http://webdam.inria.fr/

Web Data Management

Distributed Computing with MapReduce and Pig

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

http://webdam.inria.fr/Jorge/

Contents

1 MAPREDUCE **4**
1.1 Programming model 4
1.2 The programming environment 6
1.3 MAPREDUCE internals 10

2 PIG **11**
2.1 A simple session 11
2.2 The data model 14
2.3 The operators 15
2.4 Using MAPREDUCE to optimize PIG programs 18

3 Further reading **21**

4 Exercises **22**

So far, the discussion on distributed systems has been limited to data storage, and to a few data management primitives (e.g., *write()*, *read()*, *search()*, etc.). For real applications, one also needs to develop and execute more complex programs that process the available data sets and effectively exploit the available resources.

The naive approach that consists in getting all the required data at the Client in order to apply locally some processing, often looses in a distributed setting. First some processing may not be available locally. Also centralizing all the information then processing it, would simply miss all the advantages brought by a powerful cluster of hundreds or even thousands machines. We have to use distribution. One can consider two main scenarios for data processing in distributed systems.

Distributed processing and workflow. In the first one, an application disposes of large data-sets and needs to apply to them some processes that are available on remote sites. When this is the case, the problem is to send the data to the appropriate locations, and then sequence the remote executions. This is a workflow scenario that is typically implemented using *web services* and some high-level coordination language.

Distributed data and MAPREDUCE. In a second scenario, the data sets are already distributed in a number of servers, and, conversely to the previous scenario, we "push" programs to these servers. Indeed, due to network bandwidth issues, it is often more cost-effective to send a small piece of program from the Client to Servers, than to transfer large data volumes to a single Client. This leads to the MAPREDUCE approach that we present in this chapter.

This second scenario is illustrated in Figure 1. Ideally, each piece of program running as a process on a server n should work only on the data stored locally at n, achieving an optimal reduction of network traffic. More practically, we should try to limit communications by applying local processing as much as possible. We refer to this as the *data locality principle*, i.e., *the distribution strategy must be such that a program component operates, as much as possible, on data stored on the local machine.* In such a setting, the Client plays the role of a coordinator sending pieces of code to each server, initiating, and possibly coordinating a fully decentralized computation.

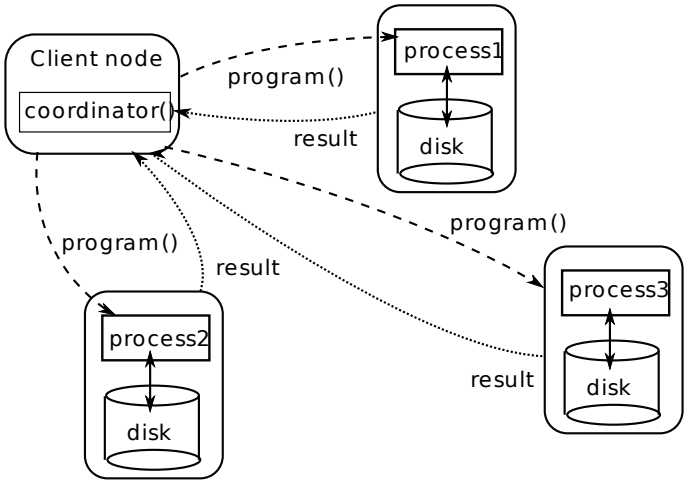

Figure 1: Distributed computing with distributed data storage

Enabling a sound, efficient and reliable distributed data processing gives rise to the following complex issues:

Parallelization. Can we split a particular task into tasks executing concurrently on independent data sets and cooperating to compute a final result? It is not always clear how to answer that question and take advantage of distributed resources. The important word here is *independence*. If a relevant data set can be partitioned, and each part be processed independently, the answer is: yes. Also, if, on the other hand, a program can be split in several tasks that operate independently, the answer is also: yes. If both conditions are satisfied, this is even better. For complex tasks, the answer may not be that simple. In other words, it is not always obvious to see which part of a program can take advantage of parallelization

Failure resilience. When there are a large number of participants involved in a complex task, it becomes necessary to cope with potential system failures. Trying to address them with traditional programming environments used in everyday application development would be a daunting task. What is called for is a programming model, and an associated software support, to facilitate the deployment, monitoring and control of such distributed programs.

In the first part of the chapter, we introduce MAPREDUCE, a programming model for large-scale parallel computing that addresses these issues. Even if developing applications with MAPREDUCE greatly reduces the effort of applications programmers, the task remains very challenging. In the second part, we present the PIGLATIN language that, based on a rich model and high-level language primitives, further allows simplifying the design of distributed data processing applications.

At the time of writing, considerable research and development efforts are devoted to the design of high-level languages that express parallel and distributed data processing. MAPREDUCE is often nowadays taken as a kind of de facto standard for the robust execution of large data-oriented tasks on dozens of computer, at least at a low, "physical" level. However, MAPREDUCE is by no means the universal solution to parallel data processing problems. The

area is still a moving territory subject to debates and alternative proposals. The last section of the chapter attempts, as usual, to provide useful references and discussions.

1 MAPREDUCE

Initially designed by the Google labs and used internally by Google, the MAPREDUCE distributed programming model is now promoted by several other major Web companies (e.g., Yahoo! and Amazon) and supported by many Open Source implementations (e.g, HADOOP, COUCHDB, MONGODB, and many others in the "NoSQL" world). It proposes a programming model strongly influenced by functional programming principles, a task being modeled as a sequential evaluation of stateless functions over non-mutable data. A function in a MAPREDUCE process takes as input an argument, outputs a result that only depends on its argument, and is side-effect free. All there properties are necessary to ensure an easy parallelization of the tasks.

Let us start by highlighting important features that help understand the scope of this programming model within the realm of data processing:

Semistructured data. MAPREDUCE is a programming paradigm for distributed processing of semistructured data (typically, data collected from the web). The programming model is designed for self-contained "documents" without references to other pieces of data, or at least, very few of them. The main assumption is that such documents can be processed *independently*, and that a large collection of documents can be partitioned at will over a set of computing machines without having to consider clustering constraints.

Not for joins. Joins (contrary to, say, in a relational engine) are not at the center of the picture. A parallel join-oriented computing model would attempt, in the first place, to put on the same server, documents that need to be joined. This is a design choice that is deliberately ignored by MAPREDUCE. (We will nonetheless see how to process joins using simple tweaks of the model.)

Not for transactions. MAPREDUCE is inappropriate to transactional operations. In a typical MAPREDUCE computation, programs are distributed to various servers and a server computation typically involves a scan its input data sets. This induces an important latency, so is not adapted to a workload consisting of many small transactions.

So, how come such an approach that does not seem to address important data processing issues such as joins and transactions, could become rapidly very popular? Well, it turns out to be very adapted to a wide range of data processing applications consisting in analyzing large quantities of data, e.g., large collections of Web documents. Also, its attractiveness comes from its ability to natively support the key features of a distributed system, and in particular failure management, scalability, and the transparent management of the infrastructure.

