A Comparative Analysis of Join Algorithms Using the Hadoop Map/Reduce Framework

Konstantina Palla

Master of Science
School of Informatics
University of Edinburgh

2009

Abstract

The Map/Reduce framework is a programming model recently introduced by Google Inc. to support distributed computing on very large datasets across a large number of machines. It provides a simple but yet powerful way to implement distributed applications without having deeper knowledge of parallel programming. Each participating node executes Map and/or Reduce tasks which involve reading and writing large datasets. In this work, we exploited the open source Hadoop implementation of the Map/Reduce framework and developed a theoretical cost model to evaluate the I/O cost induced on each node during the execution of a task. Furthermore, we used our model to evaluate and compare two basic join algorithms already provided by the framework — the *reduce side* and the *map side* join — and a third one we implemented and which makes use of a Bloom filter on the map tasks. The experimental results proved the validity of our cost model and, furthermore, stressed out that our proposed algorithm claims the least I/O cost in comparison to the others.

This work is expected to provide a good insight into the Map/Reduce framework in terms of I/O cost on the one hand, and a thorough analysis of three join implementations under the Map/Reduce dataflow on the other hand.

Acknowledgements

First, I would like to thank my supervisor, Stratis Viglas, for his guidance and help throughout this project and mostly for the invaluable support in difficult times of the project period. I would also like to thank my family for standing always by me in every choice I make.

Finally, I would like to thank the French wineries and the Tindersticks for giving a bittersweet taste to my work.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Konstantina Palla*)

To my grandmother...

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Map/Reduce is a programming framework introduced by Google to perform parallel computations on very large datasets, for example, crawled documents, or web request logs [7]. The large amounts of data requires that it is distributed over a large number machines (nodes). Each participating node contributes storage sources and all are connected under the same distributed filesystem. Additionally, each machine performs the same computations over the same locally stored data, which results in large-scale distributed and parallel processing. The Map/Reduce framework takes care of all the underlying details regarding parallelization, data distribution and load balancing while the user is concerned only about the local computations executed on every machine. These computations are divided into two categories: the *map* and the *reduce* computations. The nodes assigned with the former take as input their local data and process it producing intermediate results that are stored locally. The reduce nodes receive the map intermediate outputs, combine and process them to produce the final result which in turn is stored in the distributed file system.

Hadoop is the Apache Software Foundation open source and Java-based implementation of the Map/Reduce framework [1]. It provides the tools for processing vast amounts of data using the Map/Reduce framework and, additionally, it implements a distributed file-system similar to Google's file system. It can be used to process vast amounts of data in-parallel on large clusters in a reliable and fault-tolerant fashion. Consequently, it renders the advantages of the Map/Reduce available to users.

The idea of introducing the Map/Reduce framework into the field of databases seems intuitively right. Databases aim to process large volumes of data and consequently deploying the Map/Reduce framework for the implementation of the basic database algorithms seems more than natural. Among these algorithms, the join appears to be the most challenging and expensive in terms of I/O cost. This project exploits the Hadoop Map/Reduce framework and provides an I/O cost analysis of two joins implementations: the *reduce side* and the *map side* join. To compare the two implementations, we extracted a basic cost model associated with each map and reduce task. Based on this model, we computed the total cost of each join algorithm and moreover, we introduced an optimization to the reduce side join using a filter on the map tasks. To test and compare our theoretical expectations with the real values, we set up a cluster consisting of three nodes and conducted a series of experiments. The results underlined the validity of our cost model and moreover, suggested the general domination of the reduce side join with filter.

1.1 Background

To provide a better comprehension of the Hadoop Map/Reduce framework, we will refer to its basic characteristics.

Data distribution A cluster that runs the Hadoop Map/Reduce framework may consist of a large number of commodity machines, the so called nodes. When the data is loaded on the cluster, it is distributed to all the nodes of the cluster. For distributed data storage, the Hadoop framework implements a distributed file system called HDFS (Hadoop Distributed Filesystem). The latter follows a master/slave architecture; a single node, called *NameNode* constitutes the master server that manages the file system namespace and the access to files. The rest of the nodes are the slaves often called *DataNodes* and are assigned with the storage of the data locally on the machine they run on. When a file is loaded on the cluster, it is split into one or more blocks and the *NameNode* maps each block to a *DataNode*. In addition to this, each block is replicated across several machines to prevent data loss in case of a single machine failure.

Data locality The Map/Reduce framework aims to collocate the data with the node that performs computations on it. In detail, a file that is used as input to a Map/Reduce job is divided in a number of blocks distributed across the nodes of the cluster. The *NameNode* using knowledge of the filesystem, tries to assign the processing of each block to the *DataNode* on which the data block is actually stored. This strategy of moving the computation to the data is known as *data locality* optimization and results in high performance preventing unnecessary network transfers.

Key-Value pair orientation The Map/Reduce framework introduces a way of processing data that consists of two main phases: the *map* and the *reduce* phases. Each Map/Reduce job is executed by a number of *DataNodes* that are assigned the role of a mapper and/or a reducer.

Each mapper and reducer implements user defined functions, the map and reduce functions correspondingly. Both functions operate exclusively on $\langle key, value\rangle$ pairs. In detail,

- *The Map phase:* During the map phase, each mapper reads the input record by record, converts it into a key/value pair and propagates the pair to the user defined map function. The latter transforms the input pair (based on the user's implementation) and outputs a new key/value pair. The map function's output pairs are further propagated to subsequent phases where they are partitioned/grouped and sorted by their keys. Grouping is achieved using a partitioner which partitions the key space by applying a hash function over the keys. The total number of partitions is the same as the number of reduce tasks (reducers) for the job. Finally, these intermediate pairs are stored locally on the mappers.

- *The Reduce phase:* During the reduce phase, the *DataNodes* that implement the reducers, retrieve the intermediate data from the mappers. Specifically, each reducer fetches a corresponding partition from each mapper node in a fashion that all values with the same key are passed to the same reducer. The fetched map output pairs are merged constructing pairs of $\langle key, list(values)\rangle$ based on the same key. The newly structured pairs of key/list are propagated to the user defined reduce function. The latter receives an iterator over each list, combines these values together and returns a new key/value pair. The final output pairs are written to output files and stored in the HDFS.

The whole Map/Reduce procedure is depicted in Figure 1.1.

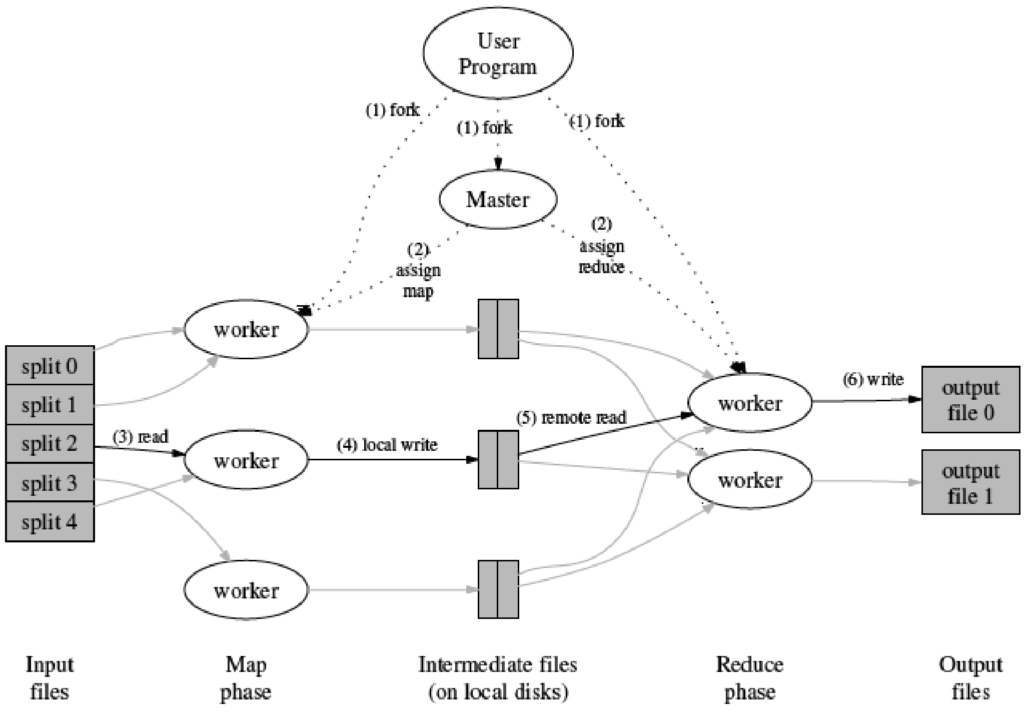

Figure 1.1: Execution Overview

The Map/reduce process consists of a sequence of actions/steps that are described as follows:

1. The input data is loaded on the cluster and is split into *M* blocks which in turn are distributed across the nodes based on the HDFS regulations. The framework then starts up many copies of the program on the participating nodes.

2. Among the participating nodes, there is one that constitutes the master node, that is the *NameNode*. The other nodes are the *DataNodes* also known as workers. The NameNode schedules the task distribution among the workers; there are *M* map and *R* reduce tasks to assign. It picks idle workers and assigns to each one a map or a reduce task following the data locality optimization approach. Each worker assigned with a map task processes a block of the input data.

3. During the map phase, a worker reads the records from the corresponding input split (reading phase of the map task). It computes key/value pairs and passes each pair to the user-defined map function. The latter produces intermediate key/value pairs which are stored in memory (buffering phase). By processing the input in splits, data parallelism is exploited.

4. The intermediate results buffered in memory, are periodically written to local disk in *R* sorted partitions so that the reduce workers can selectively fetch data for the reduce phase

(writing phase). The location of these stored results is forwarded to the master.

5. The master forwards the locations to the workers assigned with the reduce tasks. Each worker fetches the right partition of data from the local disk of each map worker and writes it in memory and/or on disk if necessary (copy/shuffle phase). After all map outputs have been successfully copied the sort phase of the reduce task begins. The name is misleading since the sort phase merges the map outputs while sorting has already taken place on the map side. Merging is executed in a way that maintains the sort order. As a result, all the occurrences of the same key are grouped.

6. The reduce worker processes the grouped sorted data and for each unique intermediate key encountered, it forwards the key and the corresponding list of intermediate values to the user-defined reduce function. The output of the reduce function is concatenated to a final output file and stored in the HDFS.

After successful completion, the output of the Map/Reduce job is available in the R output files, that is, the output consists of one output file per executed reduce task.

We conduct a thorough analysis of each intermediate phase in Chapter 2.

1.2 Related Work

Since Google researchers Dean and Ghemavat proposed the Map/Reduce programming model for data- and compute-intensive environments [7], it has received much attention from across the computing and research community. A lot of work has been done to compare the Map/Reduce model with parallel relational databases. Pavlo et al. [9] conducted experiments to evaluate both parallel DBMS and the Map/Reduce model in terms of performance and development complexity. While the Map/Reduce model outperformed in scalability and fault tolerance, the study also underlined the performance limitations of the model, in terms of computation time, attributed mainly to the fact that the model was not originally designed to perform structured data analysis and lacks many of the basic features that the relational databases routinely provide. On the other hand, advocates of the model, deployed it to investigate its computing capabilities in environments with large amounts of raw data. The work of Chu et al. [6] exploited the Map/Reduce model for parallelizing machine learning algorithms on a single machine with multiple processors. Moreover, Abouzeid et al. in their work [3] attempted to bridge the gap between the two technologies, that is, parallel databases and Map/Reduce model, suggesting a hybrid system that combines the best features from both. The acquired results appeared quite promising and pointed out the advantages of an approach that combines the efficiency of relational databases with the scalability and fault tolerance of the Map/Reduce model. However, all these pieces of work evaluated the model having as a metric the computation time and efficiency and always under a close comparison with database systems. No attempt was carried out to analyse the I/O cost that the Map/Reduce executions incur on the related systems.

Several research studies also aim to improve the model itself. Yang et al. [5] proposed a Merge component after the reduce function for performing a join operation on two datasets. However, the Map-Reduce-Merge approach introduces an extra processing step that is not there in the standard Map/Reduce framework and therefore will not be found in a standard Map/Reduce deployment. Moreover, the Pig project at Yahoo [8], the SCOPE project at Microsoft [4], and the open source Hive project [2] introduce SQL-style declarative languages over the standard Map/Reduce model in an attempt to make it more expressive and limit its schema-less disadvantages. All these projects provide ways of declaring join executions over datasets. However, since they are higher level interfaces, they deploy the mechanisms of the standard Map/Reduce framework, which they rely on, without introducing any system-level solution. All in all, all these approaches present join methods that either are not inherit in the actual model or exploit the basic mechanisms. They do not extend the model in a way that it can always support the join operation. Consequently, the only join implementations known until now under the Map/Reduce framework is the ones we analyse; the *reduce side* and the *map side* join.

1.3 Motivation, Project Aims and Contributions

The Map/Reduce framework has been extensively used for joining huge input files. Having the partitioning over the key space and the grouping based on the same key as main components of the standard map/reduce data flow, the implementation of the join algorithm using the Map/Reduce framework seems intuitively right. Indeed, there are two main join implementations; the *reduce side* and the *map side* join. Their name denotes the phase during which the actual join takes place. The reduce side join is conducted during the reduce phase when the records that have the same key value are presented to the same reducer and the user-defined reduce function is assigned with the task of joining them. On the other hand, the map side join is computed before the input data reaches the map function, and thus it requires that the data is already sorted and partitioned (full details are provided in subsequent chapters).

There is a great controversy over which join implementation the user should choose. Issues such as computation time and I/O cost are often taken into account. This study analyzes and compares the two implementations in terms of the associated I/O cost. We propose a basic cost model related to each map and reduce task and suggest that its application is capable of predicting the I/O cost related to each join implementation. Our theoretical expectations are tested through a series of diverse experiments where the predicted values of the total I/O cost are compared to the real ones. Moreover, we introduce an optimization to the simple reduce side join by applying the use of a Bloom filter during the execution of map tasks. Based on the results of our experiments we evaluate and compare all the three join implementations and investigate under which conditions each implementation claims the least I/O cost. For the purposes of our study, we set up a Hadoop Map/Reduce cluster.

1.4 Thesis Outline

The thesis has been divided into 5 chapters, starting from this introductory one. The remaining chapters are organized as follows:

- ◇ Chapter 2 presents a thorough analysis of the map and reduce tasks and introduces our cost model for each one
- ◇ Chapter 3 gives an overview of the three join implementations, the reduce side join, the map side join and our optimized version of the reduce side join and presents the cost model to estimate the I/O cost of each implementation
- ◇ Chapter 4 focuses on the detailed presentation of the conducted experiments along with the acquired results
- ◇ Chapter 5 summarizes our findings, draws overall conclusions and suggests areas of future extension

Chapter 2

I/O Cost Analysis

2.1 I/O Cost Analysis of the Map Phase

To analyze the I/O cost of the map phase, one should consider every intermediate step carefully. It would be a mistake to consider that the total I/O cost equals to 2× input split's size. When the map function is triggered, the map process does not simply read the input and writes the relevant output to disk. In contrast, the process is more complicated and includes sorting and partitioning stages where writes to buffer and spills to disk take place.

The internal steps of a map task are depicted in Figure 2.1.

