FP-Hadoop: Efficient Processing of Skewed MapReduce Jobs

Miguel Liroz-Gistau, Reza Akbarinia, Divyakant Agrawal, Patrick Valduriez

FP-Hadoop: Efficient Processing of Skewed MapReduce Jobs

Miguel Liroz-Gistau[a], Reza Akbarinia[a,*], Divyakant Agrawal[b], Patrick Valduriez[a]

[a]*INRIA Montpellier, France*
[b]*Department of Computer Science, University of California, Santa Barbara*

Abstract

Nowadyas, we are witnessing the fast production of very large amount of data, particularly by the users of online systems on the Web. However, processing this big data is very challenging since both space and computational requirements are hard to satisfy. One solution for dealing with such requirements is to take advantage of parallel frameworks, such as MapReduce or Spark, that allow to make powerful computing and storage units on top of ordinary machines. Although these key-based frameworks have been praised for their high scalability and fault tolerance, they show poor performance in the case of data skew. There are important cases where a high percentage of processing in the reduce side ends up being done by only one node.

In this paper, we present *FP-Hadoop*, a Hadoop-based system that renders the reduce side of MapReduce more parallel by efficiently tackling the problem of reduce data skew. FP-Hadoop introduces a new phase, denoted *intermediate reduce* (IR), where blocks of intermediate values are processed by intermediate reduce workers in parallel. With this approach, even when all intermediate values are associated to the same key, the main part of the reducing work can be performed in parallel taking benefit of the computing power of all available workers.

We implemented a prototype of FP-Hadoop, and conducted extensive experiments over synthetic and real datasets. We achieved excellent performance gains compared to native Hadoop, e.g. *more than 10 times in reduce time and 5 times in total execution time.*

Keywords: MapReduce, Data Skew, Parallel Data Processing

1. Introduction

In the past few years, advances in the Web have made it possible for the users of information systems to produce large amount of data. However, processing this big data is very challenging since both space and computational requirements are hard to satisfy. One solution for dealing with such requirements is to take advantage of parallel frameworks, such as MapReduce [1] or its IO-efficient versions such as Spark [2], that allow to make powerful computing and storage units on top of ordinary machines.

The idea behind MapReduce is simple and elegant. Given an input file of key-value pairs, and two functions, map and reduce, each MapReduce job is executed in two main phases. In the first phase, called map, the input data is divided into a set of splits, and each split is processed by a map task in a given worker node. These tasks apply the map function on every key-value pair of their split and generate a set of intermediate pairs. In the second phase, called reduce, all the values of each intermediate key are grouped and assigned to a reduce task. Reduce tasks are also assigned to worker machines and apply the reduce function on the created groups to produce the final results.

Although MapReduce and Spark frameworks have been praised for their high scalability and fault tolerance, they show poor performance in the case of data skew. There are important cases where a high percentage of processing in the reduce side ends up being done by only one node. Let's illustrate this by an example.

Example 1. *Top accessed pages in Wikipedia. Suppose we want to analyze the statistics*[1] *that the free encyclopedia, Wikipedia, has published about the visits of its pages by users. In the statistics, for every hour, there is a file in which for each visited page, there is a line containing some information including, among others, its URL, language and the number of visits. Given a file, we want to return for each language, the top-k% accessed pages, e.g., top 1%.*

To answer this query, we can write a simple program as in the following Algorithm[2]*:*

*Corresponding author
Email addresses: miguel.liroz_gistau@inria.fr (Miguel Liroz-Gistau), reza.akbarinia@inria.fr (Reza Akbarinia), agrawal@cs.ucsb.edu (Divyakant Agrawal), patrick.valduriez@inria.fr (Patrick Valduriez)

[1]http://dumps.wikimedia.org/other/pagecounts-raw/

[2]This program is just for illustration; actually, it is possible to write a more efficient code by leveraging on the sorting mechanisms of MapReduce.

```
map( $id$ : $\mathcal{K}_1$, $content$ : $\mathcal{V}_1$ )
    foreach line ⟨lang, page_id, num_visits, ...⟩ in content do
        emit (lang, page_info = ⟨num_visits, page_id⟩)
    end

reduce( $lang$ : $\mathcal{K}_2$, $pages\_info$ : list($\mathcal{V}_2$) )
    Sort pages_info by num_visits
    foreach page_info in top k% do
        emit (lang, page_id)
    end
```
Algorithm 1: Map and reduce functions for Example 1

In this example, the load of reduce workers may be highly skewed. In particular, the worker that is responsible for reducing the English language will receive a lot of values. According to the statistics published by Wikipedia[3], the percentage of English pages over total was more than 70% in 2002 and more than 25% in 2007. This means for example that if we use the pages published up to 2007, when the number of reduce workers is more than 4, then we have no way for balancing the load because one of the nodes would receive more than 1/4 of the data. The situation is even worse when the number of reduce tasks is high, e.g., 100, in which case after some time, all reduce workers but one would finish their assigned task, and the job has to wait for the responsible of English pages to finish. In this case, the execution time of the reduce phase is at least equal to the execution time of this task, no matter the size of the cluster.

There have been some proposals to deal with the problem of reduce side data skew. One of the main approaches is to try to uniformly distribute the intermediate values to the reduce tasks, e.g., by dynamically repartitioning the keys to the reduce workers [3]. However, this approach is not efficient in many cases, e.g., when there is only one single intermediate key, or when most of the values correspond to one of the keys.

One solution for decreasing the reduce side skew is to filter the intermediate data as much as possible in the map side, e.g., by using a *combiner function*. However, the input of the combiner function is restricted to the data of one map task, thus its filtering power is very limited for some applications. Let's illustrate this by using our problem of top-1%. Suppose we have 1TB of Wikipedia data, and 200 nodes for processing them. To be able to filter some intermediate data by the combiner function, we should have more than 1% of the total values of at least one key (language) in the map task. Thus, if we use the default splits of Hadoop (64 MB size), the combiner function can filter no data. The solution is to increase significantly the size of input splits, e.g. more than 10GB (1% of total). However, using big splits is not advised since it decreases significantly the MapReduce performance due to the following disadvantages: 1) *more map-side skew*: with big splits, there may be some map tasks that take too much time (e.g. because of their slow CPU), and this would increase significantly the total MapReduce execution time; 2) *less parallelism*: big split size means small number of map tasks, so several nodes (or at least some of their computing slots) may have nothing to do in the map phase. In our example, with 10GB splits, there will be only 100 map tasks, thus half of the nodes are idle. This performance degradation is confirmed by our experimental results reported in Section 5.11).

In this paper, we propose *FP-Hadoop*, a Hadoop-based system that uses a novel approach for dealing with the data skew in reduce side. In FP-Hadoop, there is a new phase, called *intermediate reduce (IR)*, whose objective is to make the reduce side of MapReduce more parallel. More specifically, the programmer replaces his reduce function by two functions: *intermediate reduce (IR)* and *final reduce (FR)* functions. Then, FP-Hadoop executes the job in three phases, each phase corresponding to one of the functions: map, intermediate reduce (IR) and final reduce (FR) phases. In the IR phase, even if all intermediate values belong to only one key (i.e., the extreme case of skew), the reducing work is done by using the computing power of all available workers. Briefly, the data reducing in the *IR phase* has the following distinguishing features:

- **Parallel reducing of each key:** The intermediate *values of each key* can be processed in parallel by using multiple intermediate reduce workers.
- **Distributed intermediate block construction:** The input of each intermediate worker is a block composed of intermediate values *distributed over multiple nodes* of the system, and chosen using a *scheduling strategy*, e.g. locality-aware.
- **Hierarchical execution:** The processing of intermediate values in the IR phase can be done in several levels (iterations). This permits to perform *hierarchical execution plans* for jobs such as top-k% queries, in order to decrease the size of the intermediate data more and more.
- **Non-overwhelming reducing:** The size of the intermediate blocks is bounded by configurable maximum value that prevents the intermediate reducers to be overwhelmed by very large blocks of intermediate data.

We implemented a prototype of FP-Hadoop, and conducted extensive experiments over synthetic and real datasets. The results show excellent performance gains of FP-Hadoop compared to native Hadoop. For example, in a cluster of 20 nodes with 120GB of input data, FP-Hadoop outperformed Hadoop by a *factor of about 10 in reduce time, and a factor of 5 in total execution time.*

This paper is a major extension of [4] and [5], with at least 30% of new materials. In the current paper, we

[3]http://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

propose a fault-tolerance mechanism that assures the correctness of the results in the case of failures, and reduces the amount of data to be re-processed compared to the native Hadoop. Additionally, we describe the design in more details and provide a more extensive experimental evaluation.

The rest of this paper is organized as follows. In Section 2, we explain how Hadoop's MapReduce wroks and give the necessary details to present our approach. In Section 3 we present the principles of FP-Hadoop including its programming model. Then, in Section 4, we give more details about its design. In Section 5, we report the results of our experiments done to evaluate the performance of FP-Hadoop. In Section 6, we discuss related work, and Section 7 concludes.

2. MapReduce Background

In this section, we first briefly explain how MapReduce works in Hadoop. This will be useful to understand the technical details of FP-Hadoop. Then, we give an abstract view of the MapReduce execution. This abstract view is useful to better understand the main differences between the programming models of Hadoop and FP-Hadoop.

2.1. Job Execution in Hadoop

In Hadoop, for executing a MapReduce job, we need a *master* node for coordinating the job execution, and some *worker* nodes for executing the map and reduce tasks [6]. The worker nodes can be configured with a predefined number of slots for map and reduce tasks, so that each slot is able to execute a single task at a given time.

When a MapReduce job is submitted to a node, it computes the input *splits*. The number of input splits can be personalized, but typically there is a one-to-one relationship between splits and file chunks in the filesystem, which by default have a size of 64MB. The location of these splits and some information about the job are submitted to the master that creates a job object with all the necessary information, including the map and reduce tasks to be executed. One map task is created per input split.