1.1 Programming model

Let us begin with the programming model, ignoring for the moment distribution aspects. As suggested by its name, MAPREDUCE operates in two steps (see Figure 2):

1. The first step, MAP, takes as input a list of pairs (k,v), where k belongs to a key space K_1 and v to a value space V_1. A *map()* operation, defined by the programmer, processes *independently* each pair and produces (for each pair), another *list* of pairs $(k',v') \in K_2 \times V_2$, called *intermediate pairs* in the following. Note that the key space and value space of the intermediate pairs, K_2 and V_2, may be different from those of the input pairs, K_1 and V_1.

2. Observe that the MAP phase may produce several pairs $(k'_1,v'_1),\cdots,(k'_1,v'_p),\cdots$, for the same key value component. You should think that all the values for the same key as grouped in structures of type $(K_2,list(V_2))$, for instance $(k'_1,\langle v'_1,\cdots,v'_p,\cdots\rangle)$.

3. The second step, REDUCE, phase operates on the grouped instances of intermediate pairs. Each of these instances is processed by the procedure *independently* from the others. The user-defined *reduce()* function outputs a result, usually a single value. On Figure 2, the grouped pair $(k'_1,\langle v'_1,\cdots,v'_p,\cdots\cdots\rangle)$ is processed in the REDUCE phase and yields value v''.

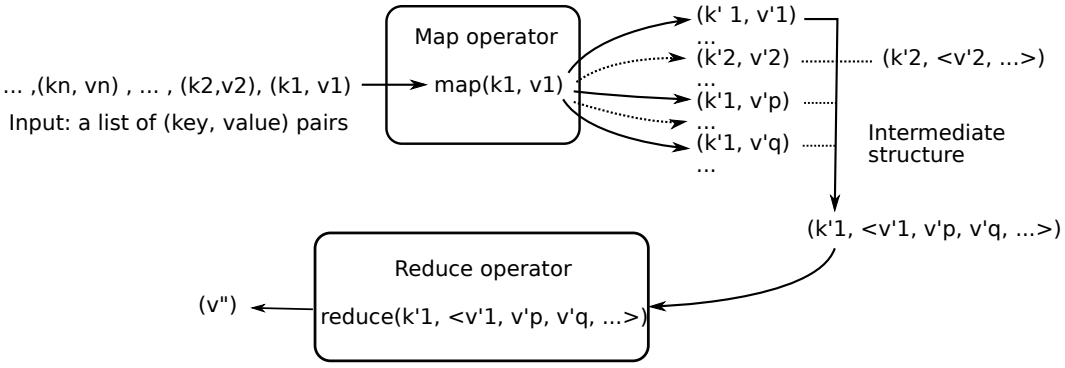

Figure 2: The programming model of MAPREDUCE

Example 1.1 As a concrete example, consider a program *CountWords()* that counts the number of word occurrences in a collection of documents. More precisely, for each word w, we want to count how many times w occurs in the entire collection.

In the MAPREDUCE programming model, we will use a user-defined function *mapCW* that takes as input a pair (i,doc), where i is a document id, and doc its content. Given such a pair, the function produces a list of intermediate pairs (t,c), where t is a term occurring in the input document and c the number of occurrences of t in the document. The MAP function takes as input a list of (i,doc) pairs and applies *mapCW* to each pair in the list.

```
mapCW(String key, String value):
  // key: document name
  // value: document contents

  // Loop on the terms in value
  for each term t in value:
```

```
    let result be the number of occurrences of t in value
    // Send the result
    return (t,result);
```

Now as a result of the MAP phase, we have for each word w, a list of all the partial counts produced. Consider now the REDUCE phase. We use a user-defined function *reduceCW* that takes as input a pair $(t, list(c))$, t being a term and $list(c)$ a list of all the partial counts produced during the MAP phase. The function simply sums the counts.

```
reduceCW(String key, Iterator values):
  // key: a term
  // values: a list of counts
  int result = 0;

  // Loop on the values list; accumulate in result
  for each v in values:
    result += v;

  // Send the result
  return result;
```

The REDUCE function applies *reduceCW* to the pair $(t, list(c))$ for each t occurring in any document of the collection. Logically, this is all there is in MAPREDUCE. An essential feature to keep in mind is that each pair in the input of either the MAP or the REDUCE phase is processed independently from the other input pairs. This allows splitting an input in several parts, and assigning each part to a process, without affecting the program semantics. In other words, MAPREDUCE can naturally be split into independent tasks that are executed in parallel.

Now, the crux is the programming environment that is used to actually take advantage of a cluster of machines. This is discussed next.

1.2 The programming environment

The MAPREDUCE environment first executes the MAP function and stores the output of the MAP phase in an intermediate file. Let us ignore the distribution of this file first. An important aspect is that intermediate pairs (k', v') are clustered (via sorting or hashing) on the key value. This is illustrated in Figure 2. One can see that all the values corresponding to a key k are grouped together by the MAPREDUCE environment. No intervention from the programmer (besides optional parameters to tune or monitor the process) is required.

Programming in MAPREDUCE is just a matter of adapting an algorithm to this peculiar two-phase processing model. Note that it not possible to adapt any task to such a model, but that many large data processing tasks naturally fit this pattern (see exercises). The programmer only has to implement the *map()* and *reduce()* functions, and then submits them to the MAPREDUCE environment that takes care of the replication and execution of processes in the distributed system. In particular, the programmer does not have to worry

about any aspect related to distribution. The following code shows a program that creates a MAPREDUCE job based on the above two functions[1].

```
// Include the declarations of Mapper and Reducer
// which encapsulate mapWC() and reduceWC()
#include "MapWordCount.h"
#include "ReduceWourdCount.h"

// A specification object for \mapreduce/ execution
MapReduceSpecification spec;

// Define input files
  MapReduceInput* input = spec.add_input();
  input->set_filepattern("documents.xml");
  input->set_mapper_class("MapWordCount");

// Specify the output files:
MapReduceOutput* out = spec.output();
out->set_filebase("wc.txt");
out->set_num_tasks(100);
out->set_reducer_class("ReduceWourdCount");

// Now run it
MapReduceResult result;
if (!MapReduce(spec, &result)) abort();
  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.
  return 0;
}
```

The execution of a MAPREDUCE job is illustrated in Figure 3. The context should be now familiar to the reader. The job is distributed in a cluster of servers, and one of these servers plays the special role of a Master. The system is designed to cope with a failure of any of its components, as explained further.

The Client node is, as usual, a library incorporated in the Client application. When the *MapReduce()* function is called, it connects to a Master and transmits the *map()* and *reduce()* functions. The execution flow of the Client is then frozen. The Master considers then the input data set which is assumed to be partitioned over a set of M nodes in the cluster. The *map()* function is distributed to these nodes and applies to the local subset of the data set (recall the data locality principle), called "bag" in what follows. These bags constitute the units of the distributed computation of the MAP: each MAP task involved in the distributed computation works on one and only one bag. Note that the input of a MAPREDUCE job can be a variety of data sources, ranging from a relational database to a file system, with all possible semistructured representations in between. In the case of a relational system, each node hosts a DBMS server and a bag consists of one of the blocks in a partition of a relational table. In the case of a file system, a bag is a set of files stored on the node.