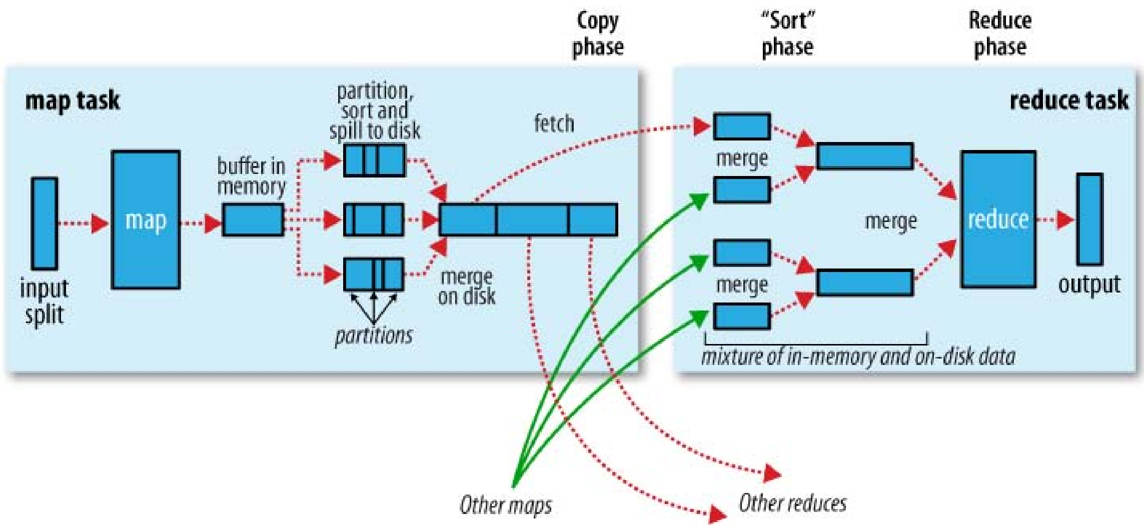

Figure 2.1: MapReduce data flow with a single map and reduce task

The crucial property here is the memory buffer size allocated to the map task. The user-defined map function outputs key-value pairs. The map task writes each key-value pair output to the buffer. When the size of the data contained in the buffer reaches a certain threshold (which is a portion of the available buffer size) the buffered data is partitioned, sorted and spilled to the local disk of the node currently running the map task (it is written to the directory indicated by the `mapred.local.dir` property value). Each time the memory buffer reaches the spill threshold, a new spill file is created. By the end of the map task there could be several spill files. After the final spill file is created, the locally stored spill files are read and merged into a single partitioned and sorted output file. This is the final output file of the current map task.

Based on the above analysis, there could be several reads and writes of the data from and to the disk. This results in greater I/O cost than the one expected if only the initial reading and the final writing were to be considered. To achieve a better understanding of the above mentioned procedure, we divide the map phase into three phases; the reading, buffering and split phase. Some details of the implementation are provided but only to such an extend that is helpful and not confusing.

2.1.1 Reading phase

During reading, the map task reads the input which is called split. Hadoop, following the data locality optimization approach, does its best to run the map task on a node where the input data resides in HDFS. Reading is not associated with writing the split locally. On the contrary, the input, in the form of key-value pairs, is processed based on the user-defined map-function and the output (new key-value pair) is pushed to the in-memory buffer of the map task so as the subsequent buffering phase can take place. The I/O cost of this phase equals to zero since the input is directly fetched from the HDFS. Defining as *splitSize* the input size in bytes, the reading cost is

$$\mathbf{cost}^{r}_{map} = 0$$

2.1.2 Buffering phase

During buffering three procedures take place; partitioning, sorting and spilling (to disk). Buffering is the phase during which the map output is serialized and written to a circular buffer. The available buffer size is defined by the configuration property `io.sort.mb` and by default is 100MB.

When the map function emits a record, it is serialized into the main buffer and metada are stored into accounting buffers. In detail, there is the main buffer, called serialization buffer, where the outputs are collected. Two additional buffers constitute the accounting buffers; partition and index buffers maintain location of key and value for records to facilitate in-memory quick sort.

When either of the serialization or the metadata buffer exceeds a threshold, the contents of the buffers will be sorted and spilled to disk in the background.

In detail, the accounting buffer can store metadata for a predefined number of records before the spill thread is triggered. Each serialized record requires 16 bytes of accounting information in addition to its serialized size to effect the sort. Having `io.sort.mb` of memory available, only `io.sort.record.percent` percentage of it is dedicated to storing metadata.

$$buffer_{mtdt} = (\texttt{io.sort.mb}) \cdot (\texttt{io.sort.record.percent})$$

The maximum number of records (metadata records) this buffer can store is

$$maxRecs_{mtdt} = \left\lfloor \frac{buffer_{mtdt}}{16} \right\rfloor$$

On the other hand, the serialization buffer has a size given as follows

$$buffer_{srl} = \texttt{io.sort.mb} - buffer_{mtdt}$$

As already mentioned, when either the serialization buffer or the metadata exceed a threshold, their contents will be sorted and spilled to disk. The corresponding thresholds are given below

Spill threshold of the serialization buffer

$$spillThr_{srl} = (buffer_{srl}) \cdot (\texttt{io.sort.spill.percent})$$

Spill threshold of the accounting buffers

$$spillThr_{mtdt} = (maxRecs_{mtdt}) \cdot (\texttt{io.sort.spill.percent})$$

where the `io.sort.spill.percent` property determines the soft limit in either of the serialization and metadata buffers and is 0.80 by default. Once reached, a thread will begin to spill the contents to disk in the background.

After spilling has been triggered, the data is divided into partitions corresponding to the reducers that they will ultimately be sent to. Within each partition the background thread performs an in-memory sort by key and finally writes the data in a file to disk. We shall refer to each written file as a *spill*. Partitioning and sorting take place in the memory buffer and consequently no additional I/O is needed. After the map function has processed all the input records, there would be a number of *spill* files on disk. The number of the *spills* along with the size of each *spill* file, highly depend on the choice of the property values that regulate the spill thresholds. In most cases, the `io.sort.record.percentage` that determines the size *metaDataBuffer* of the metadata buffer, is chosen low enough (0.05 by default) and consequently the $spillThr_{mtdt}$ is the first one to be reached. In this case, for an input split of *inputRecs*, the number of the *spill* files is

$$numSpills = \left\lceil \frac{inputRecs}{spillThr_{mtdt}} \right\rceil$$

The I/O cost of the buffering phase equals to the cost of writing the whole input split, divided in spills, on disk that is,

$$\mathbf{cost}_{map}^{buf} = splitSize$$

2.1.3 Writing phase

The last phase is the writing phase where the map output is written to disk as a single file. This phase requires that the *spill* files are merged into one file. This procedure is the most I/O costly one. The map implementation of the merging algorithm tries to perform the minimum amount of work – in terms of minimum I/O – to end up with a single sorted output file at the end. To achieve this a rather complicated algorithm is implemented. Our presentation of this algorithm will be simplified and mainly concentrated on the I/O cost involved.

The merging algorithm is repeated as many times as the number of the partitions determined by the client-user. (Recall that the defined number of the reducers equals the number of partitions). The *spill* files (that actually consist the initial input split) are stored on the local disk. Each spill file is divided in sorted partitions that are equal to the number of the reducers. In each iteration of the merging algorithm the same partitions of each of the spill files are merged. That is, if there are 3 sorted partitions in each *spill* file, let us assume partitions A, B and C, the merging algorithm will be repeated 3 times and during each iteration the spill files contribute the same partition; the first iteration will merge the A partitions of each spill file, the second will merge the B partitions and the third will merge the C partitions. The set of the same partitions being merged during each iteration is called set of segments and its size is equal to the number of *spills*. In other words, each segment is a set of records that belong to the same partition and to the same *spill* file. For the rest of this section, we will assume that there is only one partition that is, only one Reducer. This assumption will result in only one iteration of the merging algorithm. However, it will not deprive us from the deduction of a representative formula for the I/O cost since in every case the same size of input data (*splitSize*) is read and written. The difference is that when there are a lot of partitions (more than one reducer) the same procedure (merging) is applied several times (equal to the number of partitions) on a subset of the input. Nonetheless, we expect the the total I/O cost is approximately the same as when there is only one iteration of the merging algorithm. In this fashion, we will gain a better comprehension of the procedure. Under this assumption, each *spill* file is equal to a segment and the size of it is

$$spillSize = \frac{splitSize}{numSpills}$$

Each iteration of the algorithm is conducted in a number of passes. Each pass takes a number files and produces a single one. Crucial enough is this number of passes since each pass is associated with reading and writing of intermediate files to disk. The algorithm itself attempts not only to minimize the number of merges (passes) but also to ensure that all, but the final pass, merge initial segments and not files that constitute the result of previous passes. Only the final pass is the one that merges the result of previous passes. In this fashion the cost of the intermediate I/O is reduced.

In detail, we define:

- *numSpills* $\longrightarrow$ the number of *spills* to be merged
- `io.sort.factor` $\longrightarrow$ the maximum number of *spill* files that can be merged during each pass. The value of this property is 100 by default.
- *spillSize* $\longrightarrow$ the size of each *spill* file

The number of the merging passes depends on the *numSpills* and on the value of the `io.sort.factor` property and is given by:

$$numMerges = \frac{numSpills - 1}{\texttt{io.sort.factor} - 1} + 1 \cdot \left\lceil \frac{\|1 - numSpills\|}{numSpills} \right\rceil$$

If $numSpills \equiv 1$ the above formula gives a number of merges equal to zero indicating that no merge is needed since the *spill* file constitutes the only and final output file of the map phase. In this case the total writing cost is

$$cost_{wr}^{sp=1} = 0$$

and the total I/O cost for the map phase is

$$cost_{map}^{sp=1} = cost_{map}^{r} + cost_{map}^{buf} + cost_{wr}^{sp=1} = cost_{map}^{buf}$$

In case of $numSpills > 1$ and $numMerges \equiv 1$ then only one pass is needed during which all of the *spill* files are merged into one file. The I/O cost equals the cost of reading and writing all the files that is, the cost of reading and writing the input split.

$$cost_{wr}^{m=1} = 2 \cdot splitSize$$

If $numSpills > 1$ and $numMerges > 1$, the merging algorithm follows a more sophisticated implementation. The rounds that take place after the first one and before the last one are called intermediate rounds. During the first round a certain number of *spill* files are merged. Each intermediate round merges `io.sort.factor` files into a single larger file, while the last round creates a final sorted file that consists the final output of the map phase. In detail, the first round merges only a certain number of the *numSpills*. We define

$$modVar = ((numSpills - 1) \mod (\texttt{io.sort.factor} - 1))$$

and the function

$$functMod(x) = \begin{cases} 0 & \text{for} \quad modVar = 0 \\ 1 & \text{for} \quad modVar > x \end{cases},$$

The number of *spills* that the first round merges is given as follows

$$\begin{aligned} numSpills^{rnd1}_{merge} &= functMod(modVar) \cdot (modVar + 1) \\ &\quad + (1 - functMod(modVar)) \cdot \texttt{io.sort.factor} \end{aligned}$$

The I/O cost equals to the cost of reading the $numSpills^{rnd1}_{merge}$ and writing them in a single sorted file to disk.

If $numMerges > 1$

$$cost^{rnd1}_{wr} = 2 \cdot numSpills^{rnd1}_{merge} \cdot spillSize$$

Each intermediate round merges `io.sort.factor` segments into a larger file, thus creating a new file on disk. Consequently, by the end of these rounds there can be $numMerges - 2$ files on disk of $\texttt{io.sort.factor} \cdot spillSize$ size each.

If $numMerges > 2$

$$cost^{intrm-rnd}_{wr} = 2 \cdot \texttt{io.sort.factor} \cdot spillSize$$

The total cost of the intermediate rounds is

$$cost^{intrm-rnds}_{wr} = (numMerges - 2) \cdot 2 \cdot \texttt{io.sort.factor} \cdot spillSize$$

The final round includes the reading and writing of all the segments that belong to this partition. Recall that the iterations of the merging algorithm will be as many as the number of partitions. Having assumed that there is only one partition, we can deduce the following formula for the I/O cost of the final round

$$cost^{final-rnd}_{wr} = 2 \cdot splitSize$$

Total cost of writing phase To evaluate the total cost of the writing phase, we should put all the above in a single formula. Defining the following boolean function

$$r(x) = \begin{cases} 0 & \text{for} \quad numMerges \leq x \\ 1 & \text{for} \quad numMerges > x \end{cases},$$

the total cost of the writing phase is

$$\mathbf{cost}^{wr}_{map} = (1 - r(0)) \cdot cost^{sp=1}_{wr} + r(0) \cdot \left[r(1) \cdot cost^{rnd1}_{wr} + r(2) \cdot cost^{intrm-rnds}_{wr} + cost^{final-rnd}_{wr} \right]$$

2.1.4 Total cost of the map task

All in all, the total cost of the map task includes the cost of reading the whole input split, the cost of writing the spill files and finally the cost of merging. In other words, it is the sum of the costs induced by the reading, buffering and writing phases.

$$\mathbf{cost}_{map} = \mathbf{cost}^{r}_{map} + \mathbf{cost}^{buf}_{map} + \mathbf{cost}^{wr}_{map}$$

2.2 I/O Cost Analysis of the Reduce Phase

The reduce phase is more complicated than the map phase, with a high degree of parallelization and overlap in the operations of a single reduce task. Recall that we have chosen to use the I/O as the performance metric of each phase; I/O is not so heavily affected by parallelization at the granularity of a single reduce task. As such, we will focus on the I/O-bound operations of a reduce task and extract a closed cost formula of its I/O cost.

The reduce task includes three phases; the shuffle/copy, sort and reduce phases. These phases are depicted in Figure 2.2. Note that the sort phase is often called merge phase. Same as every part of the map reduce process, the reduce task, and mainly its I/O cost, highly depends on the configuration properties that regulate the setup of the cluster. In the following analysis, these properties are clearly referred to when needed.

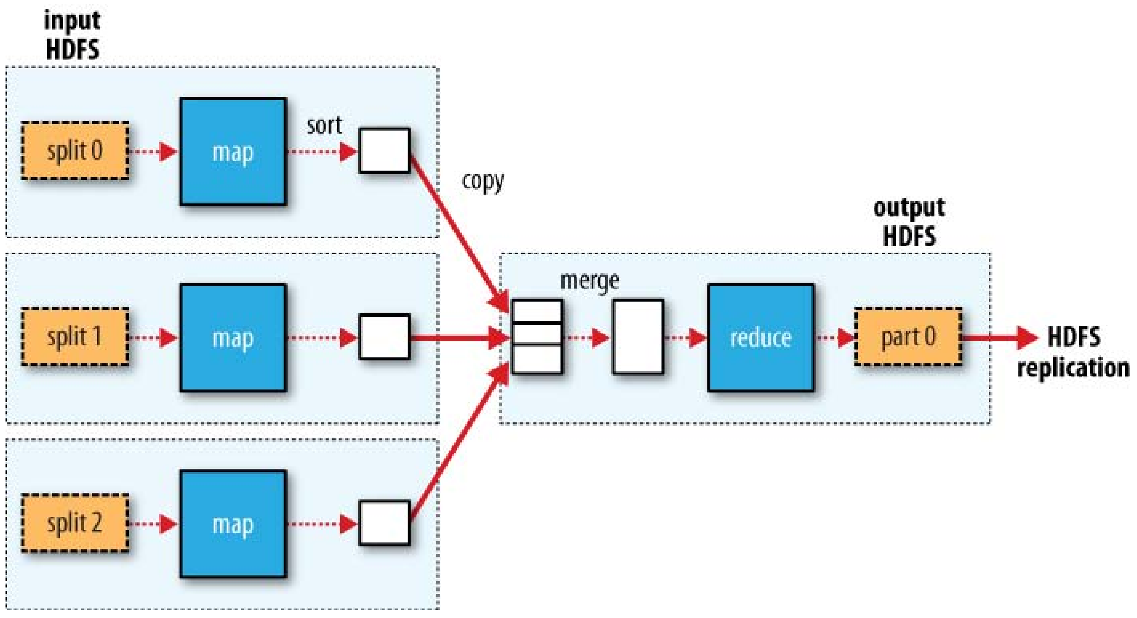

Figure 2.2: MapReduce data flow with a single reduce task

In every map-reduce job, there would be as many reduce tasks as the number of partitions and vice-versa. In other words, each reduce task is associated with a specific partition. The map output files, that actually constitute the input of the reducers, lie on the local disks of the nodes that ran the map tasks. These files are now needed by the nodes that are about to run the reduce tasks. Note that each map output file is partitioned and fully sorted that is, each map output file can include key-value pairs that belong to different partitions. Consequently, each reduce task needs to retrieve as input its particular partition from several map tasks across the cluster.

Shuffle/Copy phase During the copy phase each reduce task copies the map outputs locally. The retrieval of them is assigned to a number of copier threads that are associated with each reduce task. There is a predetermined small number of copiers that each reducer may have so that it can fetch the outputs in parallel. This number is defined by the configuration property `mapred.reduce.parallel.copies` and is 5 by default. Depending on the size of each map output (that a copier - thread is currently retrieving), it will be either copied to the reduce task-tracker's memory or to the local disk of the node currently running the reduce task. This decision is dictated by the buffer size of each reduce task. The property `mapred.child.java.opts` provides a memory of 200MB by default to each task (either map or reduce). However, only a percentage `mapred.job.shuffle.input.buffer.percent` (default 0.7) of it can be used for copying. Finally, only if the size of a map output is not greater than 25% of this available for copy buffer, would it be written to the memory. Otherwise, it is written to disk. From the above, it is clear that the configuration parameters highly affect the performance of the reduce task.

