An Efficient Solution for Processing Skewed MapReduce Jobs

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

An Efficient Solution for Processing Skewed MapReduce Jobs

Reza Akbarinia[1], Miguel Liroz-Gistau[1], Divyakant Agrawal[2], and Patrick Valduriez[1]

[1] INRIA & LIRMM, Montpellier, France
{Reza.Akbarinia, Miguel.Liroz_Gistau,
Patrick.Valduriez}@inria.fr
[2] University of California, Santa Barbara, USA
agrawal@cs.ucsb.edu

Abstract. Although MapReduce has been praised for its high scalability and fault tolerance, it has been criticized in some points, in particular, its poor performance in the case of data skew. There are important cases where a high percentage of processing in the reduce side is done by a few nodes, or even one node, while the others remain idle. There have been some attempts to address the problem of data skew, but only for specific cases. In particular, there is no proposed solution for the cases where most of the intermediate values correspond to a single key, or when the number of keys is less than the number of reduce workers.
In this paper, we propose *FP-Hadoop*, a system that makes the reduce side of MapReduce more parallel, and efficiently deals with the problem of data skew in the reduce side. 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. By using 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 by 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 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, Load Balancing

1 Introduction

MapReduce [3] is one of the most popular solutions for big data processing, in particular due to its automatic management of parallel execution in clusters of machines. Initially proposed by Google, its popularity has continued to grow over time, as shown by the tremendous success of Hadoop [1], an open-source implementation.

The idea behind MapReduce is simple and elegant. Given an input file, and two map and reduce functions, 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 has been praised for its high scalability and fault tolerance, it has also been criticized in some points, particularly its 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.

Example 1. Top accessed pages in Wikipedia. Suppose we want to analyze the statistics[3] 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 about the pages visited in that hour. More precisely, for each visited page there is a line containing some information about the page including its url, language and the number of visits. Given a file, we want to return the top-k% of pages accessed for each language (e.g. top 1%). To answer this query, we can write a simple program as in the following Algorithm[4]:

Algorithm 1: Map and reduce functions for Example 1

```
map( id : K1, content : V1 )
    foreach line ⟨lang, page_id, num_visits, ...⟩ in content do
        emit (lang, page_info = ⟨num_visits, page_id⟩)

reduce( lang : K2, pages_info : list(V2) )
    Sort pages_info by num_visits
    foreach page_info in top k% do
        emit (lang, page_id)
```

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[5], 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 repartitioning the keys to the reduce workers [8]. However,

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

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

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

this approach is not efficient in many cases, e.g. when there is only one 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 [12], thus its filtering power is very limited for many applications.

In this paper, we propose *FP-Hadoop*, a system that uses a new 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 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 by modifying Hadoop's code. We 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*.

The rest of this paper is organized as follows. In Section 2, we propose FP-Hadoop. In Section 3, we report the results of our experiments done to evaluate the performance of FP-Hadoop. In Section 4, we discuss related work, and Section 5 concludes.

2 FP-Hadoop

In this section, we propose FP-Hadoop, a new Hadoop-based system designed for dealing with data skew in MapReduce jobs. We first introduce the programming model of FP-Hadoop, its main phases, and the functions that are necessary for executing the jobs. Then, Then, we provide a more detailed description of the FP-Hadoop design, such as our technique for constructing the working blocks of the IR phase, and the scheduling of intermediate workers.

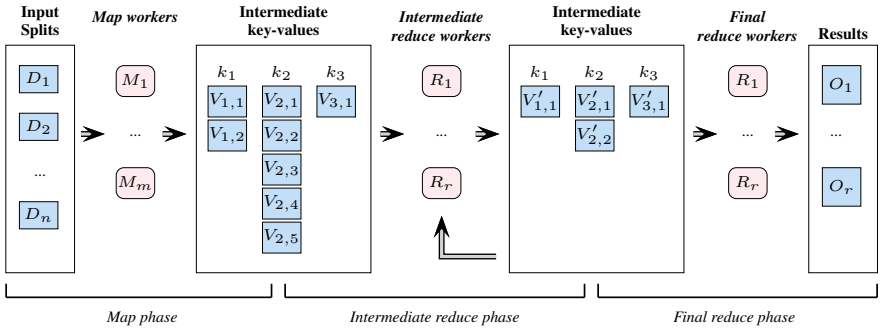


Fig. 1. FP-Hadoop job processing scheme

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 (see Figure 1). 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*, that can be configured 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.