[1]This piece of C++ code is a slightly simplified version of the full example given in the original Google paper on MAPREDUCE.

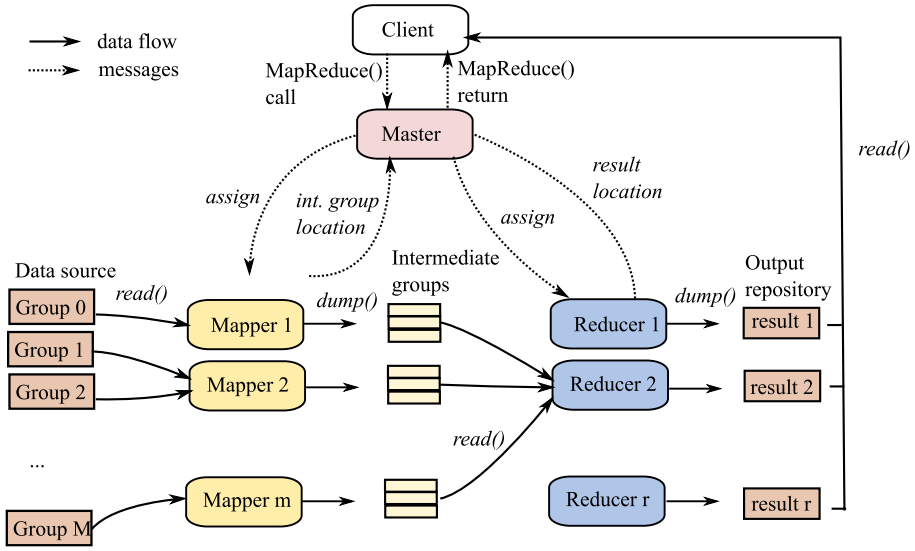

Figure 3: Distributed execution of a MAPREDUCE job.

Whatever the data source, it must support an iterator-like mechanisms that extracts pieces of data from the local bag. A piece of data may be a row in a relational DB, or a line from a file. More generally it is a self-contained object that we call *document* in the following of the chapter.

Example 1.2 Turning back to the *WordCount()* example, suppose the input consists of a collection of, say, one million 100-terms documents of approximately 1 KB each. Suppose we use as data source a large-scale file system, say GFS, with bags of 64 MBs. So, each bag consists of 64,000 documents. Therefore the number M of bags is $\lceil 1{,}000{,}000/64{,}000 \rceil \approx 16{,}000$ bags.

The number of REDUCE tasks, is supplied by the programmer, as a parameter R, along with a *hash()* partitioning function that can be used to hash the intermediate pairs in R bags for sharding purposes. If, for example, the intermediate keys consist of uniformly distributed positive integer values, the simple *modulo(key, R)* partitioning function is an acceptable candidate. In general, a more sophisticated hash function, robust to skewed distribution, is necessary.

At runtime, the MAPREDUCE Master assigns to the participating servers, called *Mappers*, the MAP task for their local chunks. The mapper generates a local list of (k_2, v_2) intermediate pairs that are placed into one of the R local intermediate bags based on the hash value of k_2 for some hash function. The intermediary bags are stored on the local disk, and their location is sent to the Master. At this point, the computation remains purely local, and no data has been exchanged between the nodes.

Example 1.3 Consider once more the *WordCount()* example in a GFS environment. Each chunk contains 64,000 documents, and 100 distinct terms can be extracted from each document. The (local) MAP phase over one bag produces 6,400,000 pairs (t, c), t being a term and c its

count. Suppose $R = 1{,}000$. Each intermediate bag $i, 0 \leq i < 1000$, contains approximately 6,400 pairs, consisting of terms t such that $hash(t) = i$.

At the end of the MAP phase, anyway, the intermediate result is globally split into R bags. The REDUCE phase then begins. The tasks corresponding to the intermediary bags are distributed between servers called *Reducers*. A REDUCE task corresponds to one of the R bags, i.e., it is specified by one of the values of the hash function. One such task is initiated by the Master that sends to an individual Reducer the id of the bag (the value of the hash function), the addresses of the different buckets of the bag, and the *reduce()* function. The Reducer processes its task as follows:

1. the Reducer reads the buckets of the bag from all the Mappers and sorts their union by the intermediate key; note that this now involves data exchanges between nodes;
2. once this has been achieved, the intermediate result is sequentially scanned, and for each key k_2, the *reduce()* function is evaluated over the bag of values $\langle v_1, v_2, \cdots, \rangle$ associated to k_2.
3. the result is stored either in a buffer, or in a file if its size exceeds the Reducer capacity.

Each Reducer must carry out a sort operation of its input in order to group the intermediate pairs on their key. The sort can be done in main memory or with the external sort/merge algorithm detailed in the chapter devoted to Web Search.

Example 1.4 Recall that we assumed $R = 1{,}000$. We need 1,000 REDUCE tasks $R_i, i \in [0,1000[$. Each R_i must process a bag containing all the pairs (t,c) such that $hash(t) = i$.

Let $i = 100$, and assume that *hash('call') = hash('mine') = hash('blog')* = 100. We focus on three Mappers M^p, M^q and M^r, each storing a bag G_i for hash key i with several occurrences of 'call', 'mine', or 'blog':

1. G_i^p =($\langle \ldots$, *('mine', 1)*, $\ldots$, *('call',1)*, $\ldots$, *('mine',1)*, $\ldots$, *('blog', 1)* $\ldots \rangle$
2. G_i^q =($\langle \ldots$, *('call',1)*, $\ldots$, *('blog',1)*, $\ldots \rangle$
3. G_i^r =($\langle \ldots$, *('blog', 1)*, $\ldots$, *('mine',1)*, $\ldots$, *('blog',1)*, $\ldots \rangle$

R_i reads G_i^p, G_i^q and G_i^r from the three Mappers, sorts their unioned content, and groups the pairs with a common key:

$$\ldots, (\text{'blog'}, \langle\ 1, 1, 1, 1\ \rangle), \ldots, (\text{'call'}, \langle\ 1, 1\ \rangle), \ldots, (\text{'mine'}, \langle\ 1, 1, 1\ \rangle)$$

Our *reduceWC*() function is then applied by R_i to each element of this list. The output is *('blog', 4), ('call', 2) and ('mine', 3).*

When all Reducers have completed their task, the Master collects the location of the R result files, and sends them to the Client node, in a structure that constitutes the result of the local *MapReduce()* function. In our example, each term appears in exactly one of the R result files, together with the count of its occurrences.

As mentioned before, the ideal situation occurs when R servers are idle and each can process in parallel a REDUCE task. Because of the two-phases process, a server playing the role of a Mapper may become a Reducer, and process (in sequence) several REDUCE tasks. Generally, the model is flexible enough to adapt to the workload of the cluster at any time. The optimal (and usual) case is a fully parallel and distributed processing. At the opposite, a MAPREDUCE job can be limited to a single machine.