When the in-memory buffer reaches a threshold size, the in-memory outputs are merged and spilled to disk creating a new local file. The property `mapred.job.shuffle.merge.percent` (0.66 by default) determines this threshold. Moreover, when the number of files on disk increases above a certain threshold, a new merge thread is triggered and a number of disk files are merged into a new larger sorted one. In this fashion, time is saved from merging later on (during the sort phase).

Sort phase The sort phase constitutes the second phase of the reduce task. It starts after all the map outputs have been successful copied in memory and/or on disk. Its name is misleading since the sort phase merges the map outputs while the sorting has already taken place on the map side. However, the merging procedure is carried out in a way that that maintains the sort order.

The merge is conducted in rounds and regulated by a number of configuration properties aiming at the least I/O cost possible. A number of merge rounds can take place merging disk and memory files and resulting in the creation of intermediate files. However, rather than have a final round that merges these intermediate files into a single sorted file, the algorithm saves an access to disk by merging the files on the fly and providing the results directly to the reduce function of the next phase, the reduce phase.

Reduce phase During the reduce phase the reduce function is invoked for every pair of key-value in the sorted output of the final merge. The output of this phase is written directly to the output file system which is typically the HDFS. In the case of HDFS, since the tasktracker node is also running a datanode, the first block replica will be written to the local disk.

2.2.1 Disk-only reduce

Aiming at the comprehension of the inner processing, we will first assume the case where each map output is directly written to the local disk of the reducer and no in-memory copying takes place.

2.2.1.1 Shuffle/Copy phase

Since no in-memory copy takes place, every map output is written to the disk as soon as it is retrieved by the copier. However, as the copies accumulate on disk, a background thread merges them into larger, sorted files. This saves some time merging later on. This merge phase (that actually consists part of the shuffle phase) is related with I/O cost and its analysis is of great interest. The property that plays important role here is the so called `io.sort.factor` that determines the maximum number of streams to merge at once and by default is 100. While the number of the copied files on disk is less than the threshold given by $(2 \cdot \texttt{io.sort.factor} - 1)$, the copiers continue writing to disk. When this threshold is reached, the merger thread is triggered and merges `io.sort.factor` files (that are already stored on the disk). Notice that this merge has only one pass during which only `io.sort.factor` files are merged into 1 larger file. This merge may take place several times as the copiers keep copying new files until all the map outputs are successfully stored to disk.

In any case, the cost will be at least equal to the total size of the map outputs, since every map output is written at least once to the disk. However, additional cost is added when the merge thread is triggered. In such a case the merge procedure increases the final cost, since already stored files are read, merged and rewritten to disk. In detail, every merge iteration reads, merges and writes `io.sort.factor` files into a new file. The total additional cost depends on the times that this merge procedure takes place which in turn depends on the total number of the map outputs and the `io.sort.factor`, since the latter determines the merge threshold $(2 \cdot \texttt{io.sort.factor} - 1)$.

Our analysis is based on two basic assumptions as follows

1. the map outputs are considered to be of comparable size and second
2. the `io.sort.factor` property satisfies the following inequality

$$(\texttt{io.sort.factor})^2 \geq N$$

where N denotes the number of the map outputs. Both assumptions lead to a more balanced model. Particularly, the second assumption reassures that there will be no overlaps between the merge rounds that is, each merge round is applied to files that have not undertaken merging yet. Consequently, the intermediate files created by the merge iterations will be of comparable size. In this fashion we can deduce formulas that provide a general understanding of the I/O cost without implicating confusing details.

If there are N map output files, in the general case that each reduce task receives N map partitions of approximate equal size (in bytes), let us call it *mapOutputSize*, then the times the merge phase will take place is

$$numMerges_{shuf} = \frac{N-1}{\texttt{io.sort.factor}-1} - 1$$

The I/O cost of the shuffle/copy phase, as already mentioned, will be at least equal to the total size of all the map outputs and will include the cost of the additional merge iterations. Each merge iteration will read `io.sort.factor` outputs and merge them into a larger file. Consequently, the I/O cost of each iteration is equal to $\texttt{io.sort.factor} \cdot mapOutputSize$.

Taking all these into account and under the assumption that $(\texttt{io.sort.factor})^2 \geq N$, the total I/O cost of the copy/shuffle phase obeys to the following formula:

$$cost^{shuf}_{reduce} = N \cdot mapOutputSize + 2 \cdot numMerges_{copy} \cdot (\texttt{io.sort.factor} \cdot mapOutputSize)$$

When the copy procedure is finished, along with the background merging that might have taken place, the total number of the files stored on disk will be:

$$numFiles^{dsk}_{shuf} = \left(N - numMerges_{shuf} \cdot \texttt{io.sort.factor}\right) + numMerges_{shuf}$$

It must be clearly stated that the end of the shuffle phase results in a number of files that are either files yielded by merging either initial unmerged map outputs. As the last formula denotes, there will be $N - numMerges_{shuf}$ and $numMerges_{shuf}$ files of each category. The size of each merged file is given by

$$fileSize^{merge}_{shuf} = \texttt{io.sort.factor} \cdot mapOutputSize$$

since each merged file contains so many map outputs as the `io.sort.factor` denotes, while the size of each unmerged file is equal to the *mapOutputSize*.

2.2.1.2 Sort phase

When all the map outputs have been copied, and probably partially merged, there are no more than $numFiles^{dsk}_{shuf}$. The reduce process moves into the so called sort phase. The sort phase actually merges the files left by maintaining their sort order. In detail, the sort phase applies the merging algorithm on these files yielding one fully sorted final file that constitutes the input to the reduce phace. Depending on the number of the disk files and the predefined `io.sort.factor`, there can be more than just one merging rounds. Particularly, the number of the merge-rounds is given as follows:

$$mergeRnds = \frac{numFiles^{dsk}_{shuf}}{\texttt{io.sort.factor}} + 1$$

Each merge round reads a number of files and merges them into a larger file. Consequently, I/O cost is present. However, the merging algorithm is implemented in a way that the least possible cost is yielded. Particularly, during the first round only a small number of files are merged. During subsequent rounds `io.sort.factor` files are merged including only files previously unmerged and finally, during the final merge no final file is constructed since the merge is conducted on the fly while feeding the reduce function of the last phase; the reduce phase.

If $mergeRnds \equiv 1$, which is the case when the $numFiles^{dsk}_{shuf}$ is less or equal to the `io.sort.factor`, then the I/O cost equals the cost of reading all the $numFiles^{dsk}_{shuf}$ plus the cost of writing them to a final file. However, I/O cost is saved by not writing them to a final file, rather implementing the merge on the fly while feeding the reduce phase at the same time. In this case the I/O cost is

$$cost^{m=1}_{srt} = 2 \cdot totalInputSize = 2 \cdot N \cdot mapOutputSize$$

If $mergeRnds > 1$ then the I/O cost evaluation includes additional computations. Let us assume that the number of the rounds is R. Under the same assumption we introduced in the copy phase that is, $(\texttt{io.sort.factor})^2 \geq N$, where N is the initial number of the map outputs, the total cost can be computed if we refer to each round separately. During the first round only a number of the files on disk are merged. This number is given by the formula $[(numFiles^{dsk}_{shuf} - 1) \mod (\texttt{io.sort.factor} - 1)] + 1$ and the assumption $(\texttt{io.sort.factor})^2 \geq N$ reassures that these files are map outputs that have not undertaken merging during the copy phase.

First round

$$cost^{rnd1}_{srt} = 2 \cdot \left[\Big((numFiles^{dsk}_{shuf} - 1) \mod (\texttt{io.sort.factor} - 1) \Big) + 1 \right] \cdot mapOutputSize$$

Another important notice is that the assumption $(\texttt{io.sort.factor})^2 \geq N$ leads to such a merging algorithm that all the rounds except the first will merge files that are the result of the merge procedure during the copy phase. These rounds, except the final, merge `io.sort.files` each. However, each of these rounds merges files that have not been used during previous rounds. In this fashion, the algorithm tends to minimize the total I/O cost.

If $mergeRnds > 2$, then except from the first and the final round, intermediate merge rounds also take place. The number of these rounds is given as

$$mergeRnds_{intrm} = mergeRnds - 2$$

Each intermediate round merge `io.sort.factor` files yielding cost that is equal to

$$\begin{aligned} cost_{srt}^{intrm-rnd} &= 2 \cdot \texttt{io.sort.factor} \cdot fileSize_{shuf}^{merge} \\ &= 2 \cdot (\texttt{io.sort.factor})^2 \cdot mapOutputSize \end{aligned}$$

Total cost of intermediate merge rounds

$$\begin{aligned} cost_{srt}^{intrm-rnds} &= mergeRnds_{intrm} \cdot cost_{srt}^{intrm-rnd} \\ &= (mergeRnds - 2) \cdot (2 \cdot (\texttt{io.sort.factor})^2 \cdot mapOutputSize) \end{aligned}$$

During the final merge round, all the merged files are read (a size equal to the total size of the initial map outputs). However, no final file is created, since the merging is conducted on the fly while the merged results are propagated to the reduce phase. Consequently,

the cost of final round is

$$\begin{aligned} cost_{srt}^{final-rnd} &= totalInputSize \\ &= N \cdot mapOutputSize \end{aligned}$$

Total cost of the sort phase Defining a boolean function as follows:

$$r(x) = \begin{cases} 0 & \text{for} \quad mergeRnds \leq x \\ 1 & \text{for} \quad mergeRnds > x \end{cases},$$

the total I/O cost of the sort phase can be written as

$$cost_{reduce}^{srt} = r(1) \cdot cost_{srt}^{rnd1} + r(2) \cdot cost_{srt}^{intrm-rnds} + cost_{srt}^{final-rnd}$$

2.2.1.3 Reduce phase

Recall that the reduce phase starts taking place during the final merge round of the sort phase. The merge in done on the fly and the results are directly entered to the reduce function. The latter processes them based on the user's implementation and the final output is directly written to the HDFS filesystem. Consequently, no bytes are read or written locally and the I/O cost is equal to zero.

$$cost_{reduce}^{rdc} = 0$$

2.2.1.4 Total cost of the reduce task

The total cost of the reduce task, as extracted in the case of disk-only reduce, is the following sum

$$cost_{reduce} = cost_{reduce}^{shuf} + cost_{reduce}^{srt} + cost_{reduce}^{rdc}$$

2.2.2 Reduce on both memory and disk

Contrary to the simplified version we presented, in a real reduce task things are more complicated. In fact, the reduce task utilizes the available memory before access to the local disk is needed. This optimized behavior results in reduced I/O cost since a great number of trips to disk are avoided. The following sections will clarify the real process behind each reduce task along with the mutations introduced to the I/O cost formulas of the disk-only case presented in Section 2.2.1. For simplicity, we still assume that all map outputs received by the reducer are of approximately equal size.

2.2.2.1 Copy/Shuffle Phase

When a copier thread retrieves one of the map outputs a decision is taken on where this output will be written/copied; in memory or on disk. There is a number of configuration properties that regulate this decision;

- the `mapred.job.shuffle.input.buffer.percent` property which determines the percentage of memory- relative to the maximum heapsize that can be allocated to storing map outputs during the shuffle. Its default value is 0.70.
- the `mapred.job.shuffle.merge.percent` property, which determines the memory threshold for fetched map outputs before an in-memory merge is started. It expressed as a percentage of memory allocated to storing map outputs in memory and its default value is 0.66.

Every reduce task is given a memory of a prespecified size. Let us call it *availMem*. However, only a percentage of this memory is provided for the purposes of the copy/shuffle phase and is given by

$$maxMem_{shuf} = (\texttt{mapred.job.shuffle.input.buffer.percent}) \cdot availMem$$

The map output is copied in memory only if its size is less than the threshold given by

$$maxMapSize = maxMem_{shuf} \cdot \texttt{max\_single\_shuffle\_seg\_fraction}$$

The `max_single_shuffle_seg_fraction` is the percentage of the in-memory limit that a single shuffle can consume and by default is equal to 0.25. A size of less than *maxMapSize* will allow map output to be written in memory, otherwise it is propagated to disk. In a real scenario, map outputs constitute segments of different size and copiers access both memory and disk. Such a case is not suitable for analysis since I/O cost depends on the output size, which in turn varies and depends on a number of job configuration parameters. However, map reduce jobs are usually conducted in machines that provide memory of adequate size allowing the map output to be below this threshold. Following this approach, our analysis will be concentrated on the case where the size of each map output is below the threshold.

Same as in Section 2.2.1, we assume N map outputs of *mapOutputSize* size each. As soon as the copiers are assigned with a map output, they start writing it in a file in memory. When the accumulated size of the copied in memory outputs reaches the following threshold

$$inMemThr = \texttt{mapred.job.shuffle.merge.percent} \cdot maxMem_{shuf}$$

a background thread merges the stored in memory files yielding a new file on disk. The number of files stored in memory and merged is given by

$$numOutputs_{merged}^{in-mem} = \left\lceil \frac{inMemThr}{mapOutputSize} \right\rceil$$

When $numOutputs_{merged}^{in-mem}$ files are written in memory, the background thread merges and writes them into a bigger file on the disk. For a size of N map outputs the in memory merge procedure will be repeated as many times as the following formula denotes

$$mergeItrs_{in-mem} = \frac{N}{numOutputs_{merged}^{in-mem}}, \quad \text{where we keep the quotient}$$

Note that each iteration has only one pass over $numOutputs_{merged}^{in-mem}$ outputs yielding one new file on disk. The size of this file is determined as follows,

$$fileSize_{merged}^{in-mem} = numOutputs_{merged}^{in-mem} \cdot mapOutputSize$$

As the shuffling phase continues, map outputs accumulate in memory, in-memory merge is triggered and new files are stored on disk. This may repeat several times and result in a great number of files. To save time merging later on, another background thread merges the files stored on disk. This is the same procedure as the one mentioned in Section 2.2.1 where the map outputs were directly written to disk. The difference is that the files written to disk are not the initial map outputs. They are merged files and each one is a result of the in-memory merge of $numOutputs_{merged}^{in-mem}$ map outputs. Recalling from the case of Section 2.2.1, when the number of disk files becomes greater than $(2 \cdot \texttt{io.sort.factor} - 1)$ the thread responsible for disk merge is triggered and merges `io.sort.factor` files. In a shell, the map outputs are first written in memory. When the memory merge threshold *mergeInMemThreshold* is reached, they are merged and spilled in a file to disk. When $(2 \cdot \texttt{io.sort.factor} - 1)$ files are written on disk they are merged in a larger file. The same assumption that is, $(\texttt{io.sort.factor})^2 \geq N$ holds here as well. This assumption reassures that during each on-disk merge only files not previously (during previous iterations) merged are being merged. In other words, each file on disk takes part in one merge iteration maximum. The on-disk merge is related with the following equation

$$mergeItrs_{onDsk} = \frac{mergeItrs_{in-mem} - 1}{\texttt{io.sort.factor} - 1} - 1$$

Both in memory and disk merge iterations result in I/O cost. The sum of those constitutes the total cost of the shuffle phase. Examining each cost separately we can extract the following formulas

$$\begin{aligned} costMerge_{in-mem} &= mergeItrs_{in-mem} \cdot numOutputs^{in-mem}_{merged} \cdot mapOutputSize \\ &= \left(\frac{N}{numOutputs^{in-mem}_{merged}} \right) \cdot \left(\left\lceil \frac{inMemThr}{mapOutputSize} \right\rceil \right) \cdot mapOutputSize \end{aligned}$$

The cost of each in-memory merge iteration equals the cost of writing $numOutputs^{in-mem}_{merged}$ map outputs to disk. No reading is conducted. On the contrary, the cost evaluation of the disk merge is more complicated. Each disk merge reads `io.sort.factor` of files (produced by in-memory merge), merges and writes them into a larger file. The size of each such file is

$$fileSize^{onDsk}_{merged} = \texttt{io.sort.factor} \cdot fileSize^{in-mem}_{merged}$$

and the cost of each merge on disk is

$$costMerge_{onDsk} = 2 \cdot mergeItrs_{onDsk} \cdot \texttt{io.sort.factor} \cdot fileSize^{in-mem}_{merged}$$

