propagated throughout the script when deriving data from the relation. If omitted, the value defaults to the datacenter where the script is being executed.

For loading GD data Rout defines a new operator GDLOAD. The syntax of the operator is similar to PigLatin’s LOAD operation and is as follows:

```
GDLOAD gd_file [USING function] AS schema
```

The above produces either a gdbag or a gdmap depending on the schema. gd_file is a string identifier representing
GDLOAD

that will produce a
is always a
uses
'output_file';
can be either a
COUNT =
gdmap
bag is a
group,
input_lines
COLLAPSE
[,
 data structure and
has to be pinned to a
AS 'input'
GDUNION
operation is applied to a
COLLAPSE
FOREACH gdbag
the
gd_alias
bag
only
bag
gdbag
or a
word;
GDGROUP =
AS ...

data structure.

We introduce the operator
is used to generate a
gdbag
COLLAPSE
gdbag
or
has to be copied to a single
COLLAPSE
word;
bag
gd_word_count;
map
data structure. Then a series
is a
FOREACH
alias
but
gd_words
[ word_count
STORE
or
gdmap
data structures using
in the datacenter to
data
gdbag
is the original
alias
that can be used to store a gdbag
or a gdmap. The syntax of the operator is as follows:
GDSTORE alias INTO gd_file [USING function]

Here, alias is the gdbag or gdmap that has to be stored
and gd_file refers to a set of key-value pairs where each
key represents a datacenter and each value represents a file
name of the distributed file system in the corresponding
datacenter.

D. User-defined Functions

A PigLatin user-defined function is implemented by
writing a subclass of class EvalFunc<T>. This defines
the work function. Rout supports the same interface
but users can optionally provide a second EvalFunc<T>
definition as a merge function. Additionally, when extending
EvalFunc<T>, a user can optionally specify
whether it’s associative or not by using the annotation
@associative=true or @associative=false
respectively, with the latter value being assumed as default. If
the work function is associative, Rrun can perform the
corresponding operation on GD data types. If the function is
non-associative, as mentioned previously, Rrun will decide
whether to perform a part of the operation while keeping
data GD, depending in the availability of a merge function.
Note that Rout currently only supports user-defined Java
functions.

E. Moving Between Pinned and GD Data Structures

We introduce the operators that can be used to generate
gdbag/gdmap data structures using bag/map data structures
and vice-versa.

1) From pinned to GD: We define a new operator named
GDCONVERT that takes a single bag/map and convert it to
a gdbag/gdmap. The syntax is as follows:
gd_alias = GDCONVERT alias

Here, alias is the original bag/map and gd_alias is the
newly generated gdbag/gdmap. Once converted, the user relinquish
the control of the location of the corresponding data
and Rrun is free to move the data across datacenters. Also,
everse, any operator that can takes gdbags/gdmaps
can be applied on the newly created gdbag/gdmap data
structure.

We also define a another new operator GDUNION that can
take two or more of bags or gdbags and generate a single
gdbag that represent all data that was previously represented
by all bags and gdbags that were passed as parameters.

//input represents data in both datacenters.
gd_input_lines = GDLOAD 'input' AS
(line:chararray);
gd_words = FOREACH input_lines GENERATE
FLATTEN(TOKENIZE(line)) AS word;
gd_word_groups = GDDGROUP gd_words BY word;
gd_word_count = FOREACH gd_word_groups GENERATE
group, COUNT(gd_words);
word_count = COLLAPSE gd_word_count;
STORE word_count INTO 'output_file';

Figure 4. WordCount - Rout (multi-datacenter)

gd_alias = GDUNION [ONSCHEMA] alias,
alias [, alias ...]

Here, each alias can be either a bag or a gdbag but
gd_alias is always a gdbag.

2) From GD to pinned: We introduce the operator
COLLAPSE that will produce a bag/map that represents all
data that was previously represented by a gdbag/gdmap.
Since the produced bag or map has to be pinned to a
single datacenter all data that were represented by the
corresponding gdbag/gdmap has to be copied to a single
datacenter determined by Rrun. If the gdbag/gdmap only
represents data that belongs to a single datacenter, the
returned gdbag/gdmap will obviously be pinned to that
datacenter, without inter-datacenter copying taking place.
The syntax of the operator is as follows:
alias = COLLAPSE gd_alias

Here gd_alias is a gdbag/gdmap data structure and
alias is a bag/map data structure.