1.3 MAPREDUCE internals

A MAPREDUCE job should be resilient to failures. A first concern is that a Mapper or a Reducer may die or become laggard during a task, due to networks or hardware problems. In a centralized context, a batch job interrupted because of hardware problem can simply be reinstantiated. In a distributed setting, the specific job handled by a machine is only a minor part of the overall computing task. Moreover, because the task is distributed on hundreds or thousands of machines, the chances that a problem occurs somewhere are much larger. For these reasons, starting the job from the beginning is not a valid option.

The interrupted task must be reassigned to another machine. The Master periodically checks the availability and reacheability of the "Workers" (Mapper or Reducer) involved in a task. If the Worker does not answer after a certain period, the action depends on its role:

Reducer. If it is a Reducer, the REDUCE task is restarted by selecting a new server and assigning the task to it.

Mapper. If it is a Mapper, the problem is more complex, because of the intermediate files. Even if the Mapper finished computing these intermediary files, a failure prevents this server to serve these files as input to some reducers. The MAP task has to be re-executed on another machine, and any REDUCE task that has not finished to read the intermediate files from this particular failed node must be re-executed as well.

This leads to the second important concern: the central role of the Master. In summary:

1. It assigns MAP and REDUCE tasks to the Mappers and the Reducers, and monitors their progress;
2. It receives the location of intermediate files produced by the Mappers, and transmits these locations to the Reducers;
3. It collects the location of the result files and sends them to the Client.

The central role of the Master is a potential architectural weakness. If the Master fails, the MAPREDUCE task is jeopardized. However, there is only one Master, and many more workers. The odds for the Master to fail are low. So it may be tolerable for many applications that when a Master fails, its clients resubmit their jobs to a new master, simply ignoring all the processing that has already been achieved for that task. Alternatively, one can realize that the issue is not really the failure of a Master but the loss of all the information that had been gathered about the computation. Using standard techniques based on replication and log files, one can provide recovery from Master failure that will avoid redoing tasks already performed.

It should be clear to the reader how complex data processing tasks can be performed using MAPREDUCE. However, the reader may be somewhat afraid by the complexity of the task facing the application programmer. In a second part of this chapter, we present the PIGLATIN language. The goal is to use a rich model and high-level language primitives, to simplify the design of distributed data processing applications.

2 PIG

The MAPREDUCE processing model is low-level. The computation of complex tasks with MAPREDUCE typically requires combining several jobs. Frequently used operations such as sort or group must be repeatedly introduced in applications as map/reduce functions, and integrated with more application specific operations. To design large-scale data processing applications, it would be definitely useful to dispose of a language that would save the burden of these low-level tasks while preserving the assets of MAPREDUCE. In some sense, this can be compared to introducing declarative languages such as SQL in databases, to facilitate the task of developing applications and thereby improve the productivity of application programmers.

To illustrate the use of high-level language primitives, we present the PIG environment and PIG (or PIGLATIN) language. In spite of sometimes clumsy ad hoc features, the language is in general quite adapted to standard large scale data processing tasks. Another advantage is that it can be tested with minimal installation overhead. PIG brings two important features with respect to the MAPREDUCE approach: (i) a richer data model, with nested data structures, and (ii) expressive data manipulation primitives that can be combined in *data flows* to obtain complex operations.

In brief, a PIG program takes as input a "bag" represented in a file. We will detail the bag data structure further, but it is a roughly speaking a *nested relation*, i.e., a relation where the entries may themselves be relations. A PIG program also produces a bag, either stored in a file or displayed on screen.

We begin with a short illustrative session, and then develop the data and processing model of PIG. The *Putting into Practice* chapter devoted to HADOOP gives practical hints and exercises to experiment with PIG.

2.1 A simple session

Consider the following simple example: given a file with a list of publications in a scientific journal, determine the average number of papers published each year. We use data coming from DBLP, a large collection of information on scientific publications, publicly available[2] in XML.

The PIG loader accepts a variety of input formats. We use here the default file format that it accepts. Each line of the file is interpreted as an entry (here a publication). Within a line, the attributes are separated by *tabs*. Suppose the input consists of the following lines:

```
2005    VLDB J. Model-based approximate querying in sensor networks.
1997    VLDB J. Dictionary-Based Order-Preserving String Compression.
2003    SIGMOD Record   Time management for new faculty.
```

[2] *http://www.sigmod.org/dblp/db/index.html*

```
2001    VLDB J. E-Services - Guest editorial.
2003    SIGMOD Record   Exposing undergraduate students to system internals.
1998    VLDB J. Integrating Reliable Memory in Databases.
1996    VLDB J. Query Processing and Optimization in Oracle Rdb
1996    VLDB J. A Complete Temporal Relational Algebra.
1994    SIGMOD Record   Data Modelling in the Large.
2002    SIGMOD Record   Data Mining: Concepts and Techniques - Book Review.
...
```

Each line gives the year a publication was published, the journal it was published in (e.g., the *VLDB Journal*) and its title.

Here is the complete PIG program that computes the average number of publications per year in SIGMOD RECORD.

```
-- Load records from the journal-small.txt file (tab separated)
articles = load '../../data/dblp/journal-small.txt'
    as (year: chararray, journal:chararray, title: chararray) ;
sr_articles = filter articles BY journal=='SIGMOD Record';
year_groups = group sr_articles by year;
avg_nb = foreach year_groups generate group, COUNT(sr_articles.title);
dump avg_nb;
```

When run on a sample file, the output may look as follows:

```
(1977,1)
(1981,7)
(1982,3)
(1983,1)
(1986,1)
...
```

The program is essentially a sequence of operations, each defining a temporary bag that can be used as input of the subsequent operations. It can be viewed as a flow of data transformation, that is linear in its simplest form but can more generally be an *acyclic workflow* (i.e., a directed acyclic graph).

We can run a step-by-step evaluation of this program with the *grunt* command interpreter to better figure out what is going on.

Load and filter. The **load** operator produces as temporary result, a bag named `articles`. PIG disposes of a few atomic types (`int`, `chararray`, `bytearray`). To "inspect" a bag, the interpreter proposes two useful commands: **describe** outputs its type, and **illustrate** produces a sample of the relation's content.

```
grunt> DESCRIBE articles;
articles: {year: chararray,journal: chararray,title: chararray}

grunt> ILLUSTRATE articles;
-------------------------------------------------------------------------------
| articles | year: chararray | journal: chararray | title: chararray      |
-------------------------------------------------------------------------------
```

```
|          | 2003            | SIGMOD Record      | Call for Book Reviews.|
-------------------------------------------------------------------------
```

The file contains a bag of tuples, where the tuple attributes are distinguished by position. After loading, `articles` also contains a bag of tuples, but the tuple attributes are now distinguished by name.

The filter operation simply selects the elements satisfying certain conditions, pretty much like a relational selection.

Group. In the example, the bags resulting from the load or from the filter do not look different than standard relations. However, a difference is that they may have two identical elements. This would happen, in the example, if the file contains two identical lines. Note that this cannot happen in a relation that is a *set* of tuples. Bags allow the repetition of elements. Furthermore, like nested relations, PIG bags can be *nested*. The result of a **group** for instance is a nested bag. In the example, the group operation is used to create a bag with one element for each distinct year:

```
grunt> year_groups = GROUP sr_articles BY year;

grunt> describe year_groups;
year_groups: {group: chararray,
    sr_articles: {year: chararray,journal: chararray,title:chararray}}

grunt> illustrate year_groups;
 group: 1990
 sr_articles:
  {
   (1990, SIGMOD Record, An SQL-Based Query Language For Networks of Relations.),
   (1990, SIGMOD Record, New Hope on Data Models and Types.)
  }
```

PIG represents bags, nested or not, with curly braces `{}`. Observe the `year_groups` example provided by the **illustrate** command. Note that the grouping attribute is by convention named `group`. All the elements with the same year compose a nested bag.