When a map task is assigned to a worker, it is executed in a Java Virtual Machine (JVM). The task reads the corresponding input split, applies the map function on each input element (e.g. line), and generates intermediate key-value pairs, which are firstly maintained in a buffer in main memory. When the content of the buffer reaches a threshold (by default 80% of its size), the buffered data is stored on the disk in a file called *spill*. Before writing the content of the buffer into the spill, the keys are divided into several *partitions* (as many as reduce tasks) using a partitioning function, and then the values of each key are sorted and written to its corresponding partition in the spill. An optional combiner function may be applied on the buffer data just before they are written into the spills. The objective of this function is to decrease the size of intermediate data that should be transferred to the reduce workers. Once a map task is completed, the generated spills are merged into a final output file and the master is notified.

In the reduce phase, each *partition* is assigned to one of the reduce tasks. Each reduce task retrieves the key-value pairs corresponding to its partition from all the map output files, and merges them using the merge-sort algorithm. The transfer of data from map workers to the reduce workers is called shuffling, and can be started when a map task finishes its work. However, the reduce function cannot be applied until all the map tasks have finished and their outputs merged and grouped. Each reduce task groups the values of the same key, applies the reduce function on the corresponding values, and generates the final output results. When, all reduce tasks of a job are completed successfully, the user is notified by the master.

2.2. An Abstract View

In an abstract view, the input of the map phase in MapReduce can be considered as a *set of data splits*, which should be processed by all workers. Thus, each map worker takes one split, processes it, and then takes another one until there are no splits in the set. Thus, we can hope for good parallelism in the map side, since no worker is idle until there is any split in the split set. But, in the reduce side, there is not such a parallelism, because the values of each key should be processed by one reduce worker. Thus, there may be situations where a high volume of the work is done by a single worker or a few number of workers, while the others are idle.

We have made FP-Hadoop more parallel and efficient than Hadoop since all reduce workers can contribute in reducing the map output even if the output belong to only one key.

3. FP-Hadoop Principles

In this section, we introduce the programming model of FP-Hadoop, its main phases, and the functions that are necessary for executing jobs. The design details of FP-Hadoop development are given in the next section.

3.1. Programming Model

In FP-Hadoop, the output of the map tasks is organized as a set of blocks (splits) which are consumed by the reduce workers. More specifically, the intermediate key-value pairs are dynamically grouped into splits, called *Intermediate Result Splits* (*IR splits* for short). The size of an IR split is bounded between two values, *MinIRSize* and *MaxIRSize*, configurable by the user. Formally, each IR split is a set of (k, V) pairs such that k is an intermediate key and V is a subset of the values generated for k by the map tasks.

FP-Hadoop executes the jobs in three different phases (see Figure 1): *map*, *intermediate reduce*, and *final reduce*. The map phase is almost the same as that of Hadoop in

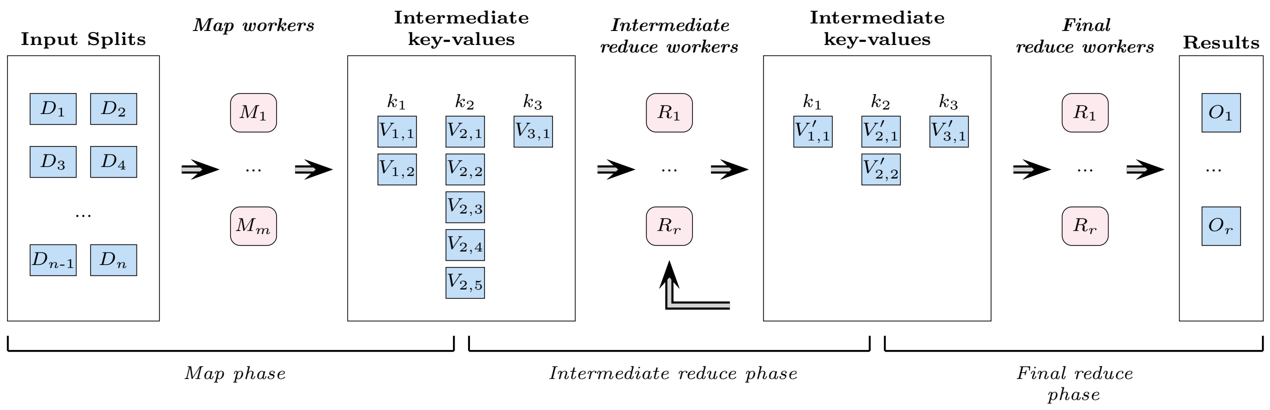

Input Splits
Map workers
Intermediate key-values
Intermediate reduce workers
Intermediate key-values
Final reduce workers
Results
D_1 D_2 D_3 D_4 ... D_{n-1} D_n
M_1 ... M_m
k_1 k_2 k_3
$V_{1,1}$ $V_{1,2}$ $V_{2,1}$ $V_{2,2}$ $V_{2,3}$ $V_{2,4}$ $V_{2,5}$ $V_{3,1}$
R_1 ... R_r
$V'_{1,1}$ $V'_{2,1}$ $V'_{2,2}$ $V'_{3,1}$
O_1 ... O_r
Map phase
Intermediate reduce phase
Final reduce phase

Figure 1: FP-Hadoop job processing scheme. The input of the intermediate reduce phase is a set of intermediate splits (blocks) which are generated dynamically using the intermediate data distributed over map workers. No intermediate reduce worker is idle until there is any intermediate split. The intermediate reduce phase can be done in several iterations.

the sense that the map workers apply the map function on the input splits, and produce intermediate key-value pairs. The only difference is that in FP-Hadoop, the map output is managed as a set of *IR fragments* that are used for constructing IR splits.

There are two different reduce functions: *intermediate reduce* (*IR*) and *final reduce* (*FR*) functions.

In the *intermediate reduce phase*, the IR function is executed in parallel by reduce workers on the IR splits, which are constructed using a scheduling strategy from the intermediate values distributed over the nodes. More specifically, in this phase, each ready reduce worker takes an IR split as input, applies the IR function on it, and produces a set of key-value pairs which may be used for constructing future IR splits. When a reduce worker finishes its input split, it takes another split and so on until there is no more IR splits. In general, programming the IR function is not very complicated; it can be done in a similar way as the *combiner function* of Hadoop. In Section 3.2, we give more details about the IR function, and how it can be programmed.

The intermediate reduce phase can be repeated in *several iterations*, to apply the IR function several times on the intermediate data and reduce incrementally the final splits consumed by the FR function (see Figure 2). The *maximum number of iterations* can be specified by the programmer, or be chosen adaptively, i.e., until the intermediate reduce tasks input/output size ratio is higher than a given threshold.

In the *final reduce phase*, the FR function is applied on the IR splits generated as the output of the intermediate reduce phase. The FR function is in charge of performing the final grouping and production of the results of the job. Like in Hadoop, the keys are assigned to the reduce tasks according to a partitioning function. Each reduce worker pulls all IR splits corresponding to its keys, merges them, applies the FR function on the values of each key, and generates the final job results. Since in FP-Hadoop the final reduce workers receive the values on which the intermediate workers have worked, the load of the final reduce workers in FP-Hadoop is usually much lower than that of the reduce workers in Hadoop.

In the next subsection, we give more details about the IR and FR functions, and explain how they can be programmed.

3.2. IR and FR Functions

To take advantage of the intermediate reduce phase, the programmer should replace his/her reduce function by intermediate and final reduce functions. Formally, the input and output of map (M), intermediate (IR) and final reduce (FR) functions is as follows:

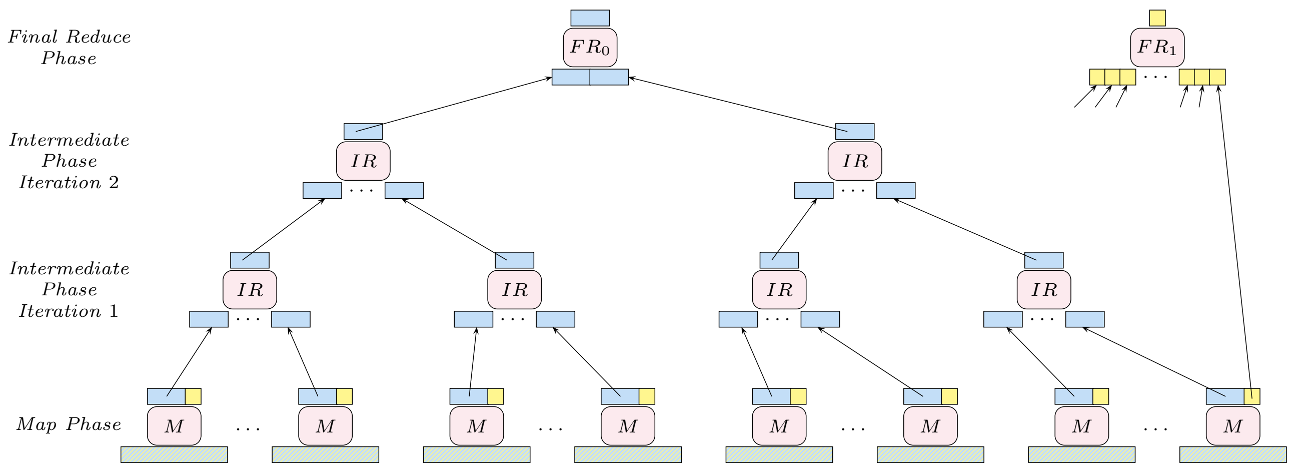
$$M : (\mathcal{K}_1, \mathcal{V}_1) \rightarrow list(\mathcal{K}_2, \mathcal{V}_2)$$
$$IR : (\mathcal{K}_2, partial\_list(\mathcal{V}_2)) \rightarrow (\mathcal{K}_2, partial\_list(\mathcal{V}_2))$$
$$FR : (\mathcal{K}_2, list(\mathcal{V}_2)) \rightarrow list(\mathcal{K}_3, \mathcal{V}_3)$$

Notice that in IR function, any partial set of intermediate values can be received as input. However, in FR function, all values of an intermediate key are passed to the function.