When the COLLAPSE operation is applied to a gdbag
the uniqueness of keys is checked across all data and an error
is thrown if this cannot be enforced.

F. Illustration

We describe how the example word count application
described in the Section I can be efficiently implemented
using Rout. The script is presented in the Figure 4.

This script illustrates how Rout can be used to simplify
GD data processing. We start by performing a GDLOAD
operation to generate a gdbag data structure. Then a series
of operations are performed to determine the final word
count value. Finally, COLLAPSE is used to generate a bag
with results in a single datacenter which is stored in the
output_file.

The Rrun executes this as two MapReduce jobs. The first
one, executed in each of the datacenters on respective parts
of data represented by alias, determines the corresponding
word count values for data in each of the datacenters. The
results are copied to a single datacenter as determined by
Rrun. As mentioned, COUNT uses SUM as a merge function
when operating on gdmaps which is used to construct the
second MapReduce job that merges results from multiple datacenters.

IV. Rout Geo-Distributed Runtime Infrastructure

In this section we describe Rrun, a middleware substrate for efficient GD execution of Rout scripts. We demonstrate the effectiveness of our techniques and heuristics in Section V.

A. Overview

When executing a Rout program on GD data, there are two types of optimizations that have to be considered: (1) generating a set of MapReduce jobs for maximizing the parallelization of the program execution and, (2) determining the optimal points in the execution for performing inter-datacenter copy operations (consolidation). Both these optimizations have to be performed to generate a final optimal dataflow graph.

As mentioned in Section II-B, the Pig runtime generates a MapReduce dataflow graph that gives an optimal execution of the corresponding Pig Latin program. Rrun starts by applying the same algorithms and generating a dataflow graph. We refer to this dataflow graph as level one dataflow graph. In this step, Rout assumes all datasets to be located in a single datacenter.

After the above step, Rrun analyzes the generated dataflow graph to determine the places where inter-datacenter copy operations have to be performed. For example, all data of a GD dataset will have to be copied to a single datacenter before performing non-associative functions operation that do not provide a merge function. The dataflow graph will be annotated accordingly. We refer to the technique used to determine the scheduling of these copying operations as copy heuristic detailed below. The resulting updated dataflow graph is referred to as level two dataflow graph. Figure 5 provides an overview of the steps in the execution of a Rout program.

Figure 5. Execution steps in a Rout program

B. Copy Heuristic

When applying the copy heuristic to a level one plan, each MapReduce job of the plan is analyzed. After the copy heuristic has been applied each MapReduce job will be annotated as associative or non-associative. Additionally some of the MapReduce jobs in the dataflow graph will be replaced by two sub-MapReduce jobs instead. Our current heuristic, which can be characterized as a lazy copying heuristic, performs annotation in more detail as follows:

- If the MapReduce job consists of all associative operations, the MapReduce job is annotated as associative.
- Else, if the very first operation of the MapReduce job is non-associative and does not provide a merge function then the MapReduce job is annotated as non-associative.
- Otherwise, the MapReduce job is replaced by two MapReduce jobs and a copy operation. The first operation of the second MapReduce job is defined as the boundary operation. The boundary operation is the merge function of the first non-associative operation if a merge function is provided for that particular operation. Otherwise boundary operation is the first non-associative operation itself. The first job consists of all operations in the MapReduce job up to (not including) boundary operation. The second job starts with the boundary operation and also contains all operations that are after the boundary operation. A copy operation is added after the first MapReduce jobs so that all data will be copied to a single datacenter after applying the first MapReduce job.

Note that even though this step determines when the copy operations should be performed it does not determine which datacenter the data should be copied to. This will be determined at runtime, when the dataflow graph is executed, considering dynamic factors as needed.