Before detailing PIG, we summarize its main features essentially contrasting it with SQL:

- Bags in PIG allow repeated elements (therefore the term *bag*) unlike relations that are sets of elements.
- Bags in PIG allow nesting as in nested relations, but unlike classical relations.
- As we will see further, in the style of semistructured data, bags also allow further flexibility by not requiring any strict typing, i.e., by allowing heterogeneous collections.
- For processing, PIG is deliberately oriented toward batch transformations (from bags to bags) possibly in multiple steps. In this sense, it may be viewed as closer to a workflow engine than to an SQL processor.

Note that these design choices have clear motivations:

- The structure of a bag is flexible enough to capture the wide range of information typically found in large-scale data processing.

- The orientation toward read/write sequential data access patterns is, of course, motivated by the distributed query evaluation infrastructure targeted by PIG program, and (as we shall see) by the MAPREDUCE processing model.
- Because of the distributed processing, data elements should be processable independently from each other, to make parallel evaluation possible. So language primitives such as references or pointers are not offered. As a consequence, the language is not adapted to problems such as graph problems. (Note that such problems are notoriously difficult to parallelize.)

The rest of this section delves into a more detailed presentation of PIG's design and evaluation.

2.2 The data model

As shown by our simple session, a PIG *bag* is a bag of PIG tuples, i.e., a collection with possibly repeated elements. A PIG *tuple* consist of a sequence of values distinguished by their positions or a sequence of (attribute name, attribute value) pairs. Each value is either atomic or itself a bag.

To illustrate subtle aspects of nested representations, we briefly move away from the running example. Suppose that we obtain a nested bag (as a result of previous computations) of the form:

```
a :  { b : chararray, c : { c' : chararray }, d : { d' : chararray } }
```

Examples of tuples in this bag may be:

$$\langle a : \{\quad \langle b:1, c:\{\langle c':2\rangle, \langle c':3\rangle\}, d:\{\langle d':2\rangle\}\rangle, \qquad \langle b:2, c:\emptyset, d:\{\langle d':2\rangle, \langle d':3\rangle\}\rangle \quad \}\rangle$$

Note that to represent the same bag in the relational model, we would need identifiers for tuples in the entire bag, and also for the tuples in the c and d bags. One could then use a relation over $b_{id}b$, one over $b_{id}c_{id}c$ and one over $b_{id}d_{id}d$:

b_{id}	b
i_1	1
i_2	2

b_{id}	c_{id}	c
i_1	j_1	2
i_1	j_3	3

b_{id}	d_{id}	d
i_1	j_2	2
i_2	j_4	2
i_2	j_5	3

Observe that an association between some b, c and d is obtained by sharing an `id`, and requires a join to be computed. The input and output of a single PIG operation would correspond to several first-normal-form relations[3]. Joins would be necessary to reconstruct the associations. In very large data sets, join processing is very likely to be a serious bottleneck.

As already mentioned, more flexibility is obtained by allowing heterogeneous tuples to cohabit in a same bag. More precisely, the number of attributes in a bag (and their types) may vary. This gives to the programmer much freedom to organize her dataflow by putting together results coming from different sources if necessary.

[3] A relation is in *first-normal-form*, 1NF for short, if each entry in the relation is atomic. Nested relations are also sometimes called not-first-normal-form relations.

Returning to the running example, an intermediate structure created by our program (`year_groups`) represents tuples with an atomic `group` value (the year) and a nested `article` value containing the set of articles published that year.

Also, PIG bags introduce lots of flexibility by not imposing a strong typing. For instance, the following is a perfectly valid bag in PIG:

```
{
  (2005, {'SIGMOD Record', 'VLDB J.'}, {'article1', article2'} )
  (2003, 'SIGMOD Record', {'article1', article2'}, {'author1', 'author2'})
}
```

This is essentially semistructured data, and can be related to the specificity of applications targeted by PIG. Input data sets often come from a non-structured source (log files, documents, email repositories) that does not comply to a rigid data model and needs to be organized and processed on the fly. Recall also that the application domain is typically that of data analysis: intermediate results are not meant to be persistent and they are not going to be used in transactions requiring stable and constrained structures.

PIG has a last data type to facilitate look-ups, namely *maps*. We mention it briefly. A map associates to a key, that is required to be a data atom, an arbitrary data value.

To summarize, every piece of data in PIG is one of the following four types:

- An *atom*, i.e., a simple atomic value.
- A *bag* of tuples (possibly heterogeneous and possibly with duplicates).
- A PIG *tuple*, i.e., a sequence of values.
- A PIG *map* from keys to values.

It should be clear that the model does not allow the definition of constraints commonly met in relational databases: key (primary key, foreign key), unicity, or any constraint that needs to be validated at the collection level. Thus, a collection can be partitioned at will, and each of its items can be manipulated independently from the others.

2.3 The operators

Table 1 gives the list of the main PIG operators operating on bags. The common characteristic of the unary operations is that they apply on a flow of tuples, that are independently processed one-at-a-time. The semantics of an operation applied to a tuple never depends on the previous or subsequent computations. Similarly, for binary operations: elementary operations are applied to a pair of tuples, one from each bag, independently from the other tuples in the two bags. This guarantees that the input data sets can be distributed and processed in parallel without affecting the result.

We illustrate some important features with examples applied to the following tiny data file *webdam-books.txt*. Each line contains a publication date, a book title and the name of an author.

```
1995    Foundations of Databases Abiteboul
1995    Foundations of Databases Hull
1995    Foundations of Databases Vianu
```

Operator	Description
foreach	Apply one or several expression(s) to each of the input tuples.
filter	Filter the input tuples with some criteria.
order	Order an input.
distinct	Remove duplicates from an input.
cogroup	Associate two related groups from distinct inputs.
cross	Cross product of two inputs.
join	Join of two inputs.
union	Union of two inputs (possibly heterogeneous, unlike in SQL).

Table 1: List of PIG operators

```
2010    Web Data Management Abiteboul
2010    Web Data Management Manolescu
2010    Web Data Management Rigaux
2010    Web Data Management Rousset
2010    Web Data Management Senellart
```

```
-- Load records from the webdam-books.txt file (tab separated)
books = load '../../data/dblp/webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
group_auth = group books by title;
authors = foreach group_auth generate group, books.author;
dump authors;
```

Figure 4: Example of **group** and **foreach**

The first example (Figure 4) shows a combination of **group** and **foreach** to obtain a bag with one tuple for each book, and a nested list of the authors.