Consequently, the total I/O cost of the shuffle/copy phase is

$$\mathbf{cost}^{shuf}_{reduce} = costMerge_{in-mem} + costMerge_{onDsk}$$

By the end of the shuffle phase there is a number of files on disk and probably unmerged outputs in memory. The number of unmerged outputs lying in memory is

$$outputs^{in-Mem}_{shuf} = N - mergeItrs_{in-mem} \cdot numOutputs^{in-mem}_{merged}$$

On the disk, there are files produced by disk merging thread and probably unmerged files, that is files created by the in-memory merge procedure.

$$files^{onDsk}_{shuf} = mergeItrs_{onDsk} + [mergeItrs_{in-mem} - (mergeItrs_{onDsk} \cdot \texttt{io.sort.factor})]$$

2.2.2.2 Sort phase

After the shuffle phase is finished, the reduce process moves on to the sort phase. The latter merges the files left on disk and in memory in a fashion that yields the least possible I/O cost. In details, the sort phase includes the last merge of all the files, a phase before they are propagated to the actual reduce function. A merging algorithm can result in the creation of several temporary files (depending on the number of rounds) before the last one is produced. However, in the sort phase a last file where all the data is stored, is never created. Rather than having a final round that merges these temporary files into a single sorted file, the algorithm saves a trip to disk by implementing the final merge on the fly and directly feeding the reduce function with the results. This final merge can come from a mixture of in-memory and on-disk segments. Particularly, a decision is made whether the remained in-memory outputs will be kept in memory or will be merged in one file and written to disk before the last final merge takes place. This decision aims at feeding the reduce function with less than `io.sort.factor` files. All in all, if the `io.sort.factor` is greater than the number of files on disk then the in-memory segments are spilled to disk, otherwise they are kept in memory and take part in the last merge as a separate file each. Defining a boolean function as follows:

$$spillToDsk(files_{shuf}^{onDsk}) = \begin{cases} 1 & \texttt{io.sort.factor} > files_{shuf}^{onDsk} \\ 0 & otherwise \end{cases}$$

<u>If $spillToDsk(files_{shuf}^{onDsk}) \equiv 1$</u>, the in-memory files are spilled to disk and merged along with the remaining files on disk. The merge is conducted with one pass, no intermediate rounds (no temporary files) and all the files on disk (increased by one due to the spill) are propagated to the reduce function. The cost in this case equals the cost of writing the in-memory merged segments and reading the total files on disk

If $spillToDsk(files_{shuf}^{onDsk}) \equiv 1$

$$cost_{srt}^{spill} = outputs_{shuf}^{in-Mem} \cdot mapOutputSize + N \cdot mapOutputSize$$

<u>If $spillToDsk(files_{shuf}^{onDsk}) \equiv 0$</u>, the in-memory segments are kept in memory and merged as separate file each along with the files on disk during the final merge. In this case the number of files on disk is greater than the `io.sort.factor` (*spillToDsk* function is equal to 0) and it is certain that the merging algorithm will have more than one rounds/passes. The total number of files to be merged is

If $spillToDsk(files_{shuf}^{onDsk}) \equiv 0$

$$files_{merged}^{srt} = files_{shuf}^{onDsk} + outputs_{shuf}^{in-Mem}$$

The number of the rounds of the merge process is given by,

$$mergeRnds_{srt} = \frac{files^{srt}_{merged} - 1}{\texttt{io.sort.factor} - 1} + 1$$

However, the last round does not yield temporary files since its results are computed on the fly while they are being propagated to the reduce function. The assumption $(\texttt{io.sort.factor})^2 \geq N$ once again reassures that all the rounds except the first will merge files that are the result of the on-disk merge procedure during the shuffle/copy phase. The first round merges the files left in-memory along with some files on disk which are the the result of the in-memory merge but have been left un-merged on disk. The intermediate rounds, except the final, merge `io.sort.factor` files each. However, each of these rounds merges files that have not been used during previous rounds. In this fashion, the algorithm tends to minimize the total I/O cost. A thorough analysis of the I/O cost of each round follows.

During the first round only a number of the files on disk are merged. This number is given by the formula

$$files^{rnd1}_{merged} = [(files^{onDsk}_{shuf} - 1) \mod (\texttt{io.sort.factor} - 1)] + 1 + outputs^{in-Mem}_{shuf}$$

The cost of this round is

$$\begin{aligned} cost^{rnd1}_{srt} &= 2 \cdot [((files^{onDsk}_{shuf} - 1) \mod (\texttt{io.sort.factor} - 1)) + 1] \\ &\quad \cdot numOutputs^{in-mem}_{merged} \cdot mapOutputSize \\ &\quad + 2 \cdot outputs^{in-Mem}_{shuf} \cdot mapOutputSize \end{aligned}$$

Each intermediate round claims

$$cost^{intrm-rnd}_{srt} = 2 \cdot \texttt{io.sort.factor} \cdot fileSize^{onDsk}_{merged}$$

The total cost of the intermediate rounds is

$$cost^{intrm-rnds}_{srt} = (mergeRnds_{srt} - 2) \cdot cost^{intrm-rnd}_{srt}$$

Finally, the last round takes place. This is the round with no final file writing, since the temporary files are merged on the fly and the results are being sent to the reduce function.

$$cost^{final-rnd}_{srt} = totalInputSize = N \cdot mapOutputSize$$

In the same fashion as in the simple scenario of Section 2.2.1, we can define a boolean function as follows:

$$r(x) = \begin{cases} 0 & \text{for} \quad mergeRnds_{srt} \leq x \\ 1 & \text{for} \quad mergeRnds_{srt} > x \end{cases},$$

and a formula that computes the total I/O cost of the sort phase (always under the assumption made) is

$$\begin{aligned} \mathbf{cost}^{srt}_{reduce} &= \left(1 - spillToDsk(files^{onDsk}_{shuf})\right) \cdot (r(1) \cdot cost^{rnd1}_{srt} + r(2) \cdot cost^{intrm-rnds}_{srt} + cost^{final-rnd}_{srt}) \\ &\quad + spillToDsk(files^{onDsk}_{shuf}) \cdot cost^{spill}_{srt} \end{aligned}$$

2.2.2.3 Reduce phase

The reduce phase is conducted in the same fashion as the one in the case of Section 2.2.1. No bytes are read or written locally and consequently it results in no I/O cost.

$$\mathbf{cost}^{rdc}_{reduce} = 0$$

2.2.2.4 Total cost of the reduce task

The total cost of the reduce task, as extracted in the case of both in-memory and disk reduce, is the following sum

$$\mathbf{cost}_{reduce} = \mathbf{cost}^{shuf}_{reduce} + \mathbf{cost}^{srt}_{reduce} + \mathbf{cost}^{rdc}_{reduce}$$

2.3 Discussion

In this chapter we studied the dataflow of a map and reduce task and extracted a cost model that evaluates the I/O cost related to each task. Our analysis was based on two basic assumptions

1. The map outputs that a reducer fetches from the mappers are of comparable size. This assumption let us use a representative size, denoted as *mapOutputSize*, for each retrieved map output and facilitated the cost analysis of the reduce task throughout Section 2.2. It is important to note that this assumption holds only when the distribution of the key values is uniform. However, in almost every case the user is prompted to deploy a sampling partitioner which samples the key space and partitions the records during the map phase in a fashion that results in fairly even partition sizes. This, in turn, ensures that the outputs presented to each reducer is approximately even. Consequently, our assumption does not degrade the quality of our model.

2. We also assumed that the second power of the value of the property `io.sort.factor` is greater than the number of the participating nodes. That is,

$$(\texttt{io.sort.factor})^2 \geq N$$

   Actually, this assumption ensures that during any merge procedure, the files that are being merged are files that have not undertaken previous merge and thus all the files to be merged are of comparable size. It is worth-mentioning that this assumption holds in almost every Map/Reduce job, since the values of the property `io.sort.factor` are chosen appropriately.

We claim that the proposed cost model is able to provide results that efficiently approach the real I/O cost values of a map and a reduce task. Chapter 4 challenges the validity of our model with a series of experiments.

Chapter 3

Join Evaluation

3.1 Introduction

In this chapter we present the two main join implementations already included in the Hadoop Map/Reduce framework: the *reduce side* join and the *map side* join. Additionally, we introduce an optimized version of the simple reduce side join which engages a Bloom filter on the map phase. Based on the cost model we deducted in Chapter 2, we focus our analysis on the I/O cost that each join algorithm induces.

3.2 Reduce Side Join

Reduce side join is the most general join approach implemented in the context of the standard map/reduce framework. As its own name implies, the join is conducted during the reduce phase. Its generality is based on that it comes with no restrictions regarding the input datasets. However, its biggest drawback is its I/O cost. In the following sections, we will present the reduce side join algorithm and analytically compute its I/O cost based on our cost formula.

3.2.1 Description

Assuming two input datasets, reduce side join computes their join deploying the standard map/reduce framework. For simplicity, we will present the case of the inner join; the map/reduce output is a new set of records (new dataset) computed by combining records of the two input datasets that have the same join key value. To further simplify our analysis, we assume that the first dataset has the join key as its primary key that is, no more than one record can have the same key.

The basic idea behind the reduce side join is that the mapper tags each record with its source, and uses the join key as the map output key so that the records with the same key are grouped for the reducer. Tagging is necessary since it will ensure that sorting and grouping of the records, during the reduce phase, are conducted in a way that will serve the join needs.

In detail, during the map phase, and more precisely in the map function, the following take place

1. Input record source tagging: as the mapper reads each record it stores its source in a *tag* value.
2. Map output key computation: computing the map output key amounts to choosing the join key. Recall that the map function receives $\langle key, value \rangle$ pairs sequentially as they are read from the input file one by one. However, depending on the job configuration, the *key* may not respond to the wanted join key. It is the user's responsibility to implement a map function such that will output as a key those column/fields values that are the wanted ones. After the computation of the right key, the map function should insert the *tag* to the new key.
3. Map output value computation: computing the map output value is equivalent to performing projecting/filtering work in a SQL statement (through the select/where clauses). After the new value is computed, the map function should also tag the new value in the same way as in the new key.

The alterations that a $\langle key, value \rangle$ pair undertakes are depicted as follows

$$\langle key, value \rangle_{map-input} \rightarrow \langle newKey, newValue \rangle \rightarrow \langle newKey + tag, newValue + tag \rangle_{map-output}$$

During the reduce phase, the map outputs that have the same key, that is *newKey*, are presented to the same reducer. Consequently, all the records that have the same value of the join key will be processed by the same reduce task. When each reducer receives a set of records (likely from multiple map tasks) they are being sorted and grouped before entering the reduce function. The records are primarily sorted on *newKey* and secondarily on *tag*. First, the *newKey* of each record is compared and in case of ties, sorting is done based on the *tag*. On the other hand, grouping depends only on the value of the *newKey*.

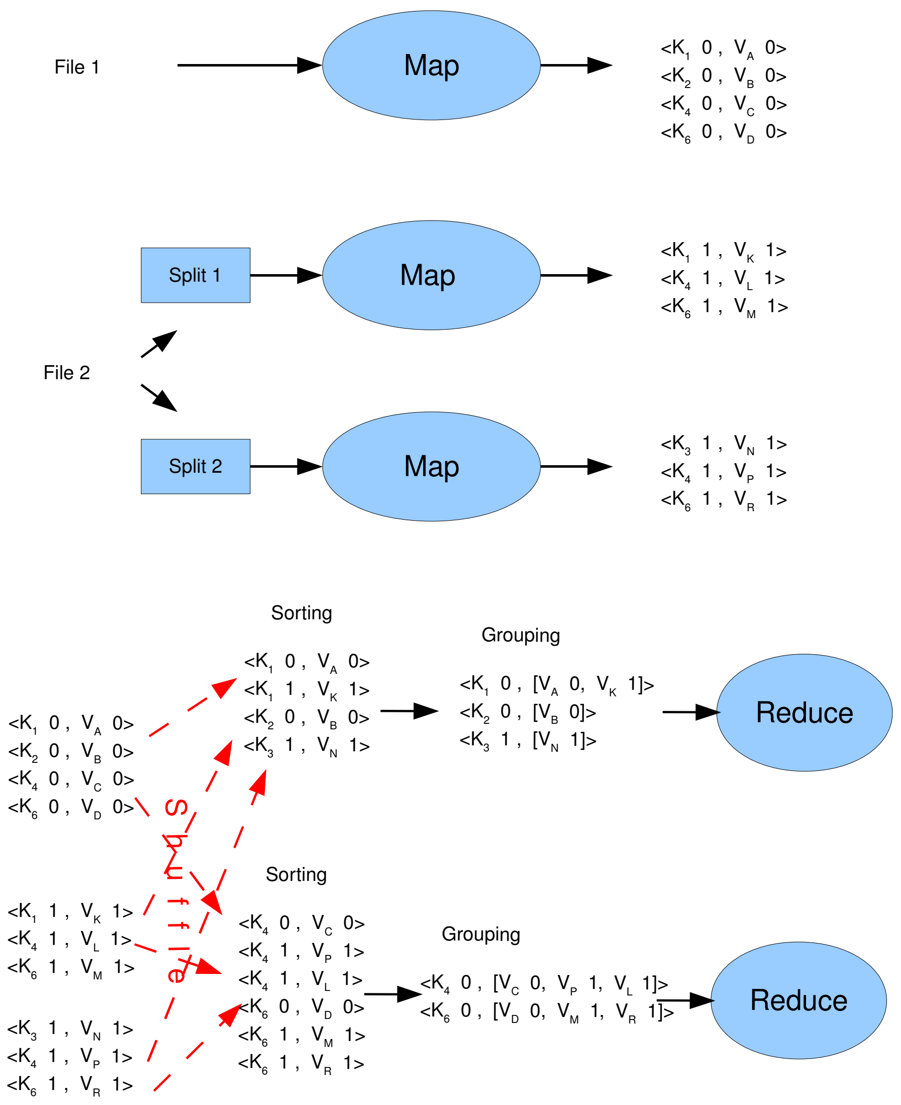

Figure 3.1: Tagging used for efficient sorting and grouping in the reduce side join

An example of how tagging takes place during the reduce side join is depicted in Figure 3.1. The map function computes the $\langle newKey, newValue \rangle$ pairs and tags them with either 0 or 1 depending on their originating source. Consequently, the final map output is of the type $\langle newKey\ tag, newValue\ tag \rangle$. During shuffling, the map outputs that have the same key, that is the same *newKey* value, are fetched by the same reduce task. Note that partitioning of the map outputs is based only on the *newKey* and not on the *tag*. If it was the case that partitioning was done based on the composite key (*newKey tag*), then records from different sources would result in different reducers despite having the same *newKey*. On the other hand, it is essential that sorting is done based on the composite key. When the fetched outputs are being merged during the reduce phase, they are being sorted as follows; first, the *newKey* keys are compared and in case of ties, the corresponding *tag* parts are examined. This approach results in an increasing order of the composite key and, moreover, it ensures that records from *file 1* precede the records from *file 2* as far as the same key is concerned. However, grouping is conducted examining only the *newKey* part. In this fashion, the records that have the same *newKey* are grouped together in a list, while the corresponding $(newValue\ tag)$ composite values populate the list maintaining the sort order.

After the inputs have been sorted and grouped, the created groups form the actual input to the reduce function. The latter is responsible for joining records that belong to the same group. Note that each group contains the records from the sources that have the same *newkey*. The code of the Hadoop framework implements the reduce-side join by regrouping the contents of each group based on the *tag* of the $(newValue\ tag)$ pairs of the group. In the example of Figure **??**, each group is subgrouped based on the tags 0 and 1. The subgroups are buffered in memory and their Cartesian product is computed. This approach serves efficiently the case where there are duplicates in either of the sources. However, it suffers from memory limitations, since in case a group has a large number of records, in-memory storage may be prohibitive.