Given a reduce function, to write the IR and FR functions, the programmer should separate the sections that can be processed in parallel and put them in IR function, and the rest in FR function. Formally, given a reduce function R, the programmer should find two functions IR and FR, such that for any intermediate key k and its list of values S, we have:

$$R(k, S) = FR(k, \langle IR(k, S_1), ..., IR(k, S_n)\rangle)$$
$$\text{for every partition } S_1 \cup ... \cup S_n = S$$

In Table 1, we enumerate some important functions, and show their intermediate and FR functions. There are many functions for which we can use the original reduce



Final Reduce Phase
FR_0
FR_1
Intermediate Phase Iteration 2
Intermediate Phase Iteration 1
IR
Map Phase
M

Figure 2: Example of hierarchical job execution in FP-Hadoop. In this example, there are two intermediate keys, and the intermediate values are shown by blue (for key1) and yellow (for key2) colors. The intermediate values of key2 (yellow) are reduced directly by a final reducer just after the map phase (i.e. without an intermediate phase), because their size is small. But those of key1 (blue) are processed in two iterations in the intermediate phase. For planning such hierarchical executions, it is sufficient to set a maximum number of iterations, e.g. more than 2, and then everything is done automatically by FP-Hadoop.

Table 1: Examples of some reduce functions and their equivalent intermediate and final functions

Reduce (R) Function	**Intermediate (*IR*) and Final (*FR*) Functions**
R = TOP-K	*IR* = *FR* = TOP-K
R = SKYLINE	*IR* = *FR* = SKYLINE
R = UNION	*IR* = *FR* = UNION
R = SUM	*IR* = *FR* = SUM
R = MAX	*IR* = *FR* = MAX
R = MIN	*IR* = *FR* = MIN
R = AVG	*IR* = (SUM, COUNT), *FR* = (SUM, COUNT) THEN SUM/COUNT

function both in the intermediate and final reduce phases, i.e., we have $IR = FR = R$. Examples of such functions are TOP-K, SKYLINE, UNION and SUM.

The following example shows how a Top-k query can be implemented in FP-Hadoop using the same function for IR and FR.

Example 2. TOP-K. Consider a job that given a scoring function computes the top-k tuples of a big table. In this job, the map function computes the score of each read tuple, and emits `(key, {tupleID, score})`, where key is the identifier of the set of values on which we want to find the top-k values. Then, both IR and FR functions can be implemented as a function that, given a set of `{tupleID, score}` pairs, returns the k pairs that have the highest scores. Thus, in practice, the intermediate reduce workers generate partial top-k results (top-k tuples of their input IR splits), and the FR function produces the final results from intermediate partial results. If the value of k is big, e.g. 1% of the input as in our motivating example in Introduction, then we may need more iterations in the intermediate phase to reduce the intermediate data more and more. In this case, the execution of the job for overloaded key(s) can be hierarchical as in Figure 2. To plan such executions, it is sufficient to set the maximum number of iterations to the required level, and then FP-Hadoop organizes multiple iterations for overloaded keys when it is needed .

A class of functions that can usually take advantage of the intermediate reduce phase is that of aggregate functions. Examples of such functions are SUM, MIN/MAX (using the same function as IR and FR), AVG and STD (using different functions for IR and FR). Aggregate functions can be classified into three groups [7]:

- **Definition 1 (Distributive).** *Let F be a reduce function, and S a set of values, and $S_1 \cdot ... \cdot S_n$ a partitioning over S. F is* distributive *if there is a function G such that $F(S) = G(\langle F(S_1), ..., F(S_n)\rangle)$.*
- **Definition 2 (Algebraic).** *Let F be a reduce function, and S a set of values, and $S_1 \cdot ... \cdot S_n$ a partitioning over S. F is* Algebraic *if there is a m-valued function G and a function H such that $F(S) = H(\langle G(S_1)_1, ..., G(S_n)_1\rangle, ..., \langle G(S_1)_m, ..., G(S_n)_m\rangle)$.*

- **Definition 3 (Holistic).** *Let F be a reduce function, and S a set of values, and $S_1 \cdot ... \cdot S_n$ a partitioning over S. F is* Holistic *if there is not a m-valued function (with bounded m) that characterizes the computation. In other words, F is holistic if it is neither distributive nor algebraic.*

Intuitively, a function is *distributive* if it can be computed in a distributed manner. Examples of such functions are COUNT, SUM, MIN, and MAX. Distributive functions can particularly benefit from the usage of intermediate reduce tasks, as they usually reduce the data size significantly. *Algebraic* functions are the functions that can be computed by using a bounded number of distributive functions. Examples of algebraic functions are `AVG` and STD. These functions can also take advantage of intermediate reduce tasks. In *holistic* functions, there is no constant bound on the storage size needed to describe a sub-aggregate. Some holistic functions, such as SKYLINE, may take great advantage of running an intermediate phase. For some other holistic functions, it may be difficult to find an efficient intermediate function.

We precise that if it is difficult to find an efficient IR function for a job, it is sufficient to specify no IR function, then the final reduce phase starts just after the map phase, i.e., like in Hadoop.

4. Design Details

In this section, we describe our design choices in FP-Hadoop. We focus on the activities that do not exist in Hadoop or are very different in FP-Hadoop, particularly: 1) management of IR fragments used for making IR splits; 2) intermediate task scheduling; 3) intermediate and final reduce task creation; 4) management of multiple iterations in the intermediate reduce phase.

4.1. IR Fragments for Constructing IR Splits

In this subsection, we describe our approach for constructing the IR splits that are the working blocks of the reducers in the intermediate reduce phase.

In Hadoop, the output of the map tasks is kept in the form of temporary files (called *spills*). Each spill contains a set of partitions, such that each partition involves a set of keys and their values. These spills are merged at the end of the map task, and the data of each partition is sent to one of the reducers.

In FP-Hadoop, the spills are not merged. Each partition of a spill generates an *IR fragment*, and the IR fragments are used for making IR splits. When a spill is produced by a map task, the information about the spill's IR fragments, which we call *IRF metadata*, is sent to the master node by using the heartbeat message passing mechanism[4]. An IR fragment is uniquely identified by the task which produced it, its spill number and the partition. The data flow between FP-Haddop components is shown in Figure 3.

Notice that the choice of using spills as the output unit of the map phase provides the following advantages. The intermediate phase may start as soon as sufficient spills have been produced, even if no map task has yet finished, thus increasing the parallelism. Moreover, the fact that spills are bounded in size guarantees that IR split maximum size is respected. This would not be the case if the whole output of a map task is used, since its size is unbounded and could be by itself bigger than the IR split limit.

For keeping IRF metadata, the master of FP-Hadoop uses a specific data structure called *IR fragment table (IRF Table)*. Each partition has an entry in IRF Table that points to a list keeping the IR fragment metadata of the partition, e.g., size, spill and the ID of the map worker where the IR fragment has been produced. The master uses the information in IRF Table for constructing IR splits and assigning them to ready reduce workers. This is done mainly based on the scheduling strategies described in the next subsection.

4.2. Scheduling Strategies

The master node is responsible for scheduling the intermediate and final reduce tasks. For this, it uses a component called *Reduce Scheduler* that schedules the tasks by using a customizable scheduling strategy. Here we describe the scheduling strategies which are used in FP-Hadoop; we present the details of *Reduce Scheduler* in Section 4.4.

For scheduling an intermediate reduce task, the most important issue is to choose the IR fragments that belong to the IR split that should be processed by the task. The strategies, which are currently implemented in FP-Hadoop are:

- *Greedy*: In this strategy, IR fragments are chosen from within the biggest partition, i.e., the partition whose IR fragments account for the highest total size. In the IRF Table, for each partition, along with the list of IR fragments we store its total size, allowing to chose the partition by a simple vertical scan of IRF Table. After choosing the partition, we select IR fragments starting from the head of the list until reaching the *MaxIRSize* value, i.e., the upper bound size for an IR split. This strategy is the default strategy in FP-Hadoop.

- *Locality-aware*: In this strategy, the objective is to choose for a worker w the IR fragments that are on its local disk or close to it. Consequently, we chose the partition p for which the IR fragments produced in w account for the biggest size, provided that p's total size is as least *MinIRSize*. From the partition's fragments, we select those local to w until reaching

[4]This mechanism is used for communication between the master and workers

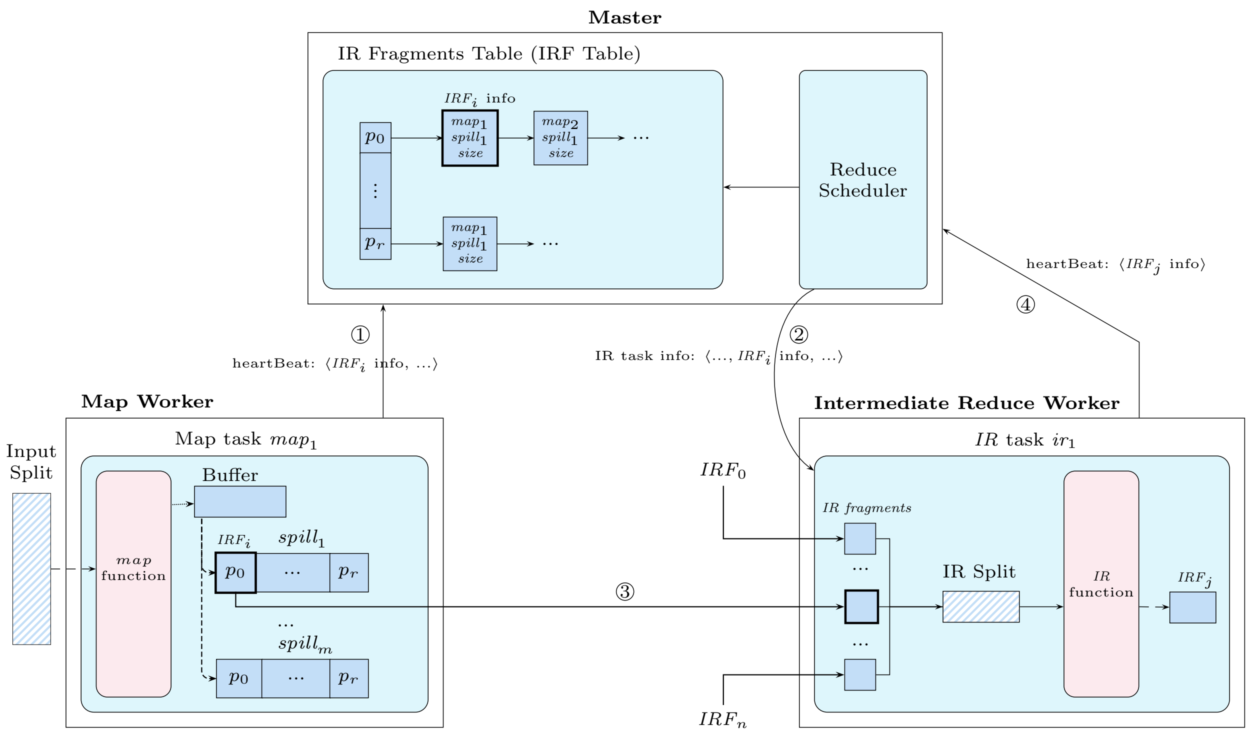


Figure 3: Data flow between a map worker, the master and an intermediate reduce worker. The communicated messages are shown in their sent order.