The operator **foreach** applies some expressions to the attributes of each input tuple. PIG provides a number a predefined expressions (projection/flattening of nested sets, arithmetic functions, conditional expressions), and allows User Defined Functions (UDF) as well. In the example, a *projection* expressed as `books.authors` is applied to the nested set result of the **group** operator. The final `authors` nested bag is:

```
(Foundations of Databases,
   {(Abiteboul),(Hull),(Vianu)})
(Web Data Management,
   {(Abiteboul),(Manolescu),(Rigaux),(Rousset),(Senellart)})
```

The **flatten** expression serves to unnest a nested attribute.

```
-- Take the 'authors' bag and flatten the nested set
flattened = foreach authors generate group, flatten(author);
```

Applied to the nested bag computed earlier, **flatten** yields a relation in 1NF:

```
(Foundations of Databases,Abiteboul)
(Foundations of Databases,Hull)
(Foundations of Databases,Vianu)
(Web Data Management,Abiteboul)
(Web Data Management,Manolescu)
(Web Data Management,Rigaux)
(Web Data Management,Rousset)
(Web Data Management,Senellart)
```

The **cogroup** operator collects related information from different sources and gathers them as separate nested sets. Suppose for instance that we also have the following file *webdam-publishers.txt*:

```
Fundations of Databases Addison-Wesley  USA
Fundations of Databases Vuibert France
Web Data Management     Cambridge University Press      USA
```

We can run a PIG program that associates the set of authors and the set of publishers for each book (Figure 5).

```
--- Load records from the webdam-publishers.txt file
publishers = load '../../data/dblp/webdam-publishers.txt'
    as (title: chararray, publisher: chararray) ;
cogrouped = cogroup flattened by group, publishers by title;
```

Figure 5: Illustration of the **cogroup** operator

The result (limited to *Foundations of databases*) is the following.

```
(Foundations of Databases,
  { (Foundations of Databases,Abiteboul),
    (Foundations of Databases,Hull),
    (Foundations of Databases,Vianu)
  },
  {(Foundations of Databases,Addison-Wesley),
   (Foundations of Databases,Vuibert)
  }
)
```

The result of a **cogroup** evaluation contains one tuple for each group with three attributes. The first one (named `group`) is the identifier of the group, the second and third attributes being nested bags with, respectively, tuples associated to the identifier in the first input bag, and tuples associated to the identifier in the second one. Cogrouping is close to joining the two (or more) inputs on their common identifier, that can be expressed as follows:

```
-- Take the 'flattened' bag, join with 'publishers'
joined = join flattened by group, publishers by title;
```

The structure of the result is however different than the one obtained with **cogroup**.

```
(Foundations of Databases,Abiteboul,Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Abiteboul,Fundations of Databases,Vuibert)
(Foundations of Databases,Hull,Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Hull,Fundations of Databases,Vuibert)
(Foundations of Databases,Vianu,Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Vianu,Fundations of Databases,Vuibert)
```

In this example, it makes sense to apply **cogroup** because the (nested) set of authors and the (nested) set of publishers are independent, and it may be worth considering them as separate bags. The **join** applies a cross product of these sets right away which may lead to more complicated data processing later.

The difference between **cogroup** and **join** is an illustration of the expressiveness brought by the nested data model. The relational join operator must deliver flat tuples, and intermediate states of the result cannot be kept as first class citizen of the data model, although this could sometimes be useful from a data processing point of view. As another illustration, consider the standard SQL **group by** operator in relational databases. It operates in two, non-breakable steps that correspond to a PIG **group**, yielding a nested set, followed by a **foreach**, applying an aggregation function. The following example is a PIG program that computes a 1NF relation with the number of authors for each book.

```
-- Load records from the webdam-books.txt file (tab separated)
books = load 'webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
group_auth = group books by title;
authors = foreach group_auth generate group, COUNT(books.author);
dump authors;
```

The possible downside of this modeling flexibility is that the size of a tuple is unbounded: it can contain arbitrarily large nested bags. This may limit the parallel execution (the extreme situation is a bag with only one tuple and very large nested bags), and force some operators to flush their input or output tuple to the disk if the main memory is exhausted.

2.4 Using MAPREDUCE to optimize PIG programs

The starting point of this optimization is that a combination of **group** and **foreach** operators of PIG can be almost directly translated into a program using MAPREDUCE. In that sense, a MAPREDUCE job may be viewed as a group-by operator over large scale data with build-in parallelism, fault tolerance and load balancing features. The MAP phase produces grouping keys for each tuple. The shuffle phase of MAPREDUCE puts these keys together in intermediate pairs (akin to the nested bags, result of the PIG **group**). Finally, the REDUCE phase provides an aggregation mechanism to cluster intermediate pairs. This observation is at the core of using a MAPREDUCE environment as a support for the execution of PIG programs.

Basically, each **(co)group** operator in the PIG data flow yields a MAPREDUCE tasks that incorporates the evaluation of PIG operators surrounding the **(co)group**. As previously explained, a *join*, can be obtained using a **cogroup** followed by a flattening of the inner nested bags. So, joins can also benefit from the MAPREDUCE environment.

To conclude, we illustrate such a MAPREDUCE evaluation with two of the examples previously discussed.

Example: group and foreach. In a first example, we use the program given in Figure 4, page 16. Following the classical query evaluation mechanism, the compilation transforms this program through several abstraction levels. Three levels are here represented. The "logical" level directly represents the dataflow process. At this point, some limited reorganization may take place. For instance, a **filter** operator should be "pushed" as near as possible to the **load** to decrease the amount of data that needs to be processed.

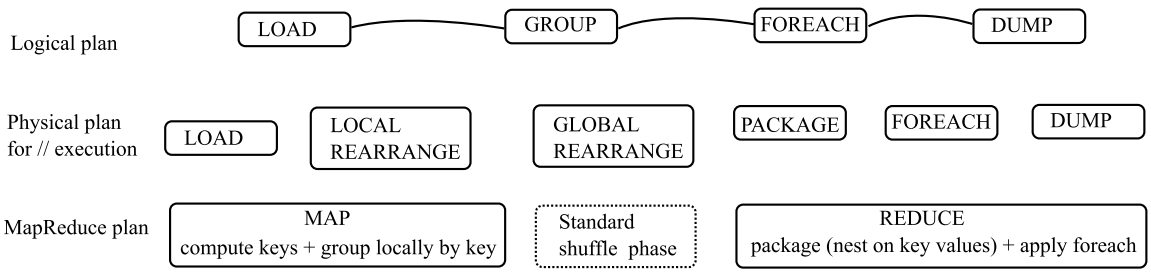

Figure 6: Compilation of a PIG program in MAPREDUCE

The second level represents the sequence of physical operations that need to be executed in a parallel query processing environment. PIG targets several parallel execution models, and this intermediate level provides the means to describe and manipulate a physical plan independently from a specific infrastructure.

The blocks in the physical plan introduce some new operators, namely REARRANGE (LOCAL and GLOBAL), and PACKAGE. REARRANGE denotes a physical operator that groups tuples with the same key, via either hashing or sorting. The distinction between LOCAL and GLOBAL stems from the parallelization context. The LOCAL operator takes place on a single node, whereas the GLOBAL operator needs to collect and arrange tuples initially affected to many nodes. The algorithms that implement these variants may therefore be quite different.