In our presentation, we will assume that the first source, that is *file 1*, has no duplicates since it has the join key as primary. As a result, we can avoid buffering in memory and use another approach for implementing the join. Our assumption ensures that every key in *file 2* has at most one matching record in the first dataset (*file 1*). The reduce function receives each group and reads in its value list sequentially. Recall that the value list of each group consists of $(newValue\ tag)$ pairs/records that have the same key (same group) and moreover they are sorted so that the corresponding records of *file 1* comes first. Consequently, the reduce function can easily detect and process only these groups that have as first value, a value tagged with 0 and furthermore they do have further elements. In this case, the join is performed by outputting the *newKey* of the group along with the combination of the first value with each of the rest (that are tagged with 1). No memory buffering is needed since only the first value is stored while the subsequents are read sequentially from the input stream.

From the above, it is clear that it is only during the reduce function that disjoint records are discarded. In other words, both datasets have to go through the map/reduce shuffle. For small datasets the corresponding I/O cost may be small. However, when it comes to large datasets, as it is in most cases for a map/reduce job, the I/O cost may increase tremendously. In the following section, we present the relevant I/O based on the formulas we introduced in Chapter 2 of the current project.

3.2.2 I/O cost analysis

3.2.2.1 Map phase

The reduce side join uses the map phase for tagging the records based on their source dataset. Tagging increases the map output bytes since it imposes additional bytes on each record. The *tag* is added on both the outputted key and value of each record. Moreover, the user may implement a map function that outputs as key and value only some of the initial fields/columns of each input record amounting to projection/selection. In this case, the map output differs even more from the initial input in size. All these should be taken into account, since they effect the I/O cost of the map phase.

The map function reads the input split record by record, applies the user defined code on each one and propagates the results to the following phases, that is buffering in memory and writing to local disk. In Chapter 2 we elaborated on the I/O cost associated with the map task. However, we assumed that the map function is the identity one, that is, it outputs the records without changing them. Consequently, the input to the following phases equaled the size of the input split. However, in an actual map/reduce job, map tasks apply map function that does change the input and results in outputs different in size than the initial input split. This is illustrated in Figure 3.2.

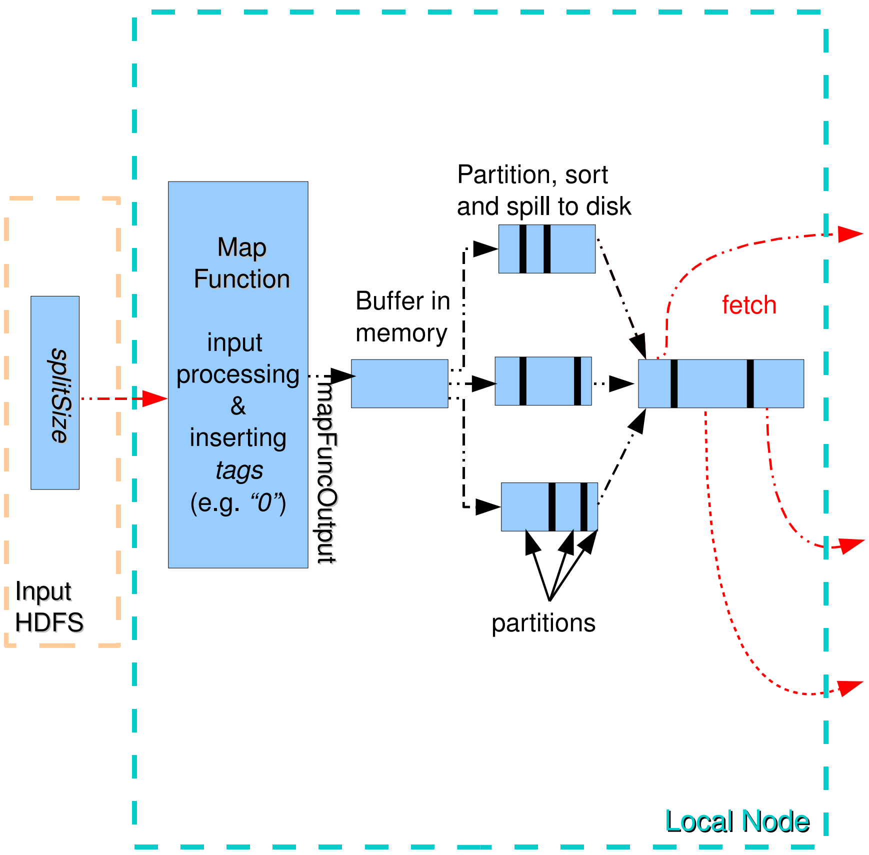

Figure 3.2: Full map task of a reduce side join

Reading phase The input split of size *splitSize* is read record by record and enters the user defined map function. In the general case where the user applies projection/selection on the fields we can assume that the output of the map function is a percentage *mapFuncProcessing* of the initial *splitSize*

$$mapFuncOutput = splitSize \cdot mapFuncProcessing$$

The percentage *mapFuncProcessing* covers the total change that the map function imposes on the *splitSize*. However, we will account for the worst case, where all the fields of each record pass intact to the buffering phase without selection or projection. Nonetheless, tagging takes place and increases the total size of each record. Consequently, the tag choice is of high importance. In our tests we assumed a simple string of type 0 or 1. Each record is tagged twice, once for the *key* and once for the *value* part. Introducing as *t* the additional bytes added to each record and as *numOfInputRecords* the number of the records included in the input split, the output size of the map function can be stated as follows

$$mapFuncOutput_{RSJ-simple} = splitSize + numOfInputRecords \cdot t$$

For our test case where each record is assigned 2 tags, one for the key and one for the value, the byte overhead is 4 bytes per record. As a result, if the input split consists of X records and the size of each record is b_X, the output of the map function is $(b_X + 4) \cdot X$, that is $4X$ bytes overhead. Of course, this is a rather small size, but in case of very large datasets, it can reach larger sizes. Refering to Chapter 2 and Section 2.1.1 the cost of the reading phase is

$$cost_r^{RSJ} = 0$$

Buffering and writing phases The I/O cost analysis of these phases is exactly the same as the one presented in Chapter 2 and Section 2.1.2. The only difference is that the input is no more equal to *splitSize* since the map function is not the identity one. Replacing the *splitSize* with the *mapFuncOutput* we can retrieve the right formulas. In detail, the corresponding formulas are altered as follows.

Cost of buffering

$$cost_{buf}^{RSJ} = mapFuncOutput_{RSJ} = splitSize + numOfInputRecords \cdot t$$

Cost of writing

$$\begin{aligned} cost_{wr}^{RSJ} &= r(0) \cdot cost_{wr}^{sp=1} + r(0) \cdot \left[r(1) \cdot cost_{wr}^{rnd1} + r(2) \cdot cost_{wr}^{intrm-rnds} + cost_{wr}^{final-rnd} \right] \Rightarrow \\ cost_{wr}^{RSJ} &\leq cost_{wr}^{rnd1} + cost_{wr}^{intrm-rnds} + 2 \cdot mapFuncOutput_{RSJ} \end{aligned}$$

Total I/O cost of map task The total I/O cost of a map task during the reduce side join is given by the formula

$$\mathbf{cost}_{map}^{RSJ} = cost_r^{RSJ} + cost_{buf}^{RSJ} + cost_{wr}^{RSJ} = mapFuncOutput_{RSJ} + cost_{wr}^{RSJ} \Rightarrow \tag{3.1}$$

$$\mathbf{cost}_{map}^{RSJ} \leq 3 \cdot mapFuncOutput_{RSJ} + cost_{wr}^{rnd1} + cost_{wr}^{intrm-rnds} \tag{3.2}$$

The inequality (3.2) reveals that there is an upper bound on the total map cost during the reduce side join which is also proportional to the size of the *mapFuncOutput*$_{RSJ}$. This remark suggests that the output of the map function is an important determinant of the total cost and consequently implies that the map function can be employed to reduce the cost. Based on this point, in Section 3.3, we introduce an optimized version of the reduce side join and we claim that it results in lower cost.

3.2.2.2 Reduce phase

The reduce task induces the most I/O cost. Among the three phases that each reduce task consists of, that is, the shuffle/copy, the sort and the reduce phase, the shuffle/copy and sort are the most expensive ones. Refering to the extensive analysis presented in Chapter 2, during the shuffle phase threads running in parallel copy the fetched map outputs either in memory or on disk. In both cases threads may run several times and merge files resulting in several trips to the local disk. Moreover, during the sort phase additional merging may occur. The impact of these phases on the I/O cost highly depends on the size of the fetched map ouputs. The larger the size of the map outputs fetched during shuffle is, the larger the I/O cost is as well.

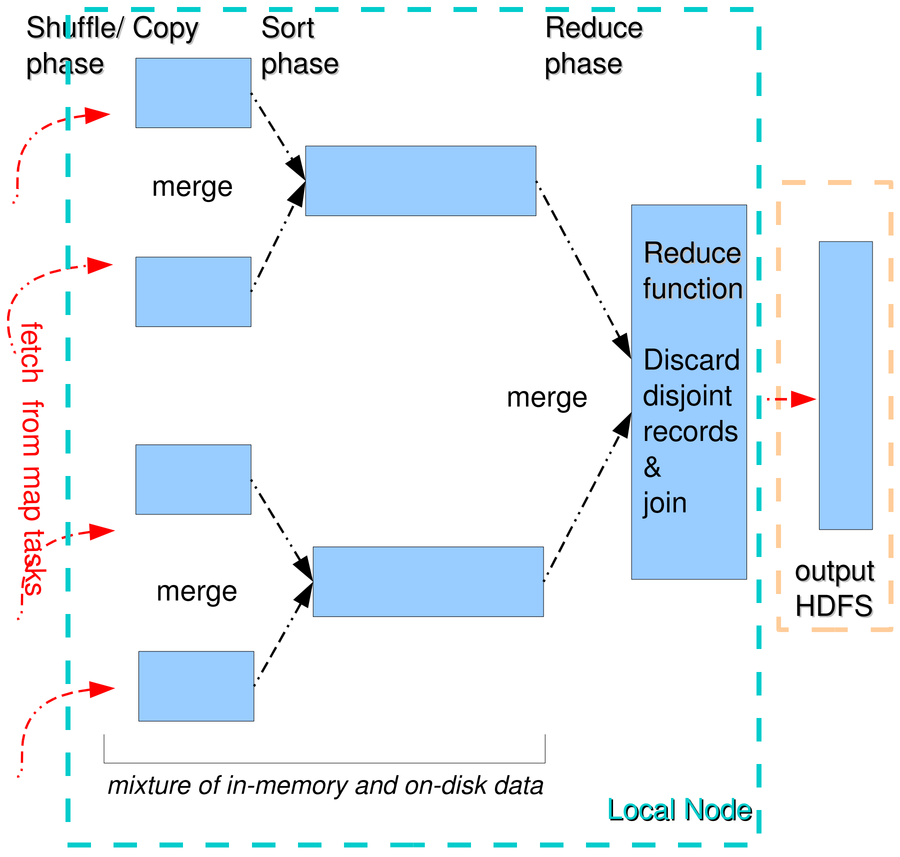

Figure 3.3: Full reduce process of a reduce side join

To deduce a general formula for the I/O cost of the reduce phase, we will refer once again to the corresponding formulas introduced in Chapter 2. As it is illustrated in Figure 3.3, the reduce side join follows the standard procedure of the reduce task. The join algorithm introduces any changes through the reduce function, that is during the reduce phase, when disjoint records are being discarded. However, the reduce phase writes the results directly to HDFS and thus no I/O cost appears. Consequently, the reduce procedure of reduce side join is dictated by the same I/O cost formulas as the ones presented in Chapter 2 and Section 2.2. Of course, the size *mapOutputSize*, that represents the segment (in bytes) that each map task propagates to a reducer, is increased by the bytes of tagging applied during the map phase.

The aim of the reduce side join is that all the records that have the same join key are directed to the same reducer. However, since no filtering is done before the initiation of the reduce task, all the records of every source that takes part in the join, propagate to reducers and go through the map/reduce shuffle. Even disjoint records are fetched and consequently redundant transfers occur. This point reduces remarkably the efficiency of the simple reduce side join and renders the need for filtering imperative.

3.3 Reduce Side Join with Bloom Filter

An efficient method to mitigate the I/O cost related to the reduce side join is to minimize the redundant transfers, that is to allow only the join records to be fetched during the shuffle phase. Based on this remark, we propose a method to optimize the simple reduce side join by introducing the use of a Bloom filter on the map tasks. We expect that filtering will induce a general I/O cost reduction.

3.3.1 Description

What is a Bloom filter? It is a compact probabilistic data structure used to test whether an element is part of a set. The test may return true for elements that are not actually members of the set (false-positives), but will never return false for elements that are in the set; for each element in the set, the filter must return true. The false positive rate is the payoff for the minimal memory space required for the storage of the filter. In fact, the memory required is only a fraction of the one needed for storing the whole set.

A Bloom filter consists of two components: a bit vector of a given length of m bits and a set of k hash functions. Every hash function $h_1, h_2 \dots h_k$ returns values in the range $\{1 \dots m\}$, that is $h_i : X \rightarrow \{1 \dots m\}$. In our implementation X is the value of the key. If we want to construct a Bloom filter for a set $A = \{a_1, a_2 \dots a_n\}$ of n elements, we compute k hashes of each element and turn on (set to 1) those bits in the bit array. To check whether a value is present, we compute k hashes, and check all of those bits in the array to see if they are turned on. Only if all of those bits are set to 1 will the filter return true.

When constructing the filter, one should choose its desired point in a trade-off between the false positive rate and the size m of the filter. The bigger the bit vector, the smaller the probability that all k bits we check will be on, that is, smaller false positive rate. Of course, the size of the bit vector depends on the number of elements that are to added to the set. The greater the number of elements is, the more bits are required for a high-performing filter.

In our tests we assigned 8 bits to every inserted element, resulting in a Bloom filter of size

$\mathit{numOfElements} \cdot 8$ bits and a low false positive rate.

The use of a Bloom filter as a part of the reduce side join implementation reduces the number of records being fetched during the shuffle/copy phase. The idea behind this optimization, is the use of a Bloom filter in every map task to filter the records and discard the ones that do not contribute to the join results. The optimized reduce side join with the use of a bloom filter can be divided in two steps

1. construction of the bloom filter

2. use of the the bloom filter in the reduce side join

Each step amounts to a map/reduce job. During the first step, a Bloom filter is constructed based on the records of one of the participating datasets. In detail, the smallest of the two datasets is chosen and the filter is constructed having as elements the keys of the chosen dataset. The whole construction is conducted through a map/reduce job, where no reduce tasks are spawned. Only map tasks take place as illustrated in Figure 3.4.