MaxIRSize. If their total size is inferior to *MinIR-Size* we keep taking fragments until reaching *MinIR-Size*, first choosing from those produced in the same rack as w and then from the same data center.

4.3. Multiple Iterations

FP-Hadoop can be configured to execute the intermediate reduce phase in several iterations, in such a way that the output of each iteration is consumed by the next iteration. Notice that the output of IR tasks in phase n produces the IR fragments that are to be consumed in phase $n + 1$.

An example of Top-k query execution over Wikipedia data (described in Section 5) is shown in Figure 4, where each phase is depicted as a rectangle whose height represents its input size and its length the execution time in seconds. In this example, there are two iterations for the intermediate reduce phase (shown with gray rectangles). We can see how each iteration further reduces the intermediate data size so that when the final reduce phase (in blue) is executed, its input size is sufficiently small.

In FP-Hadoop, there is a parameter *MaxNumIter* that defines the maximum number of iterations. Notice that this parameter just establishes the maximum number, but in practice each partition may be processed in a different number of iterations, for instance depending on its size, input/output ratio or skew.

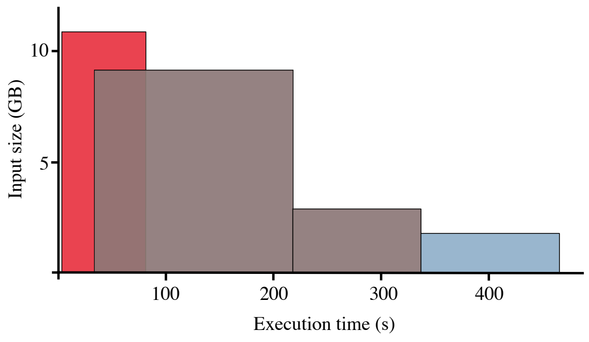


Figure 4: Input size of different phases in multiple iterations example

FP-Hadoop provides the following strategies for configuring the number of iterations:

- *Size-based:* In this approach, an iteration is launched only if its input size is more that a given threshold, *MinIterSize*. By default, FP-Hadoop uses the same value as *MinIRSize* (i.e. the minimum size of IR splits). The actual number of iterations may vary among partitions, and the maximum number may not be reached, e.g. if the size of intermediate data was originally small or has been sufficiently reduced during previous iterations. The extreme scenario corresponds to the case when the map output for a given partition is less than *MinIterSize*. In this case, the final reduce phase is directly launched.

- *Input/Output-Ratio-based:* In this approach, an iteration is launched when the ratio between the input and output size of the previous iteration is greater than a system parameter *IORatio*. The rationale is that if the input/output ratio is lower than some value, then further intermediate iterations will not help reducing the size of the partition in consideration, and hence, will not improve the job execution performance.

- *Skew-based:* In this approach, an intermediate iteration is launched for a given partition if its size is *SkewRatio* times higher than the average partition size. Thus it only executes more iterations for the overloaded partitions. For example by setting *SkewRatio* equal to 2, only the partitions whose size is at least twice the average size are consumed in a new iteration, and the others are sent to the final reduce phase.

4.4. Reduce Scheduler

To schedule the intermediate and final reduce tasks, the master node uses a component called *Reduce Scheduler*. The scheduling is done using a generic algorithm whose pseudo-code is shown in Algorithm 2. The algorithm scans *IRF Table*, and selects the *best partition* for constructing the IR split which should be consumed by the worker. Then, based on the size of the data in the partition and the maximum number of iterations, it decides whether to launch a final reduce task or an intermediate task. If it decides to launch a final reduce task, it uses all IR fragments of the partition. In the other case, i.e., an intermediate reduce task, it chooses the fragments of the IR split from the best partition based on the scheduling strategy.

The main functions that are used in the generic scheduling algorithm (Algorithm 2) are as follows:

- *isBestPartition().* Based on the scheduling strategy, this function selects a partition from which the IR fragments of the task will be chosen. For example, with the *Greedy* strategy, it selects the partition that has the biggest data size among the partitions satisfying the scheduling constraints, i.e., their total size is at least *MinIRSize*, and the number of fragments of the partition are at least *MinIRFs*, which can be configured by the user.

- *shouldRunFR().* This function decides if the final reduce task should be launched for the data of a partition. The decision depends on the *number of iterations* that should be executed in the intermediate reduce phase. If the current iteration for a partition is the last iteration to be done, and the tasks of the iteration have finished successfully, the final reduce task can be scheduled.

```
Input:  $w$: Worker with the idle slot, $P$: set of
        partitions
Result: $t$: Task to run
begin
    $chosen\_Partition \leftarrow \emptyset$
    for each $p \in P$ do
        if $isBestPartition(p, chosen\_Partition, w)$ then
            $chosen\_Partition \leftarrow p$
        end
    end
    if $shouldRunFR(chosen\_Partition)$ then
        $t \leftarrow create\_FR\_Task(chosen\_Partition)$
    else
        $t \leftarrow create\_IR\_Task(chosen\_Partition)$
    end
    return $t$
end
```
Algorithm 2: Generic scheduling algorithm

- *create_FR_Task().* This function creates a final reduce task. It simply uses all the IR fragments from the partition given by *isBestPartition* function.

- *create_IR_Task().* This function creates an intermediate reduce task. It takes the partition given by *isBestPartition* function, and chooses a set of IR fragments as the IR split of the task. The fragments are chosen based on the scheduling strategy, for instance local fragments would be favored in the locality-aware strategy, whereas in the basic Greedy scheduling strategy, IR fragments are added to the task's IR split while they do not surpass *MaxIRSize*. The task metadata are sent to the ready worker (see Figure 3).

To implement new scheduling strategies in Reduce Scheduler, usually it is sufficient to change the above functions. For instance, to implement *locality-aware* strategy instead of *Greedy* strategy, we changed isBestPartition function (that selects the partition to use) in such a way that it gives the priority to the partition that has the maximum IR fragments generated in the ready worker, and then changed generateIntermediateRT function to choose the IR fragments accordingly, e.g., favor local fragments.

4.5. Fault Tolerance

In Hadoop, the master marks a task as failed either if an error is reported (e.g., an Exception of the program or a sudden exit of the JVM) or if it has not received a progress update from the task for a given period [1]. In both cases, the task is scheduled for re-execution, if possible in a different node. During the execution of a job, the output of completed map tasks should be kept

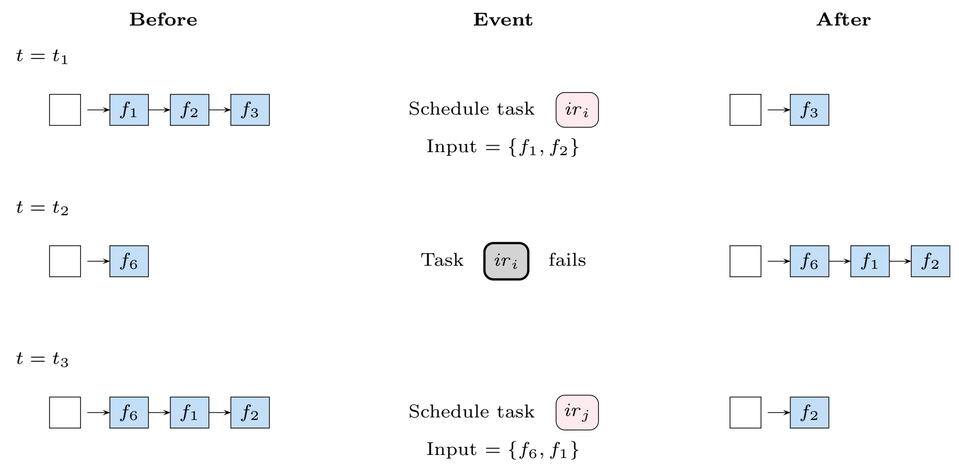
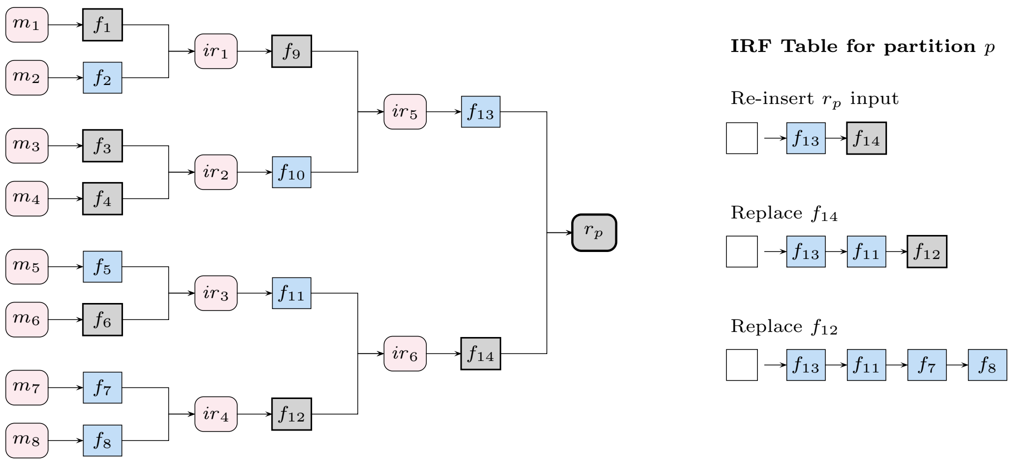


Figure 5: Task failure example.



m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8
f_1 f_2 f_3 f_4 f_5 f_6 f_7 f_8
ir_1 ir_2 ir_3 ir_4
f_9 f_{10} f_{11} f_{12}
ir_5 ir_6
f_{13} f_{14}
r_p
IRF Table for partition p
Re-insert r_p input
f_{13} f_{14}
Replace f_{14}
f_{13} f_{11} f_{12}
Replace f_{12}
f_{13} f_{11} f_7 f_8

Figure 6: Worker failure example.

available in the workers' disks until the job finishes. The reason is that any reduce task reads data from all map tasks' output. If a worker fails, intermediate data stored on its disk is not available any more. Consequently, in addition to running tasks, all successfully completed map tasks executed in the worker need to be rescheduled to make their output available again.

