FP-Hadoop: Efficient Execution of Parallel Jobs Over Skewed Data

Miguel Liroz-Gistau, Reza Akbarinia, Patrick Valduriez

FP-Hadoop: Efficient Execution of Parallel Jobs Over Skewed Data

Miguel Liroz-Gistau
Inria & LIRMM
Montpellier, France
miguel.liroz_gistau@inria.fr

Reza Akbarinia
Inria & LIRMM
Montpellier, France
reza.akbarinia@inria.fr

Patrick Valduriez
Inria & LIRMM
Montpellier, France
patrick.valduriez@inria.fr

ABSTRACT

Big data parallel frameworks, such as MapReduce or Spark have been praised for their high scalability and performance, but 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 demonstration, we illustrate the use of *FP-Hadoop*, a system that efficiently deals with data skew in MapReduce jobs. In FP-Hadoop, there is a new phase, called *intermediate reduce (IR)*, in which blocks of intermediate values, constructed dynamically, are processed by intermediate reduce workers in parallel, by using a scheduling strategy. Within the IR phase, even if all intermediate values belong to only one key, the main part of the reducing work can be done in parallel using the computing resources of all available workers. We implemented a prototype of FP-Hadoop, and conducted extensive experiments over synthetic and real datasets. We achieve excellent performance gains compared to native Hadoop, e.g. *more than 10 times in reduce time and 5 times in total execution time.*

During our demonstration, we give the users the possibility to execute and compare job executions in FP-Hadoop and Hadoop. They can retrieve general information about the job and the tasks and a summary of the phases. They can also visually compare different configurations to explore the difference between the approaches.

1. INTRODUCTION

Big data parallel frameworks, such as MapReduce [1] or its main memory efficient versions such as Spark [3] have been praised for their scalability and performance, using clusters of commodity machines. The idea behind MapReduce is simple and elegant. Given an input file, and two functions named 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.

However, MapReduce and Spark 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 us illustrate this by an example. Suppose we want to return for each language, the top-k% accessed pages of Wikipedia. In MapReduce, we would write a simple program that emits for each log file the number of visits per page and groups them by language and then by page. Then, for each language, we would sort the pages by number of visits and return the first k%. In this example, there may be a high skew in the load of reduce workers. 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[1], 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 reduce 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, yet the job would have to wait for the responsible of English pages to finish.

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 repartitioning the keys to the reduce workers [2]. 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*. But, the input of the combiner function is restricted to the data of one map task, thus its filtering power is very limited for many applications.

In this demonstration, we illustrate the usage of *Fully-Parallel Hadoop* (FP-Hadoop), a system that uses a new approach for dealing with the data skew in reduce side of MapReduce. 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

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

Proceedings of the VLDB Endowment, Vol. 8, No. 11

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, the reducing work is done by using the computing power of available workers. Briefly, the data reducing in the *IR phase* has the following distinguishing features: 1) *collaborative reducing of each key:* the intermediate values of each key can be processed in parallel by using multiple intermediate reduce workers; 2) *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*; 3) *hierarchical execution:* the processing of intermediate values in the IR phase can be done in several iterations. This permits to perform hierarchical execution plans for jobs, in order to decrease the size of the intermediate data more and more.

We implemented a prototype of FP-Hadoop by modifying native 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.*

In our demonstration, the user can execute jobs with FP-Hadoop and Hadoop, and compare visually the job executions in the two systems. For each executed job, we show detailed information about the three phases of FP-Hadoop, the tasks which have been executed at each time and in each phase, the number of working slots at each time, the size of the data which has been reduced in the intermediate phase, etc. Briefly, by using the demonstration the user can discover by himself the reason for which FP-Hadoop is so faster than Hadoop in processing skewed data.

2. FP-HADOOP

2.1 Job Execution 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.

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

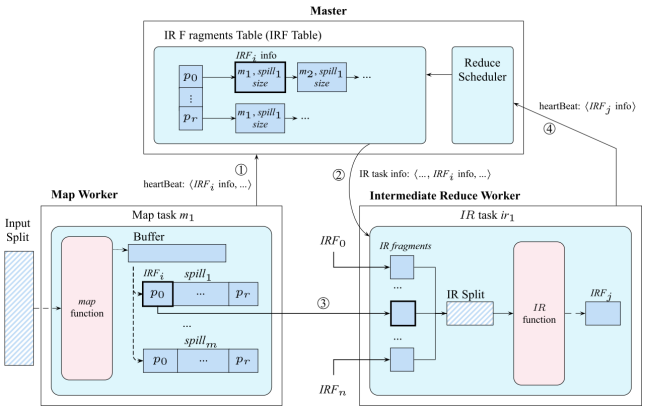


Figure 1: Architecture of FP-Hadoop.