PACKAGE relates to the PIG data model. Once a set of tuples sharing the same key are put together by a REARRANGE, a nested bag can be created and associated with the key value to form the typical nested structure produced by the **(co)group** operation. Expressions in the **foreach** operator can then be applied.

The lower level in Figure 4 shows the MAPREDUCE execution of this physical plan. There is only one MAPREDUCE job, and the physical execution proceeds as follows:

1. MAP generates the key of the input tuples (in general, this operation may involve the application of one or several functions), and groups the tuples associated to given key in intermediate pairs;

2. the GLOBAL REARRANGE operator is natively supported by the MAPREDUCE framework: recall that intermediate pairs that hash to a same value are assigned to a single Reducer, that performs a merge to "arrange" the tuples with a common key together;

3. the PACKAGE physical operator is implemented as part of the *reduce()* function, that takes care of applying any expression required by the **foreach** loop.

Example: join and group. Our second example involves a **join** followed by a **group**. It returns the number of publishers of Victor Vianu. Note that one might want to remove duplicates from the answer; this is left as an exercise.

```
-- Load records from the webdam-books.txt file (tab separated)
books = load '../../data/dblp/webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
-- Keep only books from Victor Vianu
vianu = filter books by author == 'Vianu';
--- Load records from the webdam-publishers.txt file
publishers = load '../../data/dblp/webdam-publishers.txt'
    as (title: chararray, publisher: chararray) ;
-- Join on the book title
joined = join vianu by title, publishers by title;
-- Now, group on the author name
grouped = group joined by vianu::author;
-- Finally count the publishers (nb: we should remove duplicates!)
count = foreach grouped generate group, COUNT(joined.publisher);
```

Figure 7: A complex PIG program with **join** and **group**

Figure 8 shows the execution of this program using two MAPREDUCE jobs. The first one carries out the join. Both inputs (books and publishers) are loaded, filtered, sorted on the title, tagged with their provenance, and stored in intermediate pairs (MAP phase). Specifically, the *map()* function receives rows:

1. either from the `books` input with year, title, and author.
2. or from the `publishers` input with title and publisher. again recording provenance.

Each row records its provenance, either `books` or `publishers`.

These intermediate pairs are sorted during the shuffle phase, and submitted to the *reduce()* function. For each key (title), this function must take the set of authors (known by their provenance), the set of publishers (idem), and compute their cross product that constitutes a part of the join result. This output can then be transmitted to the next MAPREDUCE job in charge of executing the **group**.

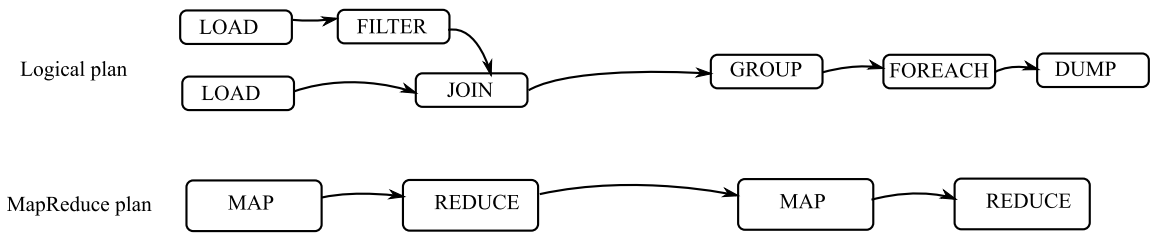

Figure 8: A multi-jobs MAPREDUCE execution

Clearly, this complex query would require an important amount of work with MAPREDUCE programming, whereas it is here fulfilled by a few PIG instructions. The advantage is more related to the software engineering process than to the efficiency of the result. the Due to the rather straighforward strategy applied by the PIG evaluator, early performance reports show that PIG execution is, not surprisingly, slightly worse than the equivalent MAPREDUCE direct implementation. This is notably due to the overhead introduced by the translation mechanism. The next section mentions alternative approaches that pursue similar goal that PIG.

3 Further reading

Distributed computing now has a long history, with Web services as a recent popular outcome. We refer the reader to the general references [TvS01] for distributed systems and [Ble96] for parallel algorithms. At the center of distributed computing we find the possibility of activating some computation on a distant machine. This leads to *remote procedure call*, an abstraction that allows interacting with a remote program while ignoring its details. Some data is sent as argument of the call. The remote program is activated with this data as input. Its result is shipped back to the caller. Note that this involves transmission of data in both directions, from the caller to the callee (parameters of the call) and back (results).

To support such communications, one needs to provide end-points for these communications, e.g. sockets. A communication happens between a local socket and a remote one. To understand each other, they need to use some common protocol for the messages, e.g., TCP, UDP, raw IP, or, in the Web Services realm, SOAP.

Based on such communications, middleware systems have been developed since the 1960's, the so-called *message-oriented* middleware. They are based on asynchronous calls, i.e., the call is made and the caller is not blocked waiting for an answers. The messages are managed in queues. Examples of such systems are IBM Websphere and Microsoft MQ serie.

The object-oriented paradigm proved to be very successful for distributed computing. Indeed, it is very natural to see an external resource as an object, i.e., a black box with a set of methods as interface. This lead to very popular systems, *object brokers*.

From a data management perspective, one may want to support transactions between the distributed machines. This leads to *transaction processing monitors*, , e.g., IBM CICS or BEA Tuxedo. Such systems provide support for persistence, distributed transactions, logging and error recovery.

By merging, object brokers and TP monitors, one obtains the *object monitors*. These systems became popular in the 1990's, notably with Corba from the Object Management Group and DCOM by Microsoft

Closer to us and targeting the Web, we find XML-RPC (in the late 1990's) that, as indicated by its name, is based on remote procedure calls using XML as underlying data format. The calls are performed using HTTP-POST.

Finally, we briefly discuss Corba that had a very important influence in the evolution of distributed computing. Corba stands for *Common Object Request Broker Architecture*. As previously mentioned, it is based on RPC and the object-oriented paradigm. The development of Corba-based components is somewhat independent of the programming language, e.g., C++ or Java may be used. An implementation of Corba consists of the deployment of a system (called an ORB) that provides the interoperability between applications distributed

on different machines. The ORB provides a large set of services, e.g., persistence, transaction, messaging, naming, security, etc. Corba and DCOM were the main supports for distribution before Web services.

There is a long history of research on so-called nested relations, e.g., [AB86], or complex objects, e.g., [AB95], that somehow paved the way for semistructured data models. An algebra for bags, vs. sets of tuples, is considered in [GM99].

Parallel query processing is an old research topic. Issues related to scalable query execution in shared-nothing architecture have been investigated since the emergence of relational systems. See [DGG+96, FKT86] for important milestones, and [DG92] for a position paper. The proposed techniques are now available in several commercial systems, including Teradata (*http://www.teradata.com*), a leading datawarehouse software company. The systems based on Google technology, and in particular MAPREDUCE have been criticized for ignoring previous advances in database technology [DS87]. A detailed discussion of the MAPREDUCE limits and contributions, viewed in a database perspective, is reported in [SAD+10]. MAPREDUCE is suitable for text processing, and more generally for data sets where relational schema does not fit. It is also a convenient tool for cost-effective environments (e.g., commodity hardware) that allow an inexpensive horizontal scalability but lead to unreliable infrastructures where the resilience brought by MAPREDUCE is a valuable asset.