C. Plan Execution

Associative MapReduce jobs are executed without copying data across datacenters. That is, such MapReduce jobs are executed individually on sub-datasets in the respective datacenters, and the results are stored in the same respective datacenters. Copy operations consolidate all corresponding data in a single datacenter.

The copy heuristics above follow a lazy execution strategy, as mentioned, in that sub-datasets of a given dataset are copied to a single datacenter only when absolutely necessary. Since in most cases the output of a MapReduce job is smaller than its input, this heuristic will make sure that the amount of data that has to be subjected to costly inter-datacenter copy operations is minimized. Also this way we can use dynamic runtime information about the clusters to determine the ideal datacenters to which the datasets should be copied. In that context the current policy is to select as destination the datacenter whose Hadoop cluster exhibits lowest utilization. Utilization here is defined here as the percentage of time the cluster is used to execute at least a single MapReduce job, and is determined by invoking corresponding Hadoop administrative interfaces at regular intervals.

Users can override this policy with a different one as required. To this end, we define an interface DatacenterValue that has a single method getDatacenterValue() which allows users to specify the criteria for selecting a given datacenter based on user
public interface DatacenterValue {
    int getDatacenterValue();
}

Figure 6. DatacenterValue interface

policies. The interface is given in the Figure 6. In Rout configurations the implementation that should be used has to be specified and is instantiated by Rout agents running in each datacenter. More details about Rout agents will be given later.

To determine the optimal datacenter to which the dataset should be copied, Rout contacts each of its agents and obtains the value exposed through their respective DatacenterValue instances. The datacenter that gives the highest value is determined to be the datacenter to which data should be copied. In our current Rrun implementation the getDatacenterValue() method returns the inverse of the utilization of the corresponding datacenter.

D. Implementation

Rout is implemented based on the Apache Pig codebase. Rrun interprets Rout programs using an extended Pig interpreter and is implemented on top of Hadoop and HDFS. Figure 7 provides an overview of the architecture of Rrun. A Rout program is submitted to a Rout server on the user’s host. This server in turn communicates with Rout agents deployed alongside Hadoop/HDFS clusters in each involved datacenter.

The server is responsible for generating the level two dataflow graph of a program. Each agent is responsible for executing the part of the program (i.e. corresponding MapReduce jobs) that operates on the data stored in its respective datacenter and sending status information regarding the MapReduce executions back to the server. Additionally agents maintain DatacenterValue instances for providing dynamic information regarding their respective clusters to the server.

Figure 7. Deployment architecture of Rrun

This section illustrates the performance benefits of Rout through several representative PigLatin/MapReduce scripts.

A. Setup

For our experiments we used two different datasets and two datacenters. Datacenters were Amazon EC2 N. Virginia (DC1) and Amazon EC2 N. California (DC2). In each datacenter we maintained a Hadoop cluster of five nodes with each 1.7 GB of memory and 1 virtual core running Ubuntu Linux. The first dataset, HADOOPDATA, is a large collection of log files of Hadoop MapReduce jobs executed by Yahoo! [18]. The total size of the dataset is 5 GB. The second dataset, WEATHERDATA, consists of daily global weather measurements obtained from 1930 to 2009 [19]. The total size of the dataset is 6 GB. Note that the trends exhibited by these datasets are amplified with increasing dataset sizes. So our solution becomes even more relevant for larger datasets that are hundreds of gigabytes or even terabytes in size as long as the Hadoop execution engine can handle them. To show this we evaluate both the full datasets and smaller subsets of them. For any dataset used by a given benchmark we refer to these two sets as Dataset2 and Dataset1 respectively, regardless of the actual dataset.

We compare the execution time of Rout scripts with corresponding PigLatin scripts which initially copy all data to a single datacenter (DC1). Obviously, along the lines of approach B. in Section I-B, a set of PigLatin scripts that differ in when they perform copying can achieve performances comparable to Rout, but the latter achieves best possible execution time for different task parameters with minimal user intervention. We omitted the scripts due to space limitations.