In FP-Hadoop, the execution of a job is done in three 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 (more details about the management of IR splits are given in Section 2.3).

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 different nodes. More specifically, in this phase, each 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.

The intermediate reduce phase can be repeated in *several iterations*, to apply the IR function several times on the intermediate data, and incrementally decrease the size of the final splits which will be 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 threshold (which can be configured by the user).

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. 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.

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 reduce (IR) and final reduce (FR) functions are as follows:

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

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 generates two functions IR and FR, such that for any intermediate key k and its list of values V, $R(k, V) = FR(k, \langle IR(k, V_1), ..., IR(k, V_n)\rangle)$ for every partition $V_1 \cup ... \cup V_n = V$.

The following example illustrates the IR and FR functions for a job that implements the average operation.

Example 2. **Avg.** Consider a job that computes the average of the numeric values that are in a big file. To implement this job in Hadoop, it is sufficient to `emit(1, value)` for each read value in the map function. Then, the reduce function computes the sum and count of all values and returns sum/count. In FP-Hadoop, in the IR function, we compute the (partial) sum and count of the values in the input IR split, and `emit(1, {sum, count})`. This allows to compute the partial counts and sums in parallel. Then, in the FR function, we compute the sum of partial sums and counts, and divide the total sum by the total count.

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`.

Please notice that if the programmer does not want to use the IR phase, he/she should specify no IR function. In this case, the final reduce phase starts just after the map phase completes, i.e., as in Hadoop.

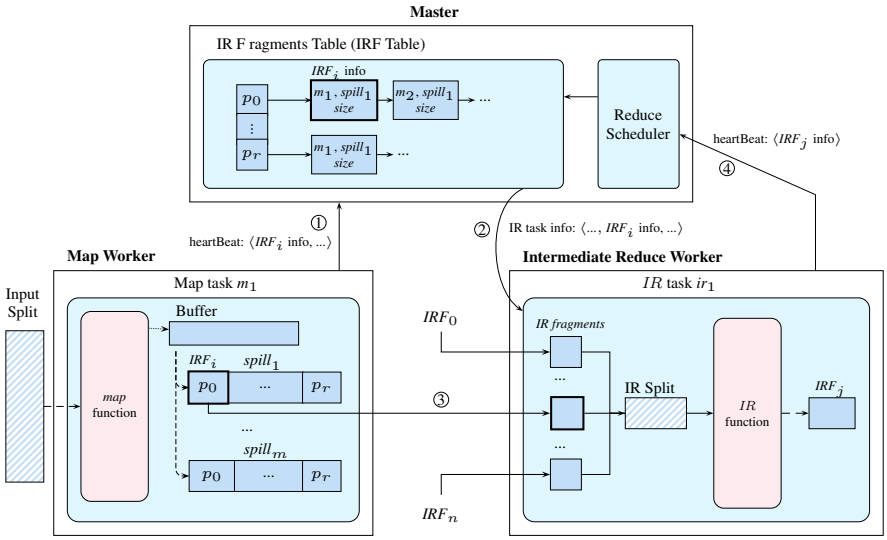


Fig. 2. Data flow between a map worker, the master and an intermediate reduce worker. The communicated messages are shown in their sent order.

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 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 of FP-Hadoop, 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[6]. The data flow between FP-Haddop components is shown in Figure 2.

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

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

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.

2.4 Scheduling of Intermediate Tasks

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. For this, the following strategies are actually implemented in FP-Hadoop:

- ***Greedy***. In this strategy, the objective is to give priority to the IR fragments of the partition that has the maximum number of values. In our implementation of IRF Table, for each partition, we keep the total size of IR fragments. Thus, by a scan of IRF Table, we can find the partition that has the highest number of values until now. After finding the partition, we scan its list, and take from the head of the list the IR fragments 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, we try to choose for a worker w the IR fragments that are on its local disk or close to it. For this, the scheduler scans IRF Table, and finds the partitions whose total data size is at least *minIRsize* (the minimum defined size for IR split), and chooses among them the partition that has the maximum local data at w. After choosing the partition, say p, the scheduler chooses a combination of p's IR fragments at w with size between *minIRsize* and *maxIRsize*. If the total size of p's IR fragments at w is lower than *minIRsize*, then the scheduler completes the IR split by first choosing IR fragments from the same rack as that of w, and then if necessary from the same data center.

In FP-Haddop, the programmer can configure the system 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 *maxIter* that defines the maximum number of iterations. Notice that this parameter sets 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. By default, FP-Hadoop implements an approach in which an iteration is launched only if its input size is more than a given threshold, *minIterSize*. The default value for the threshold is the same as *minIRsize* (i.e., the minimum size of IR splits).