In FP-Hadoop, the behavior in the case of failure is slightly different, because we have to consider intermediate reduce tasks and we can also profit from the fact that they only read data from a subset of the map tasks. As in Hadoop, we distinguish two scenarios: *task failure* and *worker failure*.

4.5.1. Task Failure

For map and final reduce tasks, the behavior is exactly the same as in native Hadoop, that is, the failed task is scheduled for re-execution, if possible in a different node.

In the case of IR tasks, we need to re-inject their input IR fragments metadata into the IRF Table. Then, the fragments will be assigned to another task, not necessarily grouped in the same IR Split. Since new IRFs may be available, the reduce scheduler may choose to combine them in different ways.

The case of an IR task failure is illustrated in the example shown in Figure 5. The state of the IRFs list corresponding to a given partition p in the IRF Table is presented before and after each event. In time $t = t_1$ task ir_i is scheduled with input fragments f_1 and f_2. Then, in time $t = t_2$ the task fails and, consequently, its input fragments are injected back in the list. Notice that since t_1 some fragments have already been consumed (e.g. f_3) and new fragments inserted (e.g. f_6). Finally, when an idle worker asks for a new task in time $t = t_3$, fragments are grouped differently and its input is composed of fragments f_6 and f_1.

4.5.2. Worker Failure

If a worker node fails, if in the node there were running tasks, they need to be re-scheduled. Map tasks are simply re-executed and the input fragments of intermediate and final reduce tasks are re-injected into the IRF Table.

In FP-Hadoop, as opposed to native Hadoop, not all completed tasks' output is needed in the future, since their data may have already been consumed and reduced by intermediate reduce tasks in subsequent iterations of the execution tree. Indeed, FP-Hadoop uses a mechanism that only requires the re-execution of a minimal amount of tasks in the case of a worker failure.

Each IR fragment stores within its metadata the information about their provenance, that is, a reference to the fragments that were used on its generation. In the failure of a worker, just after input IR fragments of running tasks are reinserted to the IRF Table, each fragment stored in the failed node is replaced by the fragments that were used on its construction. If within those fragments, there are some fragments that are stored in the failed node, they are replaced by their predecessors as well. Fragments are replaced recursively by their predecessors until they are available or they have been generated in a map task. Only in that case, the map task is scheduled for re-execution. Re-injected fragments can be consumed by new IR tasks following the standard procedure.

Let's illustrate this with the example of Figure 6, in which the execution tree for a given partition p is shown. Reduce task r was running in the worker node when it failed. IR fragments stored in the failed node are shadowed. The input fragments f_{12} and f_{14} are re-inserted to the IRF Table as in a single task failure. Then, unavailable fragments are replaced recursively by their predecessors. In this way, f_{14} is replaced by f_{11} and f_{12} and then f_{12} by f_7 and f_8. No map task needs to be re-executed and the re-injected fragments can be consumed again by new tasks.

FP-Hadoop recovery strategy only re-processes the data that is needed to finish the execution of the job, as opposed to native Hadoop, which needs to re-execute all map tasks and non-finished reduce tasks executed in the failed worker. Furthermore, as IR tasks have already reduced the size of the intermediate data, the amount of data to be treated is reduced even more, thus accelerating the recovery process.

5. Performance Evaluation

We implemented a prototype of FP-Hadoop by modifying Hadoop's components. In this section, we report on the results of our experiments for evaluating the performance of FP-Hadoop [5]. We first discuss the experimental setup such as the datasets, queries and the experimental platform. Then, we discuss the results of our tests done to study the performance of FP-Hadoop in different situations, particularly by varying parameters such as the number of nodes in the cluster, the size of input data, etc.

5.1. Setup

We run the experiments in the Grid500 platform[6] in a cluster with up to 50 nodes. The nodes are provided with Intel Quad-Core Xeon L5335 processors with 4 cores each, and 16GB of RAM. Nodes are connected through a switch providing a Gigagbit ethernet connection to each node.

We compare FP-Hadoop with standard Apache Hadoop and SkewTune [3] which is the closest related work to ours (see a brief description in Related Work Section).

In all our experiments, we use a *combiner function* (for Hadoop, FP-Hadoop and SkewTune) that is executed on the results of map tasks before sending them to the reduce tasks. This function is use to decrease the amount of data transferred from map to reduce workers, and so to decrease the load of reduce workers.

The results of the experiments are the average of three runs. We measure two metrics:

- *Execution time.* This is the time interval (in seconds) between the moment when the job is launched and the moment when it ends. This is our default metrics reported for most of the results.
- *Reduce time.* We use this metric to consider the time that is used only for shuffling and reducing. It is measured as the time interval between the moment when the last map task is finished and the end of the job.

With respect to Hadoop's configuration, the number of slots was set to the number of cores. All the experiments are executed with a number of reduce workers equal to the number of machines. We change `io.sort.factor` to 100, as advised in [6], which actually favors Hadoop. For the rest of the parameters, we employ Hadoop's default values.

The default values for the parameters which we employ in our experiments are as follows. The default number of nodes which we use in our cluster is 20. Unless otherwise specified, the input data size in the experiments is 20 GB.

In FP-Hadoop, we use the default Greedy scheduling strategy as the high throughput of the network limits the impact of the locality-aware strategy. The default value for *MinIRSize* is set to 512 MB. The value of *MaxIRSize* is always twice as that of *MinIRSize*, and the maximum number of iterations is set to 1.

5.2. Queries and Datasets

We use the following combinations of MapReduce jobs and datasets to assess the performance of our prototype:

[5]The executable code of FP-Hadoop as well as the tested jobs are accessible in the following page: `http://gforge.inria.fr/plugins/mediawiki/wiki/fp-hadoop/`

[6]`http://www.grid5000.fr`

Top-k% (TK). This job, which is our default job in the experiments, corresponds to the query from the Wikipedia example described in the introduction of the paper. The input dataset is stored in the form of lines with the schema:

VISITS(*language*, *article*, *num_views*, *other_data*)

Our query consists on retrieving for each language the k% most visited articles. The default value of *k* is 1, i.e., by default the query returns 1% of the input data. We have used real-world and synthetic datasets. The real-world dataset (TK-RD) is obtained from the logs about Wikipedia page visits[7] consisting of a set of files each containing the statistics collected for a single hour. We also produced two synthetic datasets, in which we can control the number of keys and their skew. In the first synthetic dataset (TK-SK), which is the default dataset, the number of articles per language follows a Zipfian distribution function

$$f(l, S, N) = \frac{1/l^s}{\sum_{n=1}^{N}(1/n^s)}$$

that returns the frequency of rank l, where S and N are the parameters that define the distribution, i.e., $f(1, S, N)$ returns the frequency of the most popular language. The default value for Zipf exponent parameter (S) is 1, and the parameter N is equal to the number of languages (10 by default). In the second synthetic dataset (TK-U), the articles are uniformly distributed in the keys.

We perform several tests varying the data size, among other parameters, up to 120GB. The query is implemented using a secondary sort [6], where intermediate keys are sorted first by language and then by the article's number of visits, but only grouped by language.

Inverted Index (II). This job consists of generating an inverted index with the words of the English Wikipedia articles[8], as in [3]. We use a RADIX partitioner to map letters of the alphabet to reduce tasks and produce a lexicographically ordered output. We execute the job with a dataset containing 20GB of Wikipedia articles.

PageRank (PR). This query applies the PageRank [8] algorithm to a graph in order to assign weights to the vertices. As in [3] we use the implementation provided by Cloud9[9]. And as dataset, we use the PLD graph from Web Data Commons[10] whose size is about 2.8GB.

Wordcount (WC). Finally, we use the wordcount job provided in Hadoop standard framework. We apply it to a dataset generated with the `RandomWriter` job provided in the Hadoop distribution. We test this job with a 100GB dataset.

[7]http://dumps.wikimedia.org/other/pagecounts-raw/

[8]http://dumps.wikimedia.org/enwiki/latest/

[9]http://www.umiacs.umd.edu/ jimmylin/Cloud9/docs/index.html

[10]http://webdatacommons.org/hyperlinkgraph/

5.3. Scalability

We investigate the effect of the input size on the performance of FP-Hadoop compared to Hadoop. Using TK-SK dataset, Figures 7b and 7a show the reduce time and execution time respectively, by varying the input size up to 120 GB, *MinIRSize* set to 5 GB, and other parameters set as default values described in Section 5.1. Figures 7c and 7d show the performance using TK-RD dataset with sizes up to 100 GB, while other parameters as default values described in Section 5.1. As expected, increasing the input size increases the execution time of both Hadoop and FP-Hadoop, because more data should be processed by map and reduce workers. However, the performance of FP-Hadoop is much better than Hadoop when we increase the size of input data. For example, in Figure 7a, the speed-up of FP-Hadoop vs Hadoop on execution time is around 1.4 for input size of 20GB, but this gain increases to around 5 when the input size is 120GB. For the latter data size, the reduce time of FP-Hadoop is more than 10 times lower than Hadoop.