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 2.2, 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. 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.

2.2 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. There are many functions for which we can use the original reduce 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`, `SUM`, `MIN` and `MAX`. For some reduce functions, it may be difficult to find an efficient IR 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.

2.3 Dynamic Construction of 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 the map phase, 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. In Hadoop, 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* (IRF), and the IRFs are used for making IR splits. When a spill is produced by a map task, the information about the spill's IRFs, which we call *IRF metadata*, is sent to the master by using MapReduce's heartbeat message passing mechanism.

For keeping IRF metadata, the master of FP-Hadoop uses a new 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 worker where the IRF has been produced. The master uses the information in IRF Table for constructing IR splits and assigning them to ready reduce workers (see Figure 1). This is done mainly based on the scheduling strategies described in the next subsection.

2.4 Scheduling of Intermediate Tasks

For scheduling an intermediate reduce task, the most important issue is to choose the IRFs that belong to the IR split that should be processed by the task. For this, the following strategies are actually implemented in FP-Hadoop:

- ***Greedy***. In this strategy, the objective is to give the priority to the IR fragments of the partition that has the maximum number of values. This strategy is the default strategy in FP-Hadoop.

- ***Locality-aware***. In this strategy, we try to choose for a worker w in the first place the IRFs that are on its local disk or close to it.

The user can configure FP-Haddop to execute the IR phase in several iterations, in such a way that the output of each iteration is consumed by the next iteration. There is a parameter that defines the maximum number of iterations, although in practice each partition may be processed in a different number of iterations, for instance depending on its size, input/output ratio or skew. FP-Hadoop implements by default an approach where an iteration is launched only if its input size is more than a given threshold.

2.5 Performance gains

We have performed a thorough evaluation of FP-Hadoop and compared it with native Hadoop. We obtain increasing performance gains as the input size grows. For instance, in the top-k% example described in Section 1, for an input of 120GB and a 20 nodes cluster, we decrease by 5 the total execution time and by 10 the reduce time. In general, performance gains increase when augmenting the cluster size, the data size and the skew. All these gains, and those obtained for other queries and configurations will be available for examination in our demonstration.

3. DEMONSTRATION

The goal of the demonstration is to show the potential of FP-Hadoop as a tool to accelerate the execution of parallel jobs and to illustrate the details of job execution compared to Hadoop. To such end, we have developed a Web front-end that allows the user to execute jobs and analyze their execution, both in native Hadoop and FP-Hadoop.

The demonstration consists in two phases. First, the user is provided with two possibilities: a) she can define the parameters of a job and submit it to the cluster for execution or b) select an already executed job based on a set of criteria (e.g., query, data parameters, etc.). Second, the user can analyze and compare the traces of the executions of the jobs defined before. She is offered several views, showing different aspects and statistics of the executions.

3.1 Deployment

We will deploy our prototype of FP-Hadoop in one of the clusters of Grid5000[2]. Jobs will be executed in a cluster between 20 and 40 nodes over real and synthetic datasets, and the real job statistics measured during execution. The Web platform designed for the demonstration will let the user specify the characteristics of such jobs and submit them directly to the cluster and then collect and analyze visually their execution.

3.2 Job Specification

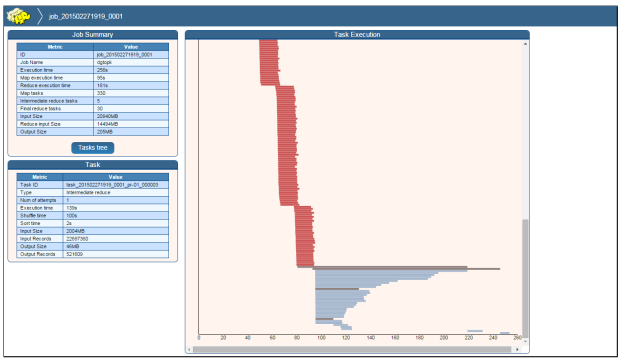
Two scenarios are possible: either the user specifies and executes a new job or she chooses one previously executed:

Scenario 1: The user is allowed to define and submit jobs to the cluster. A set of small datasets are deployed at the beginning of the demonstration, allowing the user to try FP-Hadoop with different queries and explore different parameters. Datasets and queries are set so that execution of jobs is confined to a 5 minutes time-frame, thus allowing a certain amount of interactivity. Jobs can be executed in the background, so that this time can be used for the description of the prototype and the demonstration of the analysis platform with the traces of jobs already executed.