In the following experiments, when performing any copy operations, we use gzip compression to improve its efficiency by reducing the amount of data that has to be transferred across datacenters. This actually reduces the benefits of optimized copying in Rout, as Rout precisely attempts to minimize the amount of copied data.

B. Log Debugger

This script can be used to perform simple debugging operations on HADOOPDATA. The script first determines the number of occurrences of each word present in the input dataset. Then it searches for specific words to determine if the number of their occurrences has an unexpected value. For example, there is a problem in the cluster if the word “Exception” has thousands of occurrences.

Figure 8(a) compares the execution times of the PigLatin and Rout scripts. The scripts were run on the full 5 GB dataset (Dataset2 – 3 GB in DC1 and 2 GB in DC2) and a smaller partial dataset of size 2 GB (Dataset1 – 1 GB in DC1 and 1 GB in DC2). Figures 9(a) and 10(a) drill down on the execution times for the different parts of the
execution, representing their individual execution times from left to right in the order in which they occur in the respective schedules. More precisely, PigLatin executes the script as a single MapReduce job. Due to execution techniques detailed in Section IV, Rrun executes the script as two MapReduce jobs instead. The first MapReduce job (CountMR) determines the results for input in each individual datacenter. The results are then copied to a single datacenter (Copy) and merged using another MapReduce job (MergeMR).

As the figures illustrate, Rout performs the task more efficiently than PigLatin – in only 86% of the time for the smaller Dataset1 compared to PigLatin and in only 64% of the time for the full dataset, Dataset2. This is mainly due to two reasons: (1) since Rrun executes a part of the script keeping input data GD, the amount of data that is copied across datacenters is significantly smaller; (2) for executing the count MapReduce job, Rrun employs Hadoop clusters in both datacenters, significantly reducing total execution time.

C. Log Search

The second script performs an advanced search operation, again on HADOOPDATA (thus Dataset1 and Dataset2 are the same as for the previous benchmark). The search can be for specific records in the log files. In our example we searched and listed the execution times of each MapReduce job logged.

Both Rrun and PigLatin dataflow graphs consisted of a single MapReduce job (SearchMR) but in the former case the MapReduce job was executed while keeping input data GD resulting in a MapReduce job being executed in each of the Hadoop clusters. The results from the datacenters can be considered together with no need for merging. The corresponding execution times for Dataset1 and Dataset2
are given in Figure 8(b). Figures 9(b) and 10(b) again details the execution times of the different parts, in the order in which they were executed. For Dataset1, the Rout script executed in only 64% of the total time taken to execute the corresponding PigLatin script while for Dataset2 the Rout script executed within only 57% of the time taken to execute the matching PigLatin script.

D. Weather Explorer

In the wake of thunderstorm Sandy ravaging New York, the third script explores WEATHERDATA for determining historical weather anomalies. In our experiment the script determined all days where the temperature and wind speed were extreme.

Experiments were executed for both the full 6 GB of data (Dataset2 – 3 GB in DC1 and 3 GB in DC2) and a smaller dataset of size 3 GB (Dataset1 – 1.5 GB in DC1 and 1.5 GB in DC2). WEATHERDATA are distributed based on the year of the records. Records from years 1930 to 1970 were in DC1 and the rest were in DC2. Execution times are given in the Figure 8(c). Figures 9(c) and 10(c) again show the execution times for the different parts of the execution, in the order in which they occurred.

As in the previous benchmark, the dataflow graph of the weather explorer script consisted of a single MapReduce job (WeatherMR) for both Rout and PigLatin. But in our script we used the GDJOIN operator to join temperature and wind speed inputs instead of the JOIN operator. Because of this for Rout the scripts could be executed while keeping data GD; the distribution of the input data based on the year of its recording allowed GDJOIN to be applied.

For Dataset1 the Rout script executed within only 52% of the time taken to execute the corresponding PigLatin script while for Dataset2 the Rout script executed within 51% of the time of the corresponding PigLatin script.

VI. RELATED WORK

This section describes work related to Rout and Rrun.

A. Dataflow Processing Languages and Engines

Apache Cascading [20] is another open-source dataflow engine. Cascading consists of a data processing API, integration API, process planer and process scheduler. Like Pig, Cascading operates on top of Hadoop in a single datacenter.