In practical terms, a major restriction of MAPREDUCE is the high latency that stems from both the initial dissemination of a program in the cluster prior to any execution, and the need to fully achieve the MAP phase before running the REDUCE one. This is justified for batch analysis of large data sets but make it unsuitable for transactional applications [PPR+09]. Its attractiveness on the other hand lies in its scalability and fault-tolerance, two features where parallel databases arguably show their limits, at least for web-scale data sets.

Recently, research attempts to benefit from the best of the two worlds have been undertaken. HADOOPDB [ABPA+09] is an "hybrid" distributed data management system that uses a standard relational DBMS (e.g., `PostgreSQL`) at each node, and uses MAPREDUCE as a communication layer between nodes. The relational system instance acts as a source to MAPREDUCE jobs, with the advantage of being able to run complex SQL query plans that exploit database index, saving the otherwise mandatory full scan of the data sets. Other approaches aims at providing high-level data processing languages which can then be executed in a MAPREDUCE-like environment: SCOPE [CJL+08], PIG [ORS+08, GNC+09], JAQL `http://code.google.com/p/jaql/`, and Hive [TSJ+09] are examples of some recent or ongoing efforts.

4 Exercises

Exercise 4.1 (Log processing with MAPREDUCE) *A big company stores all incoming emails in log files. How can you count the frequency of each email address found in these logs with* MAPREDUCE*?*

Exercise 4.2 (Optimizing the MAP and REDUCE phases) *The* REDUCE *phase needs to download intermediate pairs produced by the mappers. How can we reduce the cost of this exchange? The following gives some hints:*

1. *Consider again the WordCount example; propose a post-processing step, running on the mapper, that reduces the size of the files sent to the reducers.*

2. *Now, consider a* MAPREDUCE *task aiming at retrieving the inverse document frequency; does the foregoing optimization still help?*

3. *Finally, one could sort the intermediate pairs before sending them to the reducer; discuss the pros and cons of this approach.*

Exercise 4.3 (SP relational queries) *A Selection-Projection-Aggregation relational query corresponds to the simple SQL syntax:*

```
SELECT <list-attributes>
FROM <someTable>
WHERE <list-conditions>
GROUP BY <attribute>
```

Propose a MAPREDUCE *job (using pseudo-code for map() and reduce()) for the following queries:*

1.
```
SELECT title, year
FROM paper
WHERE author='Jeff Ullman'
AND   published='ACM'
```

2.
```
SELECT title, count(author)
FROM paper
WHERE year=2011
GROUP BY title
```

When is the reduce function really useful? How would you express these queries with PIG.

Exercise 4.4 (Sorting with MAPREDUCE) *How can you obtain a parallel sort with* MAPREDUCE*? For instance, what would be the* MAPREDUCE *parallel execution of the following SQL query:*

```
SELECT title
FROM paper
ORDER BY year
```

Hint: partition the input in R intervals with map(), then sort each local interval with reduce().

Exercise 4.5 (Joins with MAPREDUCE) *And, finally, how can you express joins? For instance:*

```
SELECT title, journalName
FROM   paper p, journal j
WHERE   p.idJournal = j.id
```

Hint: this requires to somewhat distord the MAPREDUCE *principles. The reduce() function should receive pairs* $(id, < p_1, \cdots, p_n >)$ *where id is a journal id and each* p_i *is a row from* `paper`*. By*

unnesting the structure, one gets the expected result. Note that the reduce phase does not reduce at all the output in that case! Such a tweak may not be accepted by all MAPREDUCE *environments.*

Exercise 4.6 (Distributed Monte Carlo) *We want to create a distributed program that approximates π. The method is based on the inscription of a circle in a square (Fig. 9).*

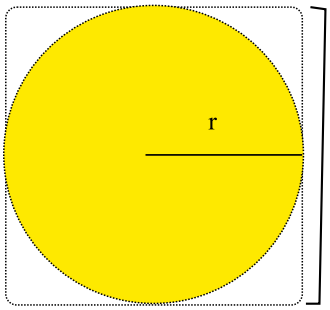

Figure 9: A method to computing π

Note that the area of the square is $A_s = (2r)^2 = 4r^2$; the area of the circle is $A_c = \pi \times r^2$. Therefore

$$\pi = 4 \times \frac{A_c}{A_s}$$

1. *Propose a parallel program that computes an approximation of π; how can you express such a program in* MAPREDUCE*?*

2. *The previous computation of π is actually a simple instance of the classical Monte Carlo method. Assume now a very large data set of geographic* regions*. Each region is identified by a key, we know its contour and can test whether a point belongs to a region thanks to a point-in-polygon (PinP()) function. We can also obtain the minimal bounding box of aa region thanks to the mbb() function. We want to calculate their areas. Propose a distributed implementation based on* MAPREDUCE*.*

Exercise 4.7 (Distributed inverted file construction) *Describe a* MAPREDUCE *job that constructs an inverted file for a very large data set of Web documents. Give the map() and reduce() functions in pseudo-code, and explain the data flow in a distributed system.*

Exercise 4.8 (Distributed PageRank) *Describe a* MAPREDUCE *job that computes one iteration of the PageRank algorithm over a collection of documents. Some hints:*

1. *the map() function takes as input the doc id, the list of URLs that refer to the document, and the current rank;*

2. *the reduce() takes as input a URL and a list of ranks; you can assume that the damping factor is a constant in this function.*

Exercise 4.9 (PIG) *Refer to the Putting into Practice chapter on* HADOOP *(page **??**) for a list of* PIG *queries.*

References

[AB86] Serge Abiteboul and Nicole Bidoit. Non first normal form relations: An algebra allowing data restructuring. *J. Comput. Syst. Sci.*, 33(3):361–393, 1986.

[AB95] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *Very Large Databases Journal (VLDBJ)*, 4(4):727–794, 1995.

[ABPA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MAPREDUCE and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):922–933, 2009.

[Ble96] Guy E. Blelloch. Programming Parallel Algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[CJL+08] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 1(2):1265–1276, 2008.

[DG92] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35(6):85–98, 1992.

[DGG+96] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 1996.

[DS87] D. DeWitt and M. Stonebraker. MAPREDUCE, a major Step Backward. DatabaseColumn blog, 1987. http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

[FKT86] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An overview of the system software of a parallel relational database machine grace. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 209–219, 1986.

[GM99] Stéphane Grumbach and Tova Milo. An algebra for pomsets. *Inf. Comput.*, 150(2):268–306, 1999.

[GNC+09] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a HighLevel Dataflow System on top of MAPREDUCE: The PIG Experience. *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1414–1425, 2009.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 1099–1110, 2008.

[PPR+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 165–178, 2009.

[SAD+10] Michael Stonebraker, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MAPREDUCE and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB*

Endowment (PVLDB), 2(2):1626–1629, 2009.

[TvS01] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2001.