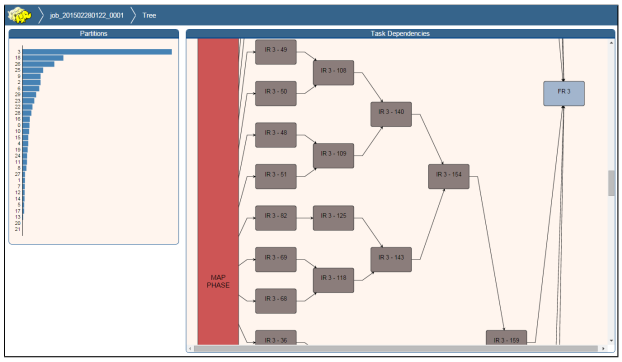
The user can play with some parameters of the queries, e.g., the percentage of selected data and its skew in Top-K, and also specific parameters of FP-Hadoop, such as the number of iterations or the size of IR fragments. The user has also the possibility to execute his/her own jobs providing the necessary traces for our analysis tool.

Scenario 2: Differences in performance between native Hadoop and FP-Hadoop manifest more clearly when jobs are executed over big input datasets, whose execution times fall outside the time constraints of the demonstration. That is why, several jobs have been executed before the demonstration and their traces made available to the user for exploration and analysis. These jobs have been configured with

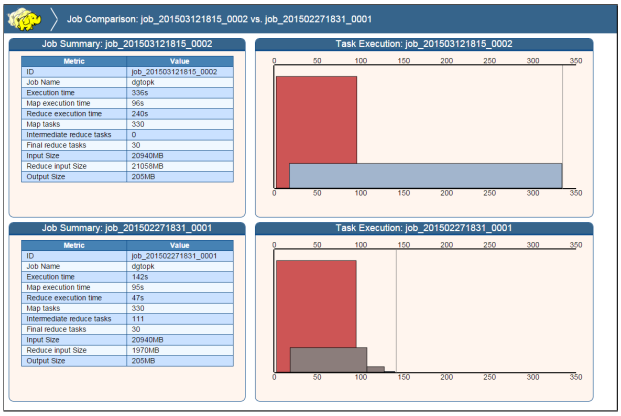
[2]https://www.grid5000.fr



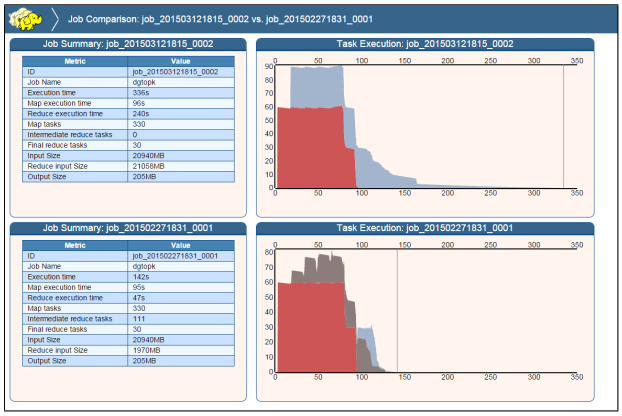
(a) Job details



(b) Task dependencies in multiple iterations of IR phase



(c) Job comparison: different phases and their input size



(d) Job comparison: number of working slots at each time in different phases

Figure 2: Example of the views provided by our demonstration tool

several queries, e.g., Top-k, Inverted Index, Wordcount; and a wide range of parameters, including diverse input sizes, number of participating nodes and cluster parameters.

3.3 Job Analysis

In the two scenarios described before the users can explore and analyze the traces of job executions and discover how FP-Hadoop behaves and compares to native Hadoop. The exploration can be done using the following views:

- **Job Details:** The user can explore the details of the execution of a job, including the general and individual task statistics and a Gantt chart of task execution where the different type of tasks (map, intermediate and final reduce) can be recognized, and the effect of data skew identified. An example of this view is shown in Figure 2(a). Map (red), intermediate (brown) and final reduce (blue) tasks can be identified by different colors and selected to get their individual statistics.

- **Task Dependencies:** The iterative nature of FP-Hadoop can result in a multi-iteration execution of the intermediate reduce phase, where data is aggregated and reduced incrementally. Our Web front-end allows the user to explore how IR tasks are organized in the multiple iterations in the IR phase, and how the number of iterations and tasks adapts to the size of the intermediate values. An example of such a view is shown in Figure 2(b).

- **Job Comparison:** Finally, the user is provided the option of comparing two jobs in a visual manner with the interfaces shown in figures 2(c) and 2(d). In the first view, tasks are aggregated into phases, and both the input data (y-axis) and execution time (x-axis) are represented. In the second, the number of occupied slots at each time is depicted, so that we can compare how FP-Hadoop exploits the parallelism provided by the cluster. This representation provides a quick visual explanation of the differences in the behavior between FP-Hadoop and native Hadoop executions.

4. REFERENCES

[1] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[2] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD Conference*, pages 25–36, 2012.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI Conference*, pages 15–28, 2012.