FlumeJava [2] is a statically typed data processing language proposed by Google. It is implemented as a Java library. A dataflow graph making use of MapReduce jobs for parallelization is generated from a FlumeJava program.

PQL is another extension of Java recently proposed by Reichenbach et al. [21]. PQL is a logic-based declarative language which facilitates the expression of parallel elements of a computing task, with regular Java code being used to deal with the sequential portions. The approach supports programmer productivity and allows for aggressive optimizations. PQL does not address (geo-)distribution.

PLINQ [22] is an extension of LINQ [23] for parallel querying of data similar to PQL. PLINQ enjoys explicit syntactic support inside .NET. In contrast to PQL, PLINQ is inspired by SQL-style relational queries (select statements), and is not fully declarative. DryadLINQ [1] is an implementation of LINQ for the [24] execution engine. None of these languages or their runtime environments currently support setups beyond a single datacenter/cluster.

Given that all the above-mentioned languages and models revolve around data structures and collections (e.g., sets, maps along with arrays in PQL) they are amenable to augmentation with GD data structures.

Many efforts focus on optimizing parallel dataflow execution instead of dealing with (geo-)distribution. This includes countless optimizations for the MapReduce framework targeting single clusters. These are complementary to our work. Parcae [25] is a novel parallel execution environment optimized for multi-program executions. The solution includes a compiler that creates flexible parallel programs, a monitor which determines and analyzes resource availability and system performance, and an executor which may replace the executing tasks with ones that are better suited for the operating environment.

Airavat [6] aims at secure analysis of third-party datasets with MapReduce, providing strong privacy guarantees in the face of untrusted tasks. When combined with a secure inter-datacenter data transfer mechanism, solutions such as Airavat can be seamlessly integrated to Rrun to improve the security guarantees of the distributed execution.

B. GD Storage

Most commercial databases such as Oracle and Microsoft SQL Server are designed to handle highly structured data that is available in a single location. Many of these databases support multi-region deployments but for high availability and/or fault tolerance. Thus they are unsuited for handling GD data.

Amazon RDS [26] is a cloud-based database solution. The back-end of RDS is a relational database deployed in Amazon datacenters. RDS supports multi-AZ (availability zone) deployments. But this is not truly GD since Amazon availability zones are located within a single datacenter.

Volley [8] is a system for automatically geo-distributing data based on the needs of an application, presupposing knowledge on the placement of its data, client access patterns, and locations. Based on this information, data will be dynamically migrated between datacenters to optimize access times. Services like Volley complement a GD dataflow engine but do not replace it.

Walter [27] is a key-value storage system that can hold massive amounts of GD data. Walter allows transactions and replication of data across multiple datacenters. A new
feature named parallel snapshot isolation allows Walter to provide strong guarantees while performing replication asynchronously. In Walter, nodes within a datacenter observe transactions on a consistent snapshot with global ordering while nodes across sites only have to follow causal ordering. COPS [10] is another key-value storage system for GD data. COPS is highly scalable and can enforce causal dependencies. The main strategy of COPS consists in enforcing causal dependencies between keys before writes are exposed.

VII. Conclusions

The cloud computing paradigm provides a partial response to data challenges. Clouds are however implemented by concrete datacenters, whose geographical location matter, and no single datacenter can host all the data in the world reliably and efficiently. The current toolchain has to mature by considering real-life constraints such as geographical distribution or administrative domains.

This paper has presented Rout, a data processing language capable of expressing programs involving datasets from multiple datacenters, including individual datasets that are partitioned across datacenters. As illustrated by our empirical evaluation, even on GD data processing programs compiled to few MapReduce stages Rout is able to reduce the execution time down to half compared to a straightforward schedule which consolidates all data first. At the same time, Rout exposes only little geo-distribution to programmers.

We are currently exploring further copy heuristics and datacenter selection policies than those presented in Section IV, the integration of access control for multi-domain policies, and the integration our GD model with other dataflow languages.

REFERENCES