The reason for this significant performance gain is that in the intermediate reduce phase of FP-Hadoop the reduce workers collaborate on processing the values of the keys containing a high number of values while in Hadoop a single task has to process all this data. This is illustrated in Figures 7e and 7f, where we compare the execution time of the longest reduce tasks of Hadoop and FP-Hadoop for TK-RD and 20GB. We can see that the longest task in Hadoop is the responsible of the poor performance in the reduce phase, which explains the total execution time. In FP-Hadoop, the longest task is 4 times shorter, and this is a consequence of the improved parallelism.

5.4. Effect of Cluster Size

We study the effect of the number of nodes of the cluster on performance. Figure 8a shows the execution time by varying the number of nodes, and other parameters set as default values described in Section 5.1. Increasing the number of nodes decreases the execution time of both Hadoop and FP-Hadoop. However, FP-Hadoop benefits more from the increasing number of nodes. In Figure 8a, with 5 nodes, FP-Hadoop outperforms Hadoop by a factor of around 1.75. However, when the number of nodes is equal to 50, the improvement factor is around 4. This increase in the gain can be explained by the fact that when there are more nodes in the system, more nodes can collaborate on the values of hot keys in FP-Hadoop. In opposition, in Hadoop, although using higher number of nodes can decrease the execution time of the map phase, it cannot significantly decrease the reduce phase time, in particular if there are intermediate keys with high number of values.

5.5. Effect of the Number of Intermediate Keys

In our tests, we use the attribute language as the intermediate key for grouping the intermediate values. Here,

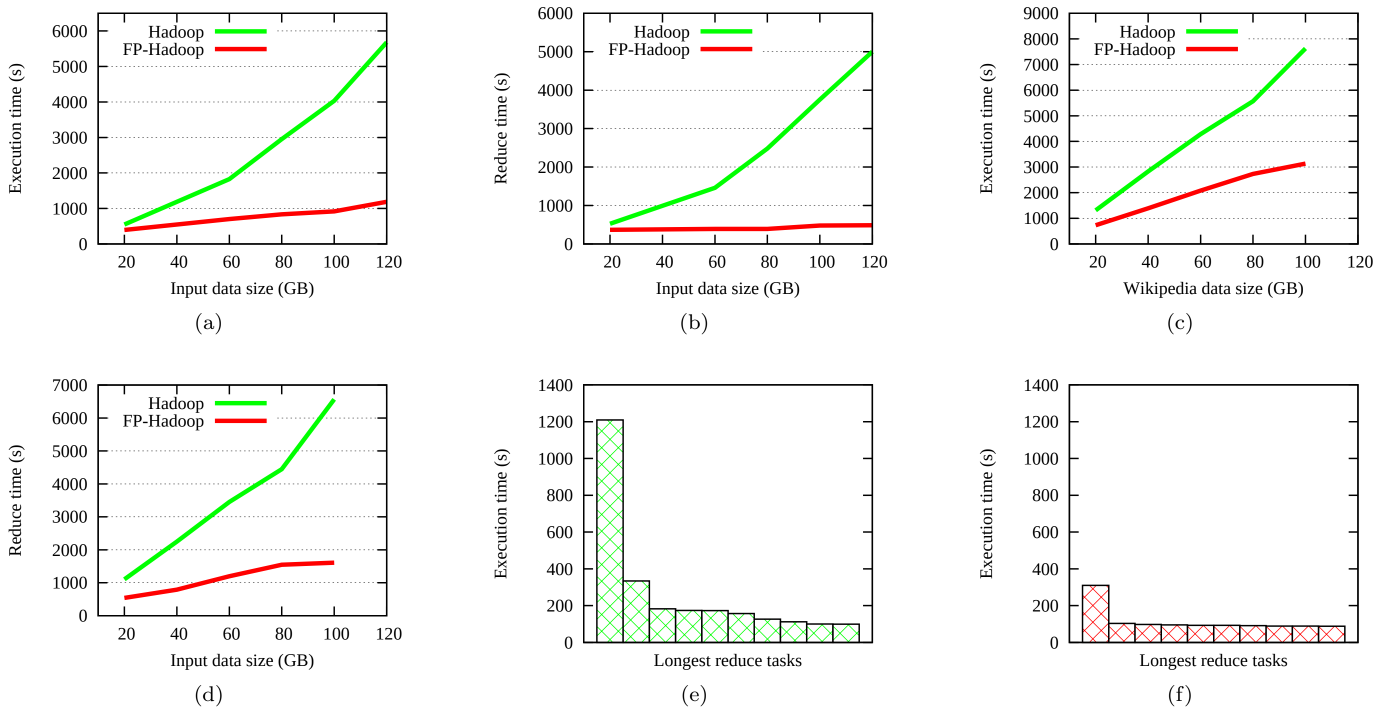


Figure 7: Scalability of FP-Hadoop: With TK-SK (a) reduce time and (b) execution time; with TK-RD (c) execution time, (d) reduce time and longest reduce tasks with 20GB for (e) Hadoop and (f) FP-Hadoop.

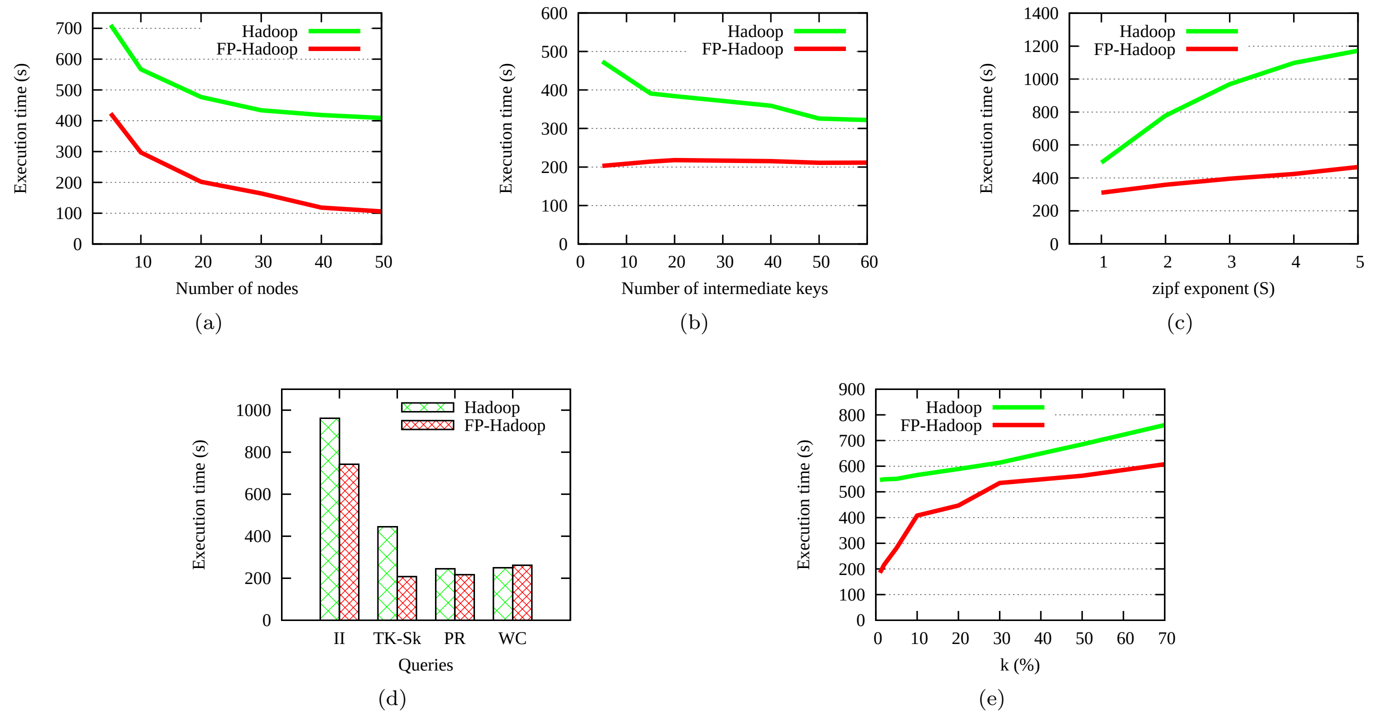


Figure 8: Effect of several parameters on the performance of FP-Hadoop: (a) number of intermediate keys, (b) zipfian exponent, (c) data skew and queries, (d) cluster size, (e) intermediate data filtering

we report the results of our experiments done to study the effect of this parameter on the performance of FP-Hadoop and Hadoop.

Figure 8b shows the execution time with varying the number of keys (languages) from 50 to 5, and other parameters set as default values. When the number of keys is lower than 20, Hadoop cannot take advantage of all available reduce nodes (there were 20 cluster nodes in this test). However, in FP-Hadoop, the intermediate reduce workers can contribute in processing the values of the keys that have high numbers of values. This is why there is a significant difference between the execution time of FP-Hadoop and Hadoop in these cases.

Even for the cases where the number of keys is higher than the number of nodes (i.e., 20), the execution time of FP-Hadoop is better than that of Hadoop, because in these cases, there are keys with high number of values, and Hadoop cannot well balance the load of reduce workers.

For Hadoop, when increasing the number of keys, the execution time decreases until some number of keys and then it becomes constant. We performed our tests with up to 200 keys, and observed that after 60 keys, there is no decrease in the execution time of Hadoop.

5.6. Effect of High Skew in Reduce Workers Load

Here, we study the effect of high data skew on performance by varying the zipf exponent (S) used for generating synthetic datasets with Zipfian distribution. The higher is S, the higher is the skew in the size of the data (articles) assigned to the keys (languages).

Figure 8c shows the execution time with varying zipf exponent S from 1 to 5, and other parameters set as default values. The figure shows that the data skew has a big negative impact on the performance of Hadoop, but its impact on FP-Hadoop is slight. The gain factor of FP-Hadoop increases by increasing S. The reason is that by increasing S, there will be intermediate keys with higher number of intermediate values, and these keys are the bottlenecks in Hadoop, because the values of each key are processed by only one reduce worker.