3 Performance Evaluation

We implemented a prototype of FP-Hadoop by modifying Hadoop's code. In this section, we report on the results of our experiments for evaluating the performance of FP-Hadoop. 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, particularly by varying parameters such as the number of nodes in the cluster, the size of input data, etc.

3.1 Setup

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

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. Our query consists of 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 Wikipedia page view statistics [7] stored in hourly log files. We also produced a synthetic dataset (TK-SK), where the number of articles per language follows a Zipfian distribution function with exponent $S = 1$ and N=10 (i.e. 10 languages). We have performed several tests varying the data size, among other parameters, up to 120GB. The query is implemented using a secondary sort [12], 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 [8]. We used a RADIX partitioner to map letters of the alphabet to reduce tasks and produce a lexicographically ordered output. We have executed the job with a dataset containing 20GB of Wikipedia articles.

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

Wordcount (WC). Finally, we have used the wordcount job provided in Apache Hadoop. We have applied it to a dataset generated with the `RandomWriter` job provided in the Hadoop distribution. We tested this job with a 100GB dataset.

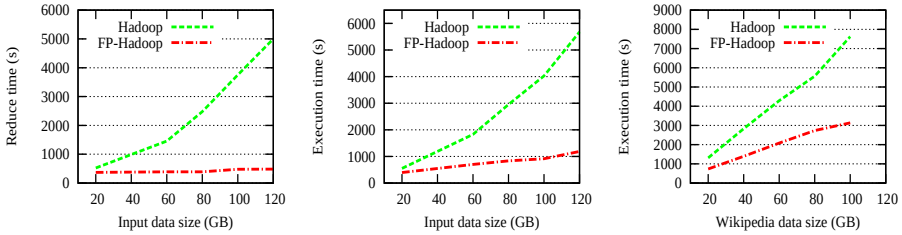
The default values for the parameters which we used in our experiments are as follows. The default number of nodes which we used in our cluster was 20. Unless otherwise specified, the input data size in the experiments was 20 GB. In FP-Hadoop, the default value for *minIRsize* was set to 512 MB. The value of *maxIRsize* is always twice as that of *minIRsize*, and the maximum number of iterations is set to 1.

We compared FP-Hadoop with Hadoop and SkewTune [8] which is the closest related work to ours (see a brief description in Related Work Section).

In all our experiments, we used 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 used to decrease the amount of data transferred from map to reduce workers, and so to decrease the load of reduce workers.

In our experiments, we have measured the *execution time*, which computes the time elapsed between the start and end of a job, and the *reduce time*, which only considers the time elapsed from the end of the last map task.

[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/



(a) Reduce time vs. input data size (b) Execution time over TK-SK data (c) Execution time over TK-RD data

Fig. 3. Scalability of FP-Hadoop

We run the experiments over a cluster of nodes (up to 50 nodes). The nodes are provided with Intel Quad-Core Xeon L5335 processors with 4 cores each, and 16GB of RAM. All the experiments were executed with a number of reduce workers equal to the number of machines. We have changed *io.sort.factor* to 100, as advised in [12], which actually favors Hadoop. For the rest of the parameters, we have used Hadoop's default values.

3.2 Scalability

We investigated the effect of the input size on the performance of FP-Hadoop compared to Hadoop. Using TK-SK dataset, Figures 3(a) and 3(b) 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 3.1. Figure 3(c) shows the performance using TK-RD dataset with sizes up to 100 GB, while other parameters as default values described in Section 3.1. As expected, increasing the input size increases the execution time of both Hadoop and FP-Hadoop, because more data will be processed by map and reduce workers. But, the performance of FP-Hadoop is much better than Hadoop when we increase the size of input data. For example, in Figure 3(b), the gain 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.

3.3 Effect of Cluster Size

We studied the effect of the number of nodes of the cluster on performance. Figure 4(a) shows the execution time by varying the number of nodes, and other parameters set as default values described in Section 3.1. 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 4(a), with 5 nodes, FP-Hadoop outperforms Hadoop by a factor of around 1.75. But, 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. But, 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.