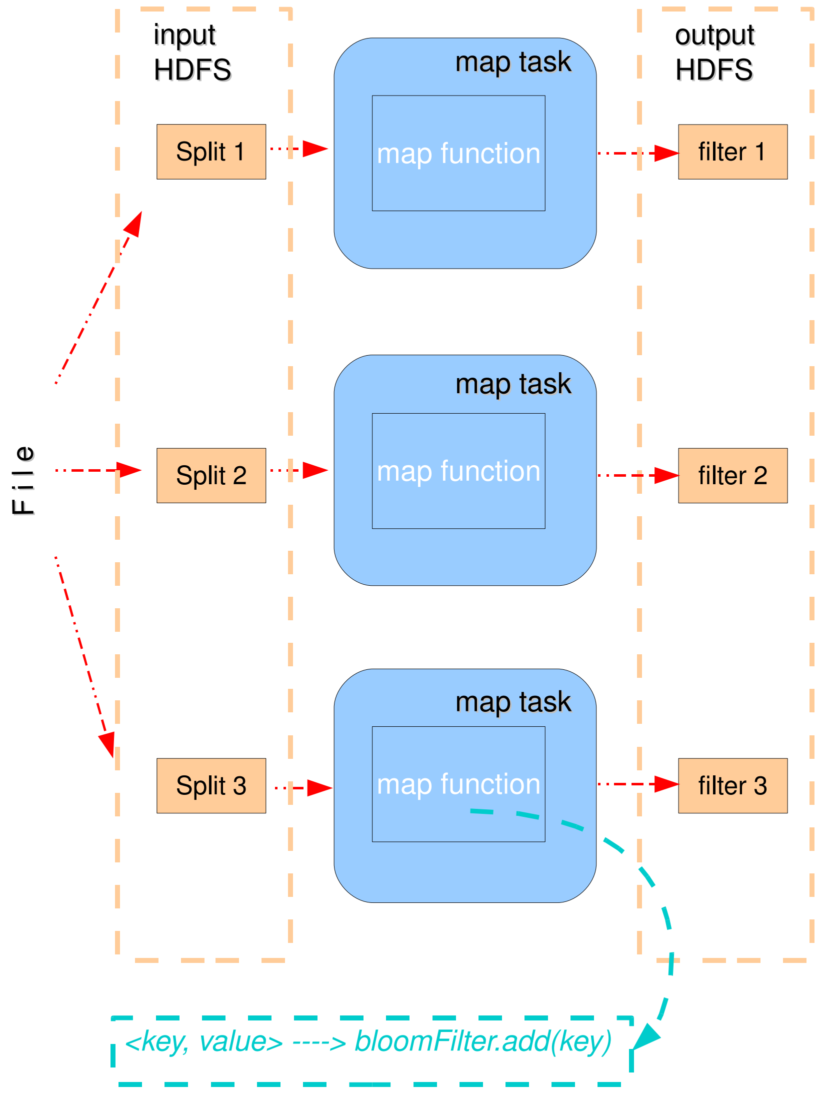

Figure 3.4: Bloom filter construction

Each map task reads the input split directly from the HDFS and constructs a Bloom filter; as the input split is read record by record, the *key* of each record is retrieved and used for the construction of the filter. Notice, that each map task consists of the map function only, that is, no buffering and writing take place. Consequently, there are no local bytes read or written and no I/O cost present. Depending on the size of the file, a number of map tasks may be initiated, leading to the creation of more than one filters. When each map task is completed, the corresponding filter is serialized and directly written to HDFS. The set of the filters will be used in the next step, that is, the optimized reduce side join.

In case of a file with duplicates, the same keys may be used for the construction of different filters. The construction of only one filter based on distinct values of the key would be ideal since during reduce side join, the memory space required for the filter in the map task would be less. This can be achieved using a map/reduce job with only one reduce task during which the construction of the filter takes place. Recall that the user can control the number of the reduce tasks. The same does not hold for the number of map tasks. The records enter the reduce function in groups based on their key. Consequently, the reduce function can retrieve only distinct values of the key and use them for the creation of the filter. However, this approach comes at a price; all the records of the input file have to go through the shuffle/copy phase and consequently I/O cost appears. A user can choose the approach that best fits his needs and job requirements. In our experiments, aiming at the least possible I/O cost, we chose the first approach where no I/O cost is present.

The second step is a map/reduce job during which each map task uses the created filters to filter the records and each reduce task evaluates the join. Details of each task are provided in the following section along with their I/O cost analysis.

3.3.2 I/O cost analysis

3.3.2.1 Map phase

The whole map procedure is the same as the one of the simple reduce side join. However, the map function does not only tag the input records but also filters them allowing only some of them to be part of the final map output. This is depicted in Figure 3.5.

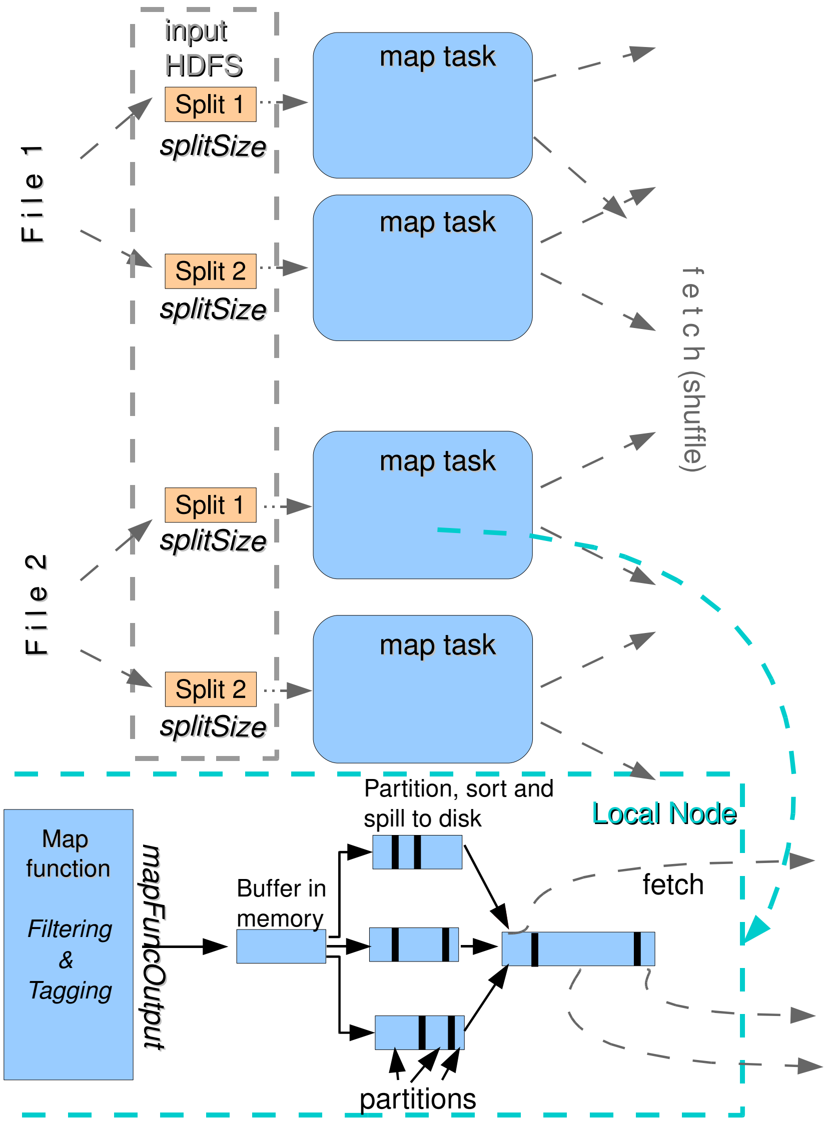

Figure 3.5: Map side of reduce side join with the use of Bloom filter

In detail, when the second jop (step) initiates, the framework copies the files (which contain the serialized filters) to the slave nodes before any tasks for the job are executed. This is achieved by a facility called `DistributedCache` which is provided by the map/reduce framework to cache files needed by the applications. Each map task recreates the filters from the copied files and stores them in memory. The key of each input record is used to query the filter/filters for membership. In case of membership, the input record is tagged with its source (in the same way as in the simple reduce side join) and passes to the buffering phase. Otherwise, it is discarded. Based on the example illustrated in Figure 3.5 and assuming that file 1 is used for the construction of the filters, the map function tags and passes all the records of file 1 but only a small number of the records of file 2. More precisely, all the records that join with records of file 1 pass. However, some dangling records may pass due to false positives.

As far as the I/O cost is concerned, it can be deduced refering to the formulas introduced in the simple reduce side join. Once again we introduce the following sizes

- *splitSize* → the size (in bytes) of the split that consists the input to every map task
- *numOfInputRecords* → the number of the records that the input split contains
- *mapFuncOutput* → the output of the map function
- *t* → the bytes that tagging adds to each record
- *bloomSel* → the selectivity of the constructed filter, that is, the percentage of the input records that pass the filter stage.

The selectivity of the filter is of high importance, since it determines the percentage of the initial input that will continue to the next I/O cost-bound phases. We define the selectivity of the filter with regard to the file that passes through the filter; it is defined as a percentage of the total records and is given as a a ratio of the total records that participate in the join results to the total records of the file.

$$bloomSel = \frac{numOfJoinRecords}{numOfInRecords}$$

Each record of the input split enters the map function. Once again, we assume that the user-defined map function does not apply any selection or projection to the records. However, tagging and filtering change the *splitSize* and result in a *mapFuncOutput* as follows

$$\begin{aligned} mapFuncOutput_{RSJ-f} &= bloomSel \cdot splitSize + (bloomSel \cdot numOfInputRecords) \cdot t \\ &= bloomSel \cdot mapFuncOutput_{RSJ} \end{aligned}$$

The cost of reading, buffering and writing phases of each map task is dictated by the same formulas presented in the simple reduce side join and fully analyzed in Chapter2 and Section 2.1. According to these formulas, the size that determines the I/O cost of the map phase is the output of the map function, that is the *mapFuncOutput*. Interesting enough is the comparison with the cost of the simple reduce side join.

The cost of the reading phase is zero. No trips to disk take place.

$$cost_r^{RSJ-f} = 0$$

The buffering is associated with the following cost

$$\begin{aligned} cost_{buf}^{RSJ-f} &= mapFuncOutput_{RSJ-f} \\ &= splitSize \cdot bloomSel + (bloomSel \cdot numOfInputRecords) \cdot t \\ &= mapFuncOutput_{RSJ} \cdot bloomSel \\ &= cost_{buf}^{RSJ} \cdot bloomSel \end{aligned}$$

while the writing phase yields a cost computed as follows

$$cost_{wr}^{RSJ-f} = r(0) \cdot cost_{wr}^{sp=1} + r(0) \cdot \left[r(1) \cdot cost_{wr}^{rnd1} + r(2) \cdot cost_{wr}^{intrm-rnds} + cost_{wr}^{final-rnd} \right] \Rightarrow$$

$$cost_{wr}^{RSJ-f} \leq cost_{wr}^{rnd1} + cost_{wr}^{intrm-rnds} + 2 \cdot mapFuncOutput_{RSJ-f}$$

Finally, the total I/O cost of the map phase during reduce side join with filter is given by the formula

$$\mathbf{cost}_{map}^{RSJ-f} = cost_{r}^{RSJ-f} + cost_{buf}^{RSJ-f} + cost_{wr}^{RSJ-f} \Rightarrow \tag{3.3}$$

$$\mathbf{cost}_{map}^{RSJ-f} = mapFuncOutput_{RSJ-f} + cost_{wr}^{RSJ-f} \Rightarrow \tag{3.4}$$

$$\mathbf{cost}_{map}^{RSJ-f} \leq 3 \cdot mapFuncOutput_{RSJ} \cdot bloomSel + cost_{wr}^{rnd1} + cost_{wr}^{intrm-rnds} \tag{3.5}$$

Based on the inequality (3.5), we can conclude that the selectivity of the Bloom filter can greatly affect the upper bound of the total I/O cost of the map phase. A comparison with the corresponding formula (3.2) of simple reduce side join underlines the fact that the cost of every map task is decreased under the use of filtering. The lower the selectivity, the greater the reduction. Consequently, for relations that yield low join ouput cardinality the gain can be substantial.

3.3.2.2 Reduce phase

The reducer retrieves the map outputs of assumingly comparable size, $mapOutputSize_{RSJ-f}$. The three phases, that is, copy/shuffle, sorting and reduce phase take place as they are already described in previous chapter. During the last phase, the user-defined reduce function undertakes the join of the records. The filtering ensures that the records which consists part of the join results (that is, they have the right key value) will successfully reach the reduce function. However, due to the false positive rate the filter is related to, a number of dangling records may also reach the reduce function. Consequently, we implemented our reduce function to apply a further selection on the records and discards any disjoint ones. The procedure is depicted in figure 3.6.

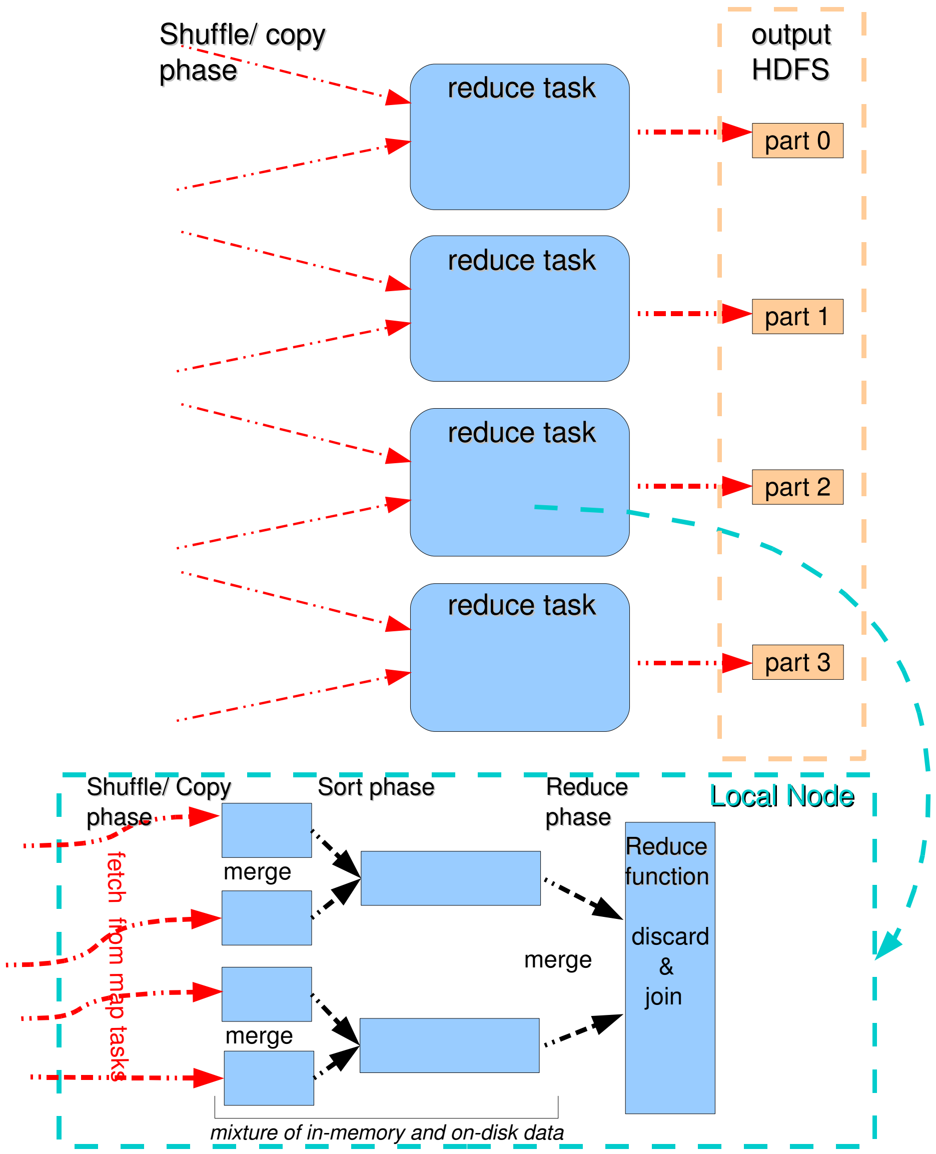

Figure 3.6: Reduce side of reduce side join with the use of Bloom filter

During the copy/shuffle phase, copier threads retrieve N map outputs of comparable size which we denote as $mapOutputSize_{RSJ-f}$. The I/O cost (as it is computed using the cost formulas of Chapter 2) is a function of the total input, that is, the N map outputs. The use of filtering on the map side reduces the size of the total input that is presented to the reduce side. Considering the $mapOutputSize_{RSJ}$, that is, the input that each map task contributes to the total input of a reduce task during the simple reduce side join, the corresponding input in case of the bloom filtering is decreased as follows

$$mapOutputSize_{RSJ-f} = mapOutputSize_{RSJ} \cdot bloomSel$$

The selectivity of the filter reduces the output of each map task resulting in an overall reduce in the input of the reduce task. Consequently, the cost formulas are applied on less records and less bytes are read and written locally. All in all, reduce side join with filtering on the map side is an optimization of the simple reduce side join and it is expected to generate less I/O cost in most cases.

3.4 Map Side Join

Joining datasets using the simple reduce side join seems to be the most straightforward approach, since it exploits the basic mechanism of map/reduce framework that sorts and partitions the data before it reaches the reduce function, where the actual join takes place. Partitioning ensures that records which have the same value of the join key result in the same reducer while sorting makes sure that records of the same dataset are grouped together. In this section, we analyse another join algorithm, the map side join, which, as its name implies, implements the join on the map side [10].

3.4.1 Description

A map-side join takes place when the data is joined before it reaches the map function. However, it lacks the generality of the reduce side join, since its efficiency requires the data inputs to be partitioned and sorted in the same way. In detail,

- A given key has to be in the same partition in each dataset so that all partitions that can hold a certain key are joined together. For this to work, all datasets should be partitioned using the same partitioner and moreover, the number of partitions in each dataset should be identical.
- The sort order of the data in each dataset must be identical. This requires that all datasets must be sorted using the same comparator.