5.7. Effect of Data Skew on Different Queries

Figure 8d shows the total execution time of FP-Hadoop by using several queries and their corresponding data as described in Section 5.1. The results show that FP-Hadoop outperforms Hadoop, in all cases except for the word-count query (WC).

The extent of the gain depends on the amount of reduction that can be performed in the intermediate phase as a result of the partial aggregation. This explains while the best case is for TK-Sk, where each IR task can reduce the data back to k tuples and also while the reduction in PR is small, since only the mass contributions can be aggregated while the node structure has to be passed along untouched.

For the word-count query, the execution time of FP-Hadoop is a little bit (3%) higher than Hadoop, and this increase corresponds to the overhead of the intermediate reduce phase. Indeed, in this query, there is no skew in the reduce side, because the partitioner can balance well the reduce load among workers. FP-Hadoop spends some time to detect that there is no skew in the intermediate results, and then launches the final reduce phase. Thus, its execution time is slightly higher than Hadoop. Notice that even this small overhead can be avoided by disabling the intermediate reduce phase.

5.8. Effect of Intermediate Data Filtering

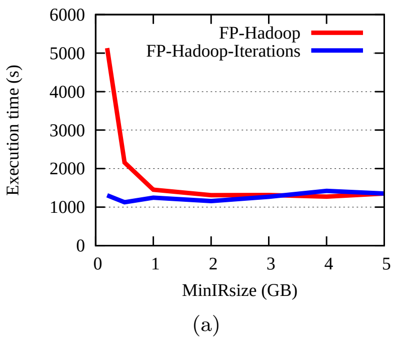
Using parameter k in the top-k% query, we can control the amount of data that may be filtered by the intermediate reduce tasks. In fact, in the top-k% query we return k% of the data that have the highest values. Thus, in the output of a map task or input data of an intermediate reduce task, if the amount of data is higher than k% of the total data, then we can keep the k% highest ranked data and filter the rest. Therefore, the lower is k, the higher is the capacity of data filtering by intermediate reduce tasks.

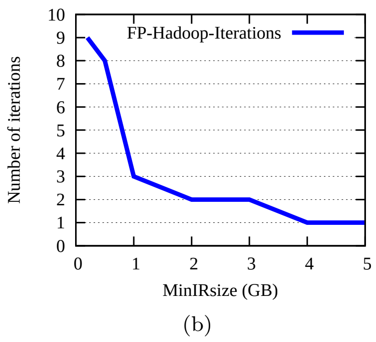
Figure 8e shows the performance of the two systems by varying k in top-k% query and other parameters set as default values. The figure shows that the lower is k, the better is the performance gain of FP-Hadoop. The reason is that with lower k values, the intermediate reduce tasks can filter more intermediate values. In particular, for values lower than 10%, FP-Hadoop can profit well from the filtering in intermediate reduce workers. For values higher than 10%, the data filtering by intermediate reduce workers decreases significantly, this is why the gain of FP-Hadoop reduces. However, even for k values higher than 10%, there is a significant difference between the execution time of FP-Hadoop and Hadoop.

5.9. Analysis of FP-Hadoop Parameters

We investigate the effect of the *MinIRSize* parameter on the performance of FP-Hadoop. For this, we use two configurations: 1) *FP-Hadoop*: the default configuration described in Section 5.1, with one iteration in the intermediate reduce phase; 2) *FP-Hadoop-Iterations*: in this configuration, we set the maximum number of iterations to be 10. As discussed in Section 4.3, the real number of iterations may be lower than this maximum value, e.g. because in an iteration the size of a partition may be lower than *MinIRSize*.

By varying *MinIRSize*, Figure 9a shows the execution time of FP-Hadoop and FP-Hadoop-Iterations by varying the *MinIRSize* parameter for processing an input of size 100GB, and other parameters set as default values. The results show that in FP-Hadoop with one iteration, when we set *MinIRSize* to small values (e.g., lower than 128MB), the execution time is not very good. This suggests to not choose very small values for *MinIRSize*, when using only one iteration. The reason is that in these cases, we cannot well take advantage of the one iteration in the intermediate reduce phase, since with very small IR Splits the amount





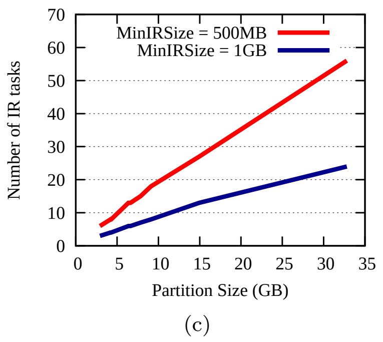


Figure 9: Analysis of FP-Hadoop parameters: (a) *MinIRSize* and iterations, (b) number of performed iterations (c) number of IR tasks.

of data that can be filtered by intermediate tasks may not by large.

However, by using more iterations in FP-Hadoop-Iterations, even with small *MinIRSize* values, the response time is good, since it uses more iterations to take advantage of the intermediate phase. Figure 9b shows the number of iterations done by FP-Hadoop-Iterations, when using different *MinIRSize* values. We observe that as *MinIRSize* increases, less iterations are needed to complete the intermediate phase.

In our experiments, when using FP-Hadoop with one iteration, the best configuration for *MinIRSize* is in the cases where it is close to the ratio of the map output size over the number of reduce workers.

We also study the number of IR tasks launched by the Reduce Scheduler. Figure 9c depicts the number of scheduled IR tasks vs. the partition sizes for TK-SK with 100GB with *MinIRSize* of 500MB and 1GB. The relation is clear: the bigger the partition, the higher the number of scheduled IR tasks. This is an expected behavior of FP-Hadoop, which tries to fully take advantage of parallelism for the overloaded partitions.

5.10. Comparison with SkewTune

Here, we compare FP-Hadoop with SkewTune [3]. Figures 10a and 10b show the reduce time and execution time of both approaches, using the data and parameters described in Section 5.1. For these experiments, we downloaded the SkewTune prototype that is publicly accessible[11]. The data which we used are the default data and sizes described in Section 5.1 (e.g. 20GB of data for TK-SK). As the results show, FP-Hadoop can outperform SkewTune with significant factors. This is particularly noticeable for TK-SK, where SkewTune cannot divide the execution of the most popular key into several tasks.

5.11. Effect of Map Split Size

As discussed in Introduction of this paper, in order to use the *combiner function* to decrease the reduce skew in the jobs such as Top-k%, we need to increase the size of the map splits significantly, giving more chance to the combiner to filter the intermediate data. In this section, we study the effect of increasing the map split size on performance of Hadoop and FP-Hadoop. Notice that in our previous experiments for all compared systems, we used the *combiner function*, and the default map split size was 64MB which was the default value in Hadoop. Figures 11a and 11b show the execution times for top-k% and II queries, respectively. We varied the map split size from 64MB to 4GB, and other parameters set as default values described in Section 5.1. In order to do so, we reinjected the input data with the corresponding file system block size, as in Hadoop the default behavior creates a map split for each file block.

For top-k% query, we observe a slight improvement in performance at the beginning (until the size of 256MB) for both Hadoop and FP-Hadoop. But, the performance degrades significantly for higher split sizes. For the inverted index (II) query, increasing the map split size has also a negative impact. The reason is that although increasing the size of the splits in the map phase may increase the chance of the *combiner function* to filter more intermediate data, using big size splits increases significantly the execution time of this phase, particularly because less map workers may participate in the map phase. Thus, the conclusion is that the combiner function can not make Hadoop as efficient as FP-Hadoop in processing high skewed data (e.g. in top-k% query) , even by using big size map splits.

5.12. Discussion

Overall the performance results show the effectiveness of FP-Hadoop for dealing with the data skew in the reduce side. For example, the results show that for 120GB of input data, FP-Hadoop can outperform Hadoop with factors of 10 and 5 in reduce time and total execution time respectively. This gain increases when augmenting the data size. The results also show that increasing the number of nodes of the cluster can significantly increase the gain of FP-Hadoop compared to Hadoop.

The results also show that there are jobs for which the IR phase has no benefits. This occurs particularly

[11]https://code.google.com/p/skewtune/

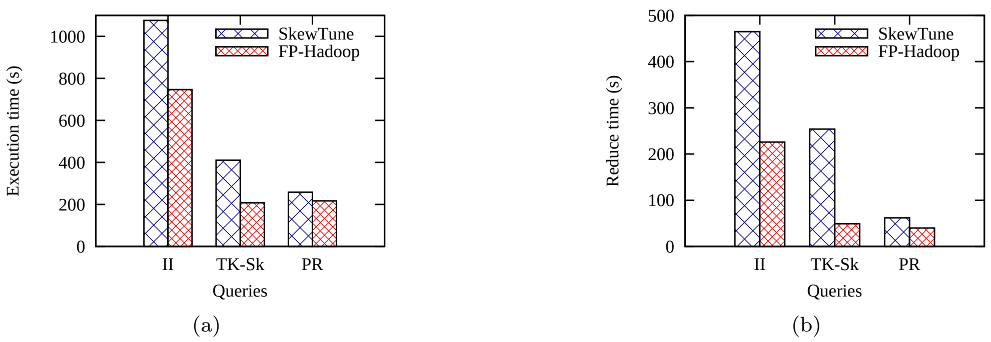


Figure 10: Comparison of FP-Hadoop and SkewTune (a) reduce time, (b) execution time.