3.4 Overhead of IR phase for Balanced Jobs

Our performance evaluation results, reported until now, show that the IR phase in FP-Hadoop significantly improves the performance of skewed jobs. Let us now investigate its overhead in the case where there is no skew in the job. For this, we have chosen the word-count (WC) query over a uniform random dataset as described in Section 3.1. In this job, the key partitioner balances perfectly the reduce load among workers, thus there is no skew in the reduce side. Figure 4(b) shows the total execution time of FP-Hadoop and Hadoop for this job. The results show that the execution time of FP-Hadoop is a little (3%) higher than Hadoop, and this increase corresponds to the overhead of the IR phase. Indeed, FP-Hadoop spends some time to detect the lack of skew in the intermediate results, and then launches the final reduce phase. Thus, its execution time is slightly higher than Hadoop. We believe that this very slight overhead of IR phase in perfectly balanced jobs can usually be tolerated. If not, the programmer simply disables the IR phase, then FP-Haddop does not launch that phase.

3.5 Comparison with SkewTune Using Different Queries

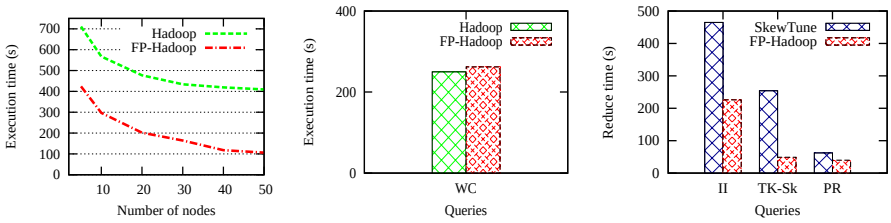
We compared FP-Hadoop with SkewTune [8] using different queries. Figure 4(c) shows the reduce time and execution time of both approaches, using the data and parameters described in Section 3.1. For these experiments, we downloaded the SkewTune prototype [11]. The data which we used are the default data and sizes described in Section 3.1 (e.g. 20GB of data for TK-SK). As the results show, FP-Hadoop can outperform SkewTune with significant factors. The main reason is that SkewTune is unable to split the computation of the tuples assigned to the same intermediate key.

4 Related Work

In the literature, there have been many efforts to improve MapReduce [9]; these include supporting loops [2], adding index [4], caching intermediate data [5], balancing data skew [7,10,8]. Hereafter, we briefly present some of them that are the most related to our work.

The approach proposed in [10] 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. [7] propose an adaptive approach

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



(a) Effect of cluster size (b) Overhead of IR phase (c) Reduce time of SkewTune vs. FP-Hadoop

Fig. 4. Effect of different parameters, and comparison with SkewTune

that collects statistics about intermediate key frequencies and assigns them to the reduce tasks dynamically at scheduling time. In a similar approach, Sailfish [11] 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., fewer than the number of reduce workers).

SkewTune [8] 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 [2] 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. SpongeFiles [6] 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. Spark [13], an alternative to MapReduce, uses the concept of Resilient Distributed Datasets (RDDs) to transparently store data in memory and persist it to disk 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, SpongeFiles and Spark, to resolve the problem of data skew when reducing the intermediate data.

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.

5 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 pool 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. 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 the number of nodes, the greater the potential gain from FP-Hadoop. They also show that the bigger the size of the input data, the larger the potential improvement from FP-Hadoop.

6 Acknowledgments

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. Hadoop. http://hadoop.apache.org (2014)
2. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: The HaLoop approach to large-scale iterative data analysis. VLDB Journal 21(2) (2012)
3. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI (2004)
4. Dittrich, J., Quiané-Ruiz, J.A., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). PVLDB 3(1) (2010)
5. Elghandour, I., Aboulnaga, A.: ReStore: reusing results of MapReduce jobs in Pig. In: SIGMOD (2012)
6. Elmeleegy, K., Olston, C., Reed, B.: SpongeFiles: Mitigating data skew in mapreduce using distributed memory. In: SIGMOD (2014)
7. Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Load balancing in MapReduce based on scalable cardinality estimates. In: ICDE. IEEE (Apr 2012)
8. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.A.: SkewTune: mitigating skew in MapReduce applications. In: SIGMOD (2012)
9. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. SIGMOD Record 40(4) (2011)
10. Ramakrishnan, S.R., Swart, G., Urmanov, A.: Balancing reducer skew in MapReduce workloads using progressive sampling. In: ACM Symposium on Cloud Computing, SoCC (2012)

11. Rao, S., Ramakrishnan, R., Silberstein, A., Ovsiannikov, M., Reeves, D.: Sailfish: a framework for large scale data processing. In: ACM Symposium on Cloud Computing, SoCC (2012)
12. White, T.: Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3rd ed.). O'Reilly (2012)
13. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI (2012)