The map side join code exploits the structure of the input datasets to perform the join. Assuming that each dataset is partitioned in the same number of partitions and using the same partitioner, records that have the same value of the join key will reside in the same partition in each dataset. Map side code builds multiple map tasks and each map task receives as its input the same partition from the input sources. In this fashion, it is ensured that each map task will consume all the records (from each dataset) that have the same value of the join key. For instance, if we have 2 input datasets consisting of 3 partitions each, three map tasks will initiate and each map task will retrieve two partitions, one from each dataset. To ensure that each map task retrieves the whole corresponding partition from each dataset, the framework sets the property `mapred.min.split.size` to *Long.MAX_VALUE*. Otherwise, the framework might split the partitions read from each dataset. Having the relevant partitions as its input and before calling the map function, each map task evaluates the join. The latter is conducted in-memory, yields no I/O cost and the results are presented to the map function. The above mentioned procedure is depicted in Figure 3.7.

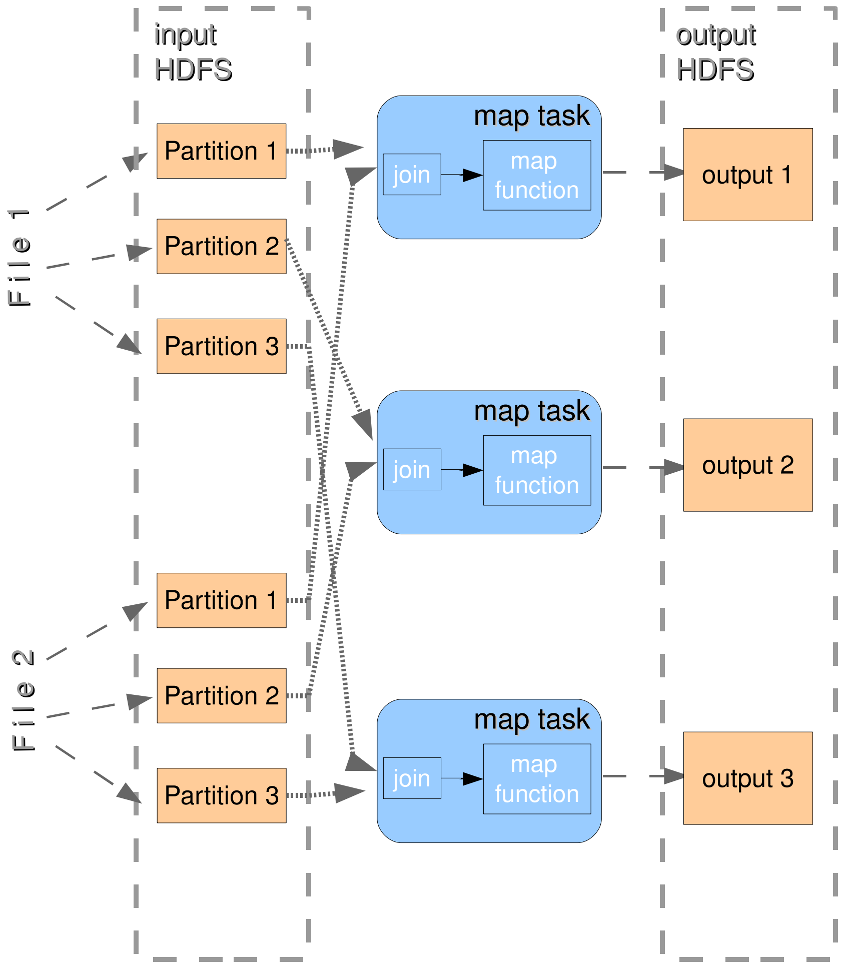

Figure 3.7: Map side join

3.4.2 I/O cost analysis

If the input datasets are partitioned and sorted in the fashion we described above, the join is conducted before the map function is called and the result can be directly written to HDFS using the map function. No reduce phase takes place. The latter, eliminates the I/O cost associated with the reduce task. Moreover, since the join results are directly written to the HDFS no buffering and writing occur during the map task. Consequently, no bytes are written or read locally. In detail,

$$cost^{MSJ}_{map} = 0 \qquad \text{and} \qquad cost^{MSJ}_{reduce} = 0$$

In this case, the total I/O cost induced by the map side join in terms of a job that involves a number of map tasks, is zero.

$$cost_{MSJ} = 0$$

However, if the input datasets do not conform to the above mentioned requirements, then preprocessing is needed. Preprocessing includes all the necessary procedures that the datasets should undergo to fit the requirements of the map side join; additional map/reduce jobs should take place that will sort and partition the datasets in the same way. Each job increases the total cost of the map side join by the size denoted by the basic formulas introduced in Chapter 2. Determining as *numSets* the number of input datasets to be joined, the additional cost imposed by preprocessing is determined as follows

$$cost^{MSJ}_{preproc} = \sum_{i=1}^{numSets} cost^{set-i}_{preproc}$$

Note that the $cost^{set-i}_{preproc}$ refers to the total I/O cost imposed by the Map/Reduce job that undertakes the preprocessing of each dataset. In other words, it involves a number of map and reduces tasks.

Eventually, the total cost of the map side join, examined as a whole procedure, is

$$\mathbf{cost}^{MSJ}_{total} = cost_{MSJ} + cost^{MSJ}_{preproc} = cost^{MSJ}_{preproc} = \sum_{i=1}^{numSets} cost^{set-i}_{preproc}$$

All in all, what makes map side join quite attractive is the potential for elimination of the reduce phase. Each map task reads the data directly from the HDFS and no reduce is needed. Consequently, there is a great reduction in cluster load. It is particularly ideal when its input is the output of another map/reduce job, since in this case it is ensured that the input conforms to the requirements and no preprocessing is needed. In this case map side join claims zero I/O cost and is the best choice. If this is not the case map side join requires preprocessing and reduce side join appears an efficient competitor. Moreover, during map side join the cluster might lose its ability to manage data locality for the map tasks; map/reduce tries to initiate the map tasks on the same node where the input data is stored. However, since map side join combines partitions of different files, it is likely that a map task will not initiate at the same node where the data resides.

3.5 Discussion

In the preceding sections we presented three join implementations and used our proposed cost model to theoretically evaluate them in terms of I/O cost. Our analysis provided some interesting remarks that can be outlined as follows.

- In the reduce side join, as suggested by the formula 3.2, there is a potential for I/O cost reduction by implementing a map function to reduce the size of data that pass to the following phases
- The redundant transfers, occurring during the shuffle phase of the reduce side join, are expected to increase the I/O cost induced on each reduce task
- The reduce side join with Bloom filter, exploiting the potential related to the map function, introduces filtering of the records during the function and is expected to yield I/O cost reduction.
- The map side join, undoubtedly, dominates over the two reduce side joins yielding zero cost. However, the preprocessing needed is expected to degrade its performance

In Chapter 4 we experimentally evaluate the above mentioned expectations.

At this point, we should mention that there are also other ways to conduct a join using the Map/Reduce framework. For instance, if one input is small enough to fit in memory, it can be replicated to every map and read sequentially during the pass of the second dataset. Alternatively, a hash map of it can be constructed. In this way, the join is conducted in the same way a hash join is. Moreover, the associated I/O cost is reduced, comparing to the basic implementations, since only one input is needed to follow the whole Map/Reduce dataflow. However, these approaches are only efficient for small datasets. In every other case one of the two basic join implementations is chosen.

Chapter 4

Experimental evaluation

In this chapter we present the results of experiments conducted using a map/reduce cluster. We carried out experiments to evaluate the validity of the general I/O cost formulas presented in Chapter 2. Moreover, we conducted additional experiments to compare the join algorithms introduced in Chapter 3 in terms of I/O cost.

We set up a Hadoop cluster consisting of 3 nodes. The Hadoop version we used is 0.20.0 running on Java 1.6.0. We deployed the system with the default configuration settings except where explicitly specified otherwise in the text. For each experiment, we stored all input and output data in the Hadoop distributed file system (HDFS). Two of the three nodes had a single 1.86 GHz Intel Core 2 Duo processor running 64-bit Ubutu Linux 9.04 (kernel version 2.6.28) with 2GB RAM and one 160GB SATA hard disk. The third node had a single 2.26GHz Pentium 4 processor running 64-bit Debian Linux 4 (kernel version 2.6.26) with 1.5GB RAM and one 300GB IDE hard disk. Hadoop uses a ”master” HDFS daemon (*NameNode*) to coordinate node activities while the rest of the nodes are the slave nodes (*DataNodes*). For our purposes, we deployed one node to implement the master and slave node at the same time, while two nodes were only slave nodes.

4.1 Cost Model Validation

The formulas presented in Chapter 2 estimate the I/O cost related to a single map and a single reduce task. Their extraction was based on the implementation details of every task. To experimentally verify our theoretical expectations, we deployed the cluster and conducted a number of experiments. This section presents the corresponding results. The real I/O cost of a map and reduce task (of a map/reduce job) is compared to the corresponding theoretical cost model.

4.1.1 Simple experiment

Our first experiment consisted of a simple map/reduce job during which 3 map and 3 reduce tasks were spawned. As input we used a file with a cardinality of $2,000,000$ records and a total size of $181,266,670$ bytes. The distribution of the key values was uniform and thus each reduce task received inputs of comparable size. In the same fashion as in our theoretical approach, during each map and reduce task, we used map and reduce functions that do not alter the size of the input, that is, we followed their identity implementation that propagates the records intact.

The input file was loaded on the HDFS filesystem and was split in 3 blocks, *block_A*, *block_B* and *block_C*. Following the replication policy of the standard map/reduce framework and having the configuration property `dfs.replication` set to 2, a total of 6 blocks were stored in the HDFS filesystem. Recall, that blocks are not replicated on each drive on a single datanote; to prevent data loss, block replication is across distinct datanodes. The map/reduce job initiated and the `Job Tracker`, executed on the *NameNode*, collocated the input data with the compute nodes; `node 1` carried out 2 map tasks and processed the blocks *block_B* and *block_C* while `node 2` carried out 1 map task and processed the block *block_A*. Each map task retrieved as its input the whole block, that is, no split of the blocks took place. The allocation of the reduce tasks was divided among the nodes and each node undertook a reduce task. The above is illustrated in Figure 4.1 .

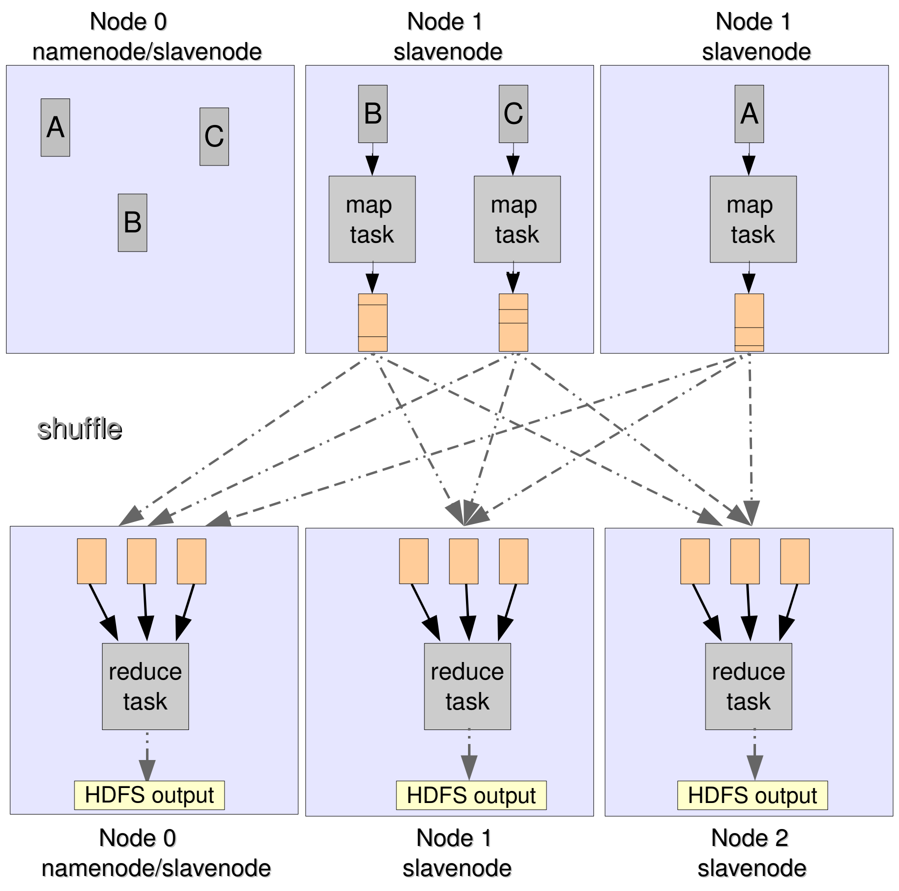

Figure 4.1: Task allocations of the first test

In the following sections, we first use our cost model to evaluate the I/O cost of a single map and a reduce task (Section 4.1.1.1) and compare the results with the real cost values provided by the framework (Section 4.1.1.2).

4.1.1.1 Model predictions

For the application of our cost model, the values of some configuration properties should be known. These are given in Table 4.1.

Map Task		**Reduce Task**	
Property	**Value**	**Property**	**Value**
`io.sort.mb`	$10^2 \cdot 2^{20}$	`mapred.job.shuffle.input.buffer.percent`	0.70
`io.sort.record.percent`	0.05	`mapred.job.shuffle.merge.percent`	0.66
`io.sort.spill.percent`	0.80	`max.single.shuffle.seg.fraction`	0.25
`io.sort.factor`	100	`io.sort.factor`	100

Table 4.1: Cluster's properties values during simple experiment

Map task Sine each map task is associated with the same cost analysis, we chose to present the results of one of the three map tasks. The metrics provided by the map/reduce framework referring to the map task conducted on the slavenode 1 are depicted in Table 4.2.

Map-Reduce Framework	
Map input records	738,727
Map output bytes	67,108,925
Map input bytes	67,108,925
Map output records	738,727

Table 4.2: Simple exp.-map task metrics

The `Map input bytes` row refers to the split size that was the input to this node, that is, the size of `block_B`. As expected, this size is approximately one third of the total size ($181,266,670$ bytes) of the input file. The `Map output bytes` is the size of the map function output. This value is the same as the input bytes, since the map function passes the input records intact.

Before applying our cost formula, we define

- *splitSize* = 67,108,925 bytes $\rightarrow$ the input split size of the map task
- *inRecs* = 738,727 records $\rightarrow$ the number of input records

Taking into account the properties values in Table 4.1, we present the results of our model analytically for each one of the three phases of the map task: reading, buffering and writing phases. The cost of the **reading phase** is

$$\mathbf{cost}^{r}_{map} = 0$$

The buffering phase is related to the estimation of several intermediate sizes needed for the further I/O cost evaluation. These sizes are given as follows.

The size of the accounting buffer is

$$buffer_{mtdt} = (\texttt{io.sort.mb}) \cdot (\texttt{io.sort.record.percent}) = 5,242,880 \text{ bytes}$$

The maximum number of records (metadata records) the accounting buffer can store is

$$maxRecs_{mtdt} = \left\lfloor \frac{buffer_{mtdt}}{16} \right\rfloor = 327,680 \text{ records}$$

Spill threshold of the accounting buffer

$$spillThr_{mtdt} = (maxRecs_{mtdt}) \cdot (\texttt{io.sort.spill.percent}) = 262,144 \text{ records}$$

For the input split of *inRecs*, the number of the *spill* files is

$$numSpills = \left\lceil \frac{inputRecs}{spillThr_{mtdt}} \right\rceil = 3$$

Each spill has a size as given below

$$spillSize = \frac{splitSize}{numSpills} \approx 2,2369,642 \text{ bytes}$$

The I/O cost of the **buffering phase** equals the cost of writing the whole input split, divided in spills, on disk that is,

$$\mathbf{cost}_{map}^{buf} = splitSize = 67,108,925 \text{ bytes}$$

Since the number of spills is greater than one, the number of merges that take place during the writing phase is

$$numMerges = \frac{numSpills - 1}{\texttt{io.sort.factor} - 1} + 1 \cdot \left\lceil \frac{\|1 - numSpills\|}{numSpills} \right\rceil = 1$$

Referring once again to our formulas, since the number of merges is equal to one, the cost of the **writing phase** is

$$\begin{aligned}\mathbf{cost}_{map}^{wr} &= (1 - r(0)) \cdot cost_{wr}^{sp=1} + r(0) \cdot \left[r(1) \cdot cost_{wr}^{rnd1} + r(2) \cdot cost_{wr}^{intrm-rnds} + cost_{wr}^{final-rnd} \right] \\ &= 2 \cdot splitSize \\ &= 134,217,850 \text{ bytes}\end{aligned}$$

Finally, the **total cost** of the map task is

$$\mathbf{cost}_{map} = \mathbf{cost}_{map}^{r} + \mathbf{cost}_{map}^{buf} + \mathbf{cost}_{map}^{wr} = 201,326,775 \text{ bytes}$$

The total cost refers to the total bytes read and written from/to the local disk during the execution of the task.