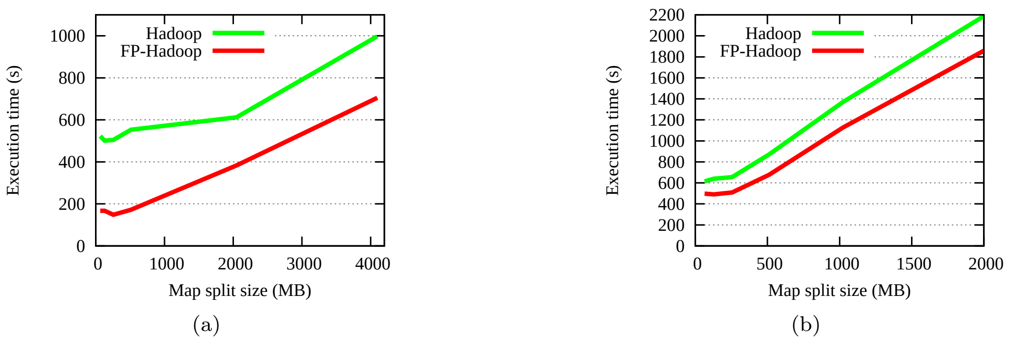


Figure 11: Effect of map split size on (a) Top-k% (TK-SK) and (b) Inverted Index.

in the cases where there is no skew in the intermediate data or the skew can be significantly decreased in the map phase by using a combiner function, e.g. in the word count job which we tested. In these cases, it is sufficient to not declare an IR function, then FP-Hadoop executes the job without performing the IR phase.

6. Related Work

In the literature, there have been many efforts to improve MapReduce [9], particularly by supporting high level languages on top of it, e.g., Pig [10], optimizing I/O cost, e.g., by using column-oriented techniques for data storage [11, 12, 13], supporting loops [14], adding index [15], caching intermediate data [16], supporting special operations such as join and Skyline [17, 18, 19, 20, 21, 22, 23, 24], balancing data skew [25, 26], etc. Hereafter, we briefly present some of them that are the most related to our work.

The approach proposed in [26] tries to balance data skew in reduce tasks by subdividing keys with large value sets. It requires some user interaction or user knowledge of statistics or sampling, in order to estimate in advance the values size of each key, and then subdivide the keys with large values. Gufler et al. [25] propose an adaptive approach which collects statistics about intermediate key frequencies and assigns them to the reduce tasks dynamically at scheduling time. In a similar approach, Sailfish [27] collects some information about the intermediate keys, and uses them for optimizing the number of reduce tasks and partitioning the keys to reducer workers. However, these approaches are not efficient when all or a big part of the intermediate values belong to only one key or a few number of keys (i.e., less than the number of reduce workers).

SkewTune [3] adopts an on-the-fly approach that detects straggling reduce tasks and dynamically repartitions their input keys among the reduce workers that have completed their work. This approach can be efficient in the cases where the slow progress of a reduce task is due to inappropriate initial partitioning of the key-values to reduce tasks. But, it does not allow the collaboration of reduce workers on the same key.

Haloop [14] extends MapReduce to serve applications that need iterative programs. Although iterative programs in MapReduce can be done by executing a sequence of MapReduce jobs, they may suffer from big data transfer between reduce and map workers of successive iterations. Haloop offers a programming interface to express iterative programs and implements a task scheduling that enables data reuse across iterations. However, it does not allow hierarchical execution plans for reducing the intermediate values of one key, as in our intermediate reduce phase. Memory Map Reduce (M3R) [28] is an implementation of

Hadoop that keeps the results of map tasks in memory and transfers them to the reduce tasks via message passing, i.e., without passing via the local disks. M3R is very efficient, but can be used only for the applications in which intermediate key-values can fit in memory. SpongeFiles [29] is a system that uses the available memory of nodes in the cluster to construct a distributed-memory, for minimizing the disk spilling in MapReduce jobs, and thereby improving performance. The idea of using main memory for data storage has been also exploited in Spark [2], an alternative to MapReduce, that uses the concept of Resilient Distributed Datasets (RDDs) to transparently store data in memory and persist it to disc only when needed. The concept of intermediate reduce phase proposed in FP-Hadoop can be used as a complementary mechanism in the systems such as Haloop, M3R, SpongeFiles and Spark, to resolve the problem of data skew when reducing the intermediate data.

In MapReduce Online [30], instead of waiting for reduce tasks to pull the map outputs, the map tasks push their results periodically to the reduce tasks. This allows to increase the overlap between the map and shuffle phases, and consequently to reduce the total execution time. Similarly, we also benefit from an increased overlap as we do not need to wait for the end of a map task in order to start transferring its output. But, MapReduce Online does not resolve the problem data skew in overloaded keys.

There have been systems proposing new phases to MapReduce in order to deal with special problems. For example in Map-Reduce-Merge [19], in addition to the map and reduce phases, a third phase called merge is added to MapReduce in order to merge the reduce outputs of two different MapReduce jobs. The merge phase is particularly used for implementing multi-join operations. However, Map-Reduce-Merge and other solutions proposed for join query processing, e.g. [17, 22], cannot be used for resolving the problem of data skew due to overloaded keys.

In general, none of the existing solutions in the literature can deal with data skew in the cases when most of the intermediate values correspond to a single key, or when the number of keys is less than the number of reduce workers. But, FP-Hadoop addresses this problem by enabling the reducers to work in the IR phase on dynamically generated blocks of intermediate values, which can belong to a single key.

7. Conclusion

In this paper, we presented FP-Hadoop, a system that brings more parallelism to the MapReduce job processing by allowing the reduce workers to collaborate on processing the intermediate values of a key. We added a new phase to the job processing, called intermediate reduce phase, in which the input of reduce workers is considered as a set of IR Splits (blocks). The reduce workers collaborate on processing IR splits until finishing them, thus no reduce worker becomes idle in this phase. In the final reduce phase, we just group the results of the intermediate reduce phase.

By enabling the collaboration of reduce workers on the values of each key, FP-Hadoop improves significantly the performance of jobs, in particular in the case of skew in the values assigned to the intermediate keys, and this is done without requiring any statistical information about the distribution of values.

We evaluated the performance of FP-Hadoop through experiments over synthetic and real datasets. The results show excellent gains compared to Hadoop. For example, over a cluster of 20 nodes with 120GB of input data, FP-Hadoop can outperform Hadoop by a factor of about 10 in reduce time, and a factor of 5 in total execution time. The results show that the higher is the number of nodes, the higher can be the gain of FP-Hadoop. They also show that the bigger is the size of the input data, the higher can be the improvement gain of FP-Hadoop.

Aknowledgements

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: 6th Symposium on Operating System Design and Implementation (OSDI), 2004.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2012.

[3] Y. Kwon, M. Balazinska, B. Howe, J. A. Rolia, Skewtune: mitigating skew in mapreduce applications, in: SIGMOD, 2012.

[4] R. Akbarinia, M. Liroz-Gistau, D. Agrawal, P. Valduriez, An efficient solution for processing skewed mapreduce jobs, in: Database and Expert Systems Applications (DEXA), 2015.

[5] M. Liroz-Gistau, R. Akbarinia, P. Valduriez, Fp-hadoop: Efficient execution of parallel jobs over skewed data, PVLDB 8 (12) (2015) 1856–1859.

[6] T. White, Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3rd ed.), O'Reilly, 2012.

[7] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh, Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals, Data Min. Knowl. Discov. 1 (1) (1997) 29–53.

[8] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, Computer Networks 30 (1-7) (1998) 107–117.

[9] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, B. Moon, Parallel data processing with mapreduce: a survey, SIGMOD Record 40 (4) (2011) 11–20.

[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: SIGMOD, 2008.

[11] A. Floratou, J. M. Patel, E. J. Shekita, S. Tata, Column oriented storage techniques for mapreduce, PVLDB 4 (7) (2011) 419–429.

[12] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, S. Wu, Llama: leveraging columnar storage for scalable join processing in the mapreduce framework, in: SIGMOD, 2011.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2) (2008) 4–26.

[14] Y. Bu, B. Howe, M. Balazinska, M. D. Ernst, The haloop approach to large-scale iterative data analysis, VLDB J. 21 (2).

[15] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, J. Schad, Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing), PVLDB 3 (1) (2010) 518–529.

[16] I. Elghandour, A. Aboulnaga, Restore: reusing results of mapreduce jobs in pig, in: SIGMOD, 2012.

[17] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, J. D. Ullman, Fuzzy joins using mapreduce, in: ICDE, 2012.

[18] J. Huang, R. Zhang, R. Buyya, J. Chen, MELODY-JOIN: efficient earth mover's distance similarity joins using mapreduce, in: ICDE, 2014.

[19] H. chih Yang, A. Dasdan, R.-L. Hsiao, D. S. Parker, Map-reduce-merge: simplified relational data processing on large clusters, in: SIGMOD, 2007.

[20] D. Jiang, A. K. H. Tung, G. Chen, Map-join-reduce: Toward scalable and efficient data analysis on large clusters, IEEE Trans. Knowl. Data Eng. 23 (9) (2011) 1299–1311.

[21] Y. N. Silva, J. M. Reed, Exploiting mapreduce-based similarity joins, in: SIGMOD, 2012.

[22] A. Okcan, M. Riedewald, Processing theta-joins using mapreduce, in: SIGMOD, 2011.

[23] D. Deng, G. Li, S. Hao, J. Wang, J. Feng, Massjoin: A mapreduce-based method for scalable string similarity joins, in: ICDE, 2014.

[24] S. Fries, B. Boden, G. Stepien, T. Seidl, Phidj: Parallel similarity self-join for high-dimensional vector data with mapreduce, in: ICDE, 2014.

[25] B. Gufler, N. Augsten, A. Reiser, A. Kemper, Load balancing in MapReduce based on scalable cardinality estimates, in: ICDE, IEEE, 2012.

[26] S. R. Ramakrishnan, G. Swart, A. Urmanov, Balancing reducer skew in mapreduce workloads using progressive sampling, in: ACM Symposium on Cloud Computing (SoCC), 2012.

[27] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, D. Reeves, Sailfish: a framework for large scale data processing, in: ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012, 2012.

[28] A. Shinnar, D. Cunningham, B. Herta, V. A. Saraswat, M3r: Increased performance for in-memory hadoop jobs, PVLDB 5 (12) (2012) 1736–1747.

[29] K. Elmeleegy, C. Olston, B. Reed, Spongefiles: Mitigating data skew in mapreduce using distributed memory, in: SIGMOD, 2014.

[30] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, R. Sears, Mapreduce online, in: NSDI, 2010.