## 16.1 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.

### 16.1.1 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 16.2):

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 16.2, the grouped pair $(k'_1, \langle v'_1, \cdots, v'_p, \cdots \cdots \rangle)$ is processed in the REDUCE phase and yields value $v''$.

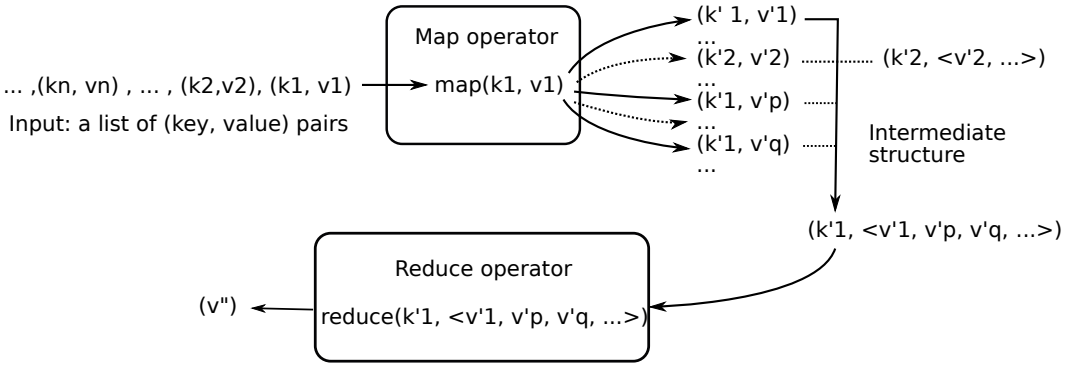


Figure 16.2: The programming model of MAPREDUCE

**Example 16.1.1** 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.

### 16.1.2 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 16.2. 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"
```

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

```
// 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 16.3. 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.

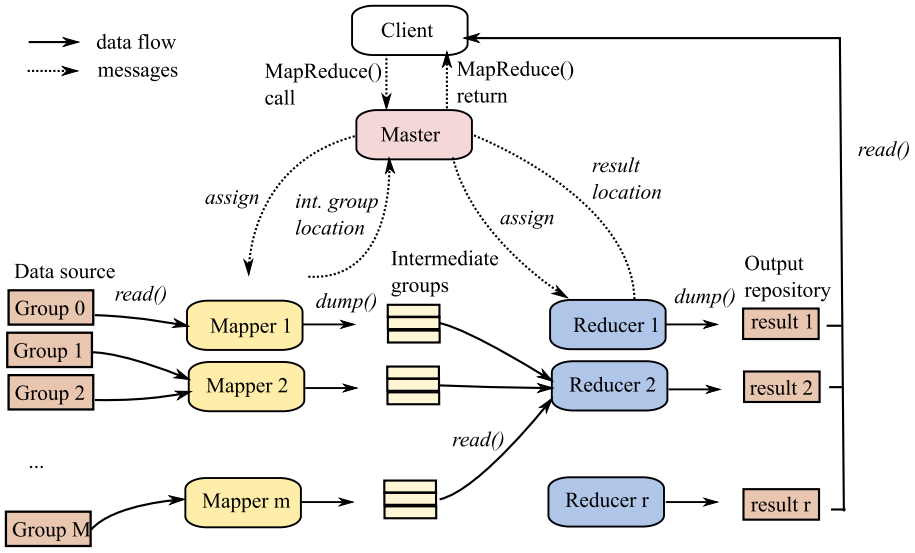


Figure 16.3: Distributed execution of a MAPREDUCE job.

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.

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 16.1.2** 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 16.1.3** 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 16.1.4** 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:

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

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.

### 16.1.3 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.