Reduce task As shown in Figure 4.1, during the reduce phase 3 reduce tasks were spawned and each one was executed on a different node. In the same fashion as in the map phase, we will predict the cost of one reduce task. The metrics of the reduce task of node 0 are given in Table 4.3

Map-Reduce Framework	
Reduce input records	666,667
Reduce shuffle bytes	61,756,077
Reduce output records	666,667

Table 4.3: Simple exp.-reduce task metrics

During the experiment, every task (map or reduce) is provided with a prespecified memory of size 200 MB. In our theoretical cost model, this size corresponds to the variable *availMem*. The reduce task receives 3 map outputs of total $61,756,077$ bytes as depicted in the `Reduce shuffle bytes` row. Since the distribution of the key values is uniform, the size of each map output is given as follows

$$mapOutputSize = \frac{totalInput}{3} = 20,585,359 \text{ bytes}$$

Following the same analysis of the cost results presented in the map phase, we will estimate the I/O cost of each phase of the reduce task; the shuffle, sort and reduce phases using our formulas. To achieve this, the estimation of additional sizes will precede.

The memory provided for the purposes of the shuffle phase is

$$maxMem_{shuf} = (\texttt{mapred.job.shuffle.input.buffer.percent}) \cdot availMem = 140,000,000 \text{ bytes}$$

Note that the available memory *availMem* might be a little less than 200 MB during runtime. The threshold that determines whether a map output will be written in memory or on disk during the shuffle phase is

$$maxMapSize = maxMem_{shuf} \cdot \texttt{max\_single\_shuffle\_seg\_fraction} \quad = \quad 35,000,000 \text{ bytes}$$

Since the size of each map output is approximately $mapOutputSize = 20,585,359$ bytes, all 3 map outputs are written in memory. The memory threshold that triggers a merge is given as follows

$$inMemThr = \texttt{mapred.job.shuffle.merge.percent} \cdot maxMem_{shuf} \Rightarrow \ = 92,400,000 \text{ bytes}$$

The number of map outputs that can be written in memory before the merge thread is triggered and spills them to disk is

$$numOutputs_{merged}^{in-mem} = \left\lceil \frac{inMemThr}{mapOutputSize} \right\rceil = 5$$

Since the input consists only of 3 map outputs no merge thread is triggered. Consequently

$$mergeItrs_{in-mem} = \frac{3}{numOutputs_{merged}^{in-mem}} = 0 \qquad \text{and} \qquad costMerge_{in-mem} = 0$$

The absence of in-memory merge yields no *spill* files to disk and consequently no disk merge takes place.

$$mergeItrs_{onDsk} = 0 \qquad \text{and} \qquad costMerge_{onDsk} = 0$$

The total I/O cost of the **shuffle/copy phase** is

$$\mathbf{cost}^{shuf}_{reduce} = costMerge_{in-mem} + costMerge_{onDsk} = 0$$

The number of unmerged outputs lying in memory and on disk is

$$outputs^{in-Mem}_{shuf} = 3 \qquad \text{and} \qquad files^{onDsk}_{shuf} = 0$$

During the sort phase, there are only 3 files in-memory. Since the number of files on disk is zero and the `io.sort.factor` is equal to 100, the boolean function

$$spillToDsk(files^{onDsk}_{shuf}) = \begin{cases} 1 & \texttt{io.sort.factor} > files^{onDsk}_{shuf} \\ 0 & otherwise \end{cases}$$

has a value of $spillToDsk(0) = 1$ determining that the in-memory files are spilled to disk and merged into one file. Consequently the corresponding cost formula yields,

$$\begin{aligned} cost^{spill}_{srt} &= outputs^{in-Mem}_{shuf} \cdot mapOutputSize + N \cdot mapOutputSize = 2 \cdot totalInput \\ &= 123{,}512{,}154 \text{ bytes} \end{aligned}$$

The same result can be retrieved using the final cost formula of the sort phase

$$\begin{aligned} \mathbf{cost}^{srt}_{reduce} &= \Big(1 - spillToDsk(files^{onDsk}_{shuf})\Big) \cdot (r(1) \cdot cost^{rnd1}_{srt} + r(2) \cdot cost^{intrm-rnds}_{srt} + cost^{final-rnd}_{srt}) \\ &\quad + spillToDsk(files^{onDsk}_{shuf}) \cdot cost^{spill}_{srt} \\ &= \mathbf{cost}^{spill}_{srt} \end{aligned}$$

The cost of the **reduce phase**, which is the last phase of the task, is

$$\mathbf{cost}^{rdc}_{reduce} = 0$$

Finally, the **total cost** of the reduce task is

$$\mathbf{cost}_{reduce} = \mathbf{cost}^{shuf}_{reduce} + \mathbf{cost}^{srt}_{reduce} + \mathbf{cost}^{rdc}_{reduce} = 123{,}512{,}154 \text{ bytes}$$

The total cost refers to the total number of bytes read and written from/to the local disk during the execution of the task.

4.1.1.2 Comparison with real results

We draw interesting conclusions by comparing the predicted values with the real ones provided by the framework. To enable the comparison, Tables 4.4 and 4.5 are given referring to the map and reduce tasks correspondingly.

map task theoretical cost		
Phases	**Local Bytes Read**	**Local Bytes Written**
reading	0	0
buffering	0	67,108,925
writing	67,108,925	67,108,925
Total cost	67,108,925	134,217,850

map task real cost		
	Local Bytes Read	**Local Bytes Written**
Total cost	68,600,608	137,172,910

Table 4.4: Simple experiment-predicted and real cost of the map task

Observing the values in Table 4.4, we conclude that our formulas provided a good approximation of the real map I/O cost. The total amount of bytes read and written are only slightly under predicted compared to the real values. This can be attributed to several reasons; during the creation of the intermediate files (as the *spills* are) the map/reduce framework adds checksum bytes that increase the final read and written bytes. Furthermore, bytes are added due to headers attached to each spill file needed for the shuffle phase. Moreover, the rounding that took place in the calculation affects the final result. However, this error is only minor compared to the total cost and does not deprive us from a good approximation.

reduce task theoretical cost		
Phases	**Local Bytes Read**	**Local Bytes Written**
shuffle	0	0
sort	61,756,077	61,756,077
reduce	0	0
Total cost	61,756,077	61,756,077

reduce task real cost		
	Local Bytes Read	**Local Bytes Written**
Total cost	61,756,065	61,756,065

Table 4.5: Simple experiment- predicted and real cost of the reduce task

Regarding the reduce task, the predictions obtained by our formulas greatly approach the real I/O cost of the reduce task. There are only slight differences which can be attributed to the same reasons as the ones provided in the analysis of the map task. The good approximation underlined the potential of our formulas. However, the simplicity of the experiment, mainly because of the small size of the input, led to the simplification of our formulas; neither in the map or in the reduce task did multiple merge rounds take place. For this reason, we conducted one further experiment.

4.1.2 Challenging experiment

The test job presented in the previous section proved that our cost formulas can give a result that adequately approximates the real one. However, the default configurations of the corresponding map/reduce job simplified the application of the formulas. By having a relatively small input file of $181,266,670$ bytes and keeping the default configuration properties of the map/reduce framework, each task was provided with an efficienct amount of memory. As a result, the number of merge iterations during the buffering phase of the map task and during the shuffle and sort phases of the reduce task were limited to one or zero invoking a simplified version of our formulas. Nonetheless, in this section we deal with the results of a more demanding test that challenges further the validity of our cost formulas.

For the purposes of this experiment, we loaded a file of $50,000,000$ records and $4,731,666,670$ bytes in size. To limit the available memory during the execution of the phases of each task we tweaked the configuration parameters presented in Table 4.6.

Map Task	
Property	**Value**
`io.sort.mb`	$10 \cdot 2^{20}$
`io.sort.record.percent`	0.05
`io.sort.spill.percent`	0.80
`io.sort.factor`	10

Reduce Task	
Property	**Value**
`mapred.job.shuffle.input.buffer.percent`	0.017
`mapred.job.shuffle.merge.percent`	0.66
`max.single.shuffle.seg.fraction`	0.25
`io.sort.factor`	10

Table 4.6: Cluster's properties values during challenging experiment

Every other relevant property was left intact at its default value. The execution of the job on the cluster spawned 71 map tasks and 97 reduce tasks. The size of the input file was in the order of gigabytes. Recall also that our testbed was a cluster of three nodes. Both contributed to a lower absolute performance. They also contributed, however, to an increased complexity of each map/reduce task and in turn made the estimation of the I/O cost through our formulas more challenging. As in the first test we used a uniform key distribution while both map and reduce functions were the identity functions. Following the same analysis as in the simple experiment, we present (Section 4.1.2.1) and compare (Section 4.1.2.2) the results of a single map and a single reduce task.

4.1.2.1 Model predictions

Map task We chose one of the 71 spawned map tasks and used our cost model to predict the related I/O cost. The Hadoop framework provided us with the following metrics (Table 4.7) regarding the input and output bytes of the task. Based on these, our model is challenged to predict the I/O cost that this task induced on the node.

Map-Reduce Framework	
Map input records	715,017
Map output bytes	67,108,929
Map input bytes	67,108,929
Map output records	715,017

Table 4.7: Challenging exp.-map task metrics

Based on Table 4.7, we define

- *splitSize* = 67,108,929 bytes $\longrightarrow$ the input split size of the map task
- *inRecs* = 715,017 records $\longrightarrow$ the number of input records

Using the cost model presented in Chapter 2, we will estimate the I/O cost of each phase of this map task; reading, buffering and writing.

The cost of the **reading phase** is

$$\mathbf{cost}^{r}_{map} = 0 \text{ bytes}$$

Intermediate sizes should be estimated for the evaluation of the buffering phase. Based on the corresponding configuration properties provided in Table 4.6, these sizes are as follows.

The size of the accounting buffer provided for the storage of metadata is

$$buffer_{mtdt} = (\texttt{io.sort.mb}) \cdot (\texttt{io.sort.record.percent}) = 524,288 \text{ bytes}$$

The maximum number of records (metadata records) the accounting buffer can store is

$$maxRecs_{mtdt} = \left\lfloor \frac{buffer_{mtdt}}{16} \right\rfloor = 32,768 \text{ records}$$

The `io.sort.record.percentage` that determines the size $buffer_{mtdt}$ of the metadata buffer, is chosen low enough (0.05 by default) and consequently we can assume that the spill thread is triggered only due to reaching the threshold of the accounting buffer, that is, the threshold of the serialization buffer is not taken into account. The spill threshold of the accounting buffer is

$$spillThr_{mtdt} = (maxRecs_{mtdt}) \cdot (\texttt{io.sort.spill.percent}) = 26,214.4 \text{ records}$$

For the input split of *inRecs*, the number of *spill* files is

$$numSpills = \left\lceil \frac{inputRecs}{spillThr_{mtdt}} \right\rceil = 28$$

Each spill has a size as given below

$$spillSize = \frac{splitSize}{numSpills} \approx 2,396,747 \text{ bytes}$$

The I/O cost of the **buffering phase** equals to the cost of **writing** the whole input split, divided in spills, on disk that is,

$$\mathbf{cost}_{map}^{buf} = splitSize = 67,108,929 \text{ bytes (only writing)}$$

Since the number of spills (that is, $numSpills = 28$) is greater than one, the number of merges that take place during the writing phase is

$$numMerges = \frac{numSpills - 1}{\texttt{io.sort.factor} - 1} + 1 \cdot \left\lceil \frac{\|1 - numSpills\|}{numSpills} \right\rceil = 4$$

Since the number of merges is greater than one, that is, 4 merge rounds, we should find the cost of each merge round. In detail, there is the first round, two intermediate rounds and the final round. The first round merges only a certain number out of the total 28 *spill* files. After estimating all the necessary intermediate sizes needed, we have the following results.

The number of *spill* files merged during the first round is

$$\begin{aligned} numSpills_{merge}^{rnd1} &= functMod(modVar) \cdot (modVar + 1) \\ &\quad + (1 - functMod(modVar)) \cdot \texttt{io.sort.factor} \\ &= 10 \end{aligned}$$

The I/O cost of this round equals the cost of reading the $numOfSpillsMerged_{rnd1}$ spills and writing them in a single sorted file to disk. The intermediate rounds, are associated with the cost of merging `io.sort.factor` files each while the final round reads and writes the whole input split. Consequently, the cost of the first round includes reading and writing 10 spills.

$$cost_{wr}^{rnd1} = 2 \cdot numSpills_{merge}^{rnd1} \cdot spillSize = 4,793,494 \text{ bytes (reading and writing)}$$

while the cost of intermediate rounds is

$$\begin{aligned} cost_{wr}^{intrm-rnds} &= (numOfMerges - 2) \cdot 2 \cdot \texttt{io.sort.factor} \cdot spillSize \\ &= 95,869,880 \text{ bytes (reading and writing)} \end{aligned}$$

and the cost of the final round is

$$cost_{wr}^{final-rnd} = 2 \cdot splitSize = 134,217,858 \text{ bytes (reading and writing)}$$

The total cost of the **writing phase** is

$$\begin{aligned} \mathbf{cost}_{map}^{wr} &= (1 - r(0)) \cdot cost_{wr}^{sp=1} + r(0) \cdot \left[r(1) \cdot cost_{wr}^{rnd1} + r(2) \cdot cost_{wr}^{intrm-rnds} + cost_{wr}^{final-rnd} \right] \\ &= cost_{wr}^{rnd1} + cost_{wr}^{intrm-rnds} + cost_{wr}^{final-rnd} \\ &= 246,960,852 \text{ bytes} \end{aligned}$$

Finally, the **total cost** of the map task is

$$\mathbf{cost}_{map} = \mathbf{cost}^{r}_{map} + \mathbf{cost}^{buf}_{map} + \mathbf{cost}^{wr}_{map} = 314,069,781 \text{ bytes}$$

The total cost refers to the total bytes read and written from/to the local disk during the execution of the task.

Reduce task During the reduce phase 97 reduce tasks were spawned distributed among the three nodes of the cluster. In the same fashion as in the map phase, we will analyse the results of one reduce task. The metrics of the task are depicted in Table 4.8.

Map-Reduce Framework	
Reduce input records	515,454
Reduce shuffle bytes	49,447,944
Reduce output records	515,454

Table 4.8: Challenging exp.-reduce task metrics

We aim at finding a good approximation of the I/O cost related with the reduce task. The reduce task in question, retrieves 71 map outputs, one from each map task, resulting in a total input of 49,447,944 bytes, as depicted in the `Reduce shuffle bytes` row of Table 4.8. We will estimate the I/O cost of each phase of the reduce task; the shuffle, sort and reduce phases using our formulas. To achieve this, once again, the estimation of additional sizes will precede.

Same as in the first experiment, the prespecified memory for each task is 200 MB. However, during runtime not all of this memory is available. We define

$$availMem = 200,000,000 \cdot 0.97 = 186,000,000 \text{ bytes}$$

Since the distribution of the key values is uniform, the size of each map output is given as follows

$$mapOutputSize = \frac{totalInput}{71} \approx 696,450 \text{ bytes}$$

The memory provided for the purposes of the shuffle phase is

$$\begin{aligned} maxMem_{shuf} &= (\texttt{mapred.job.shuffle.input.buffer.percent}) \cdot availMem \\ &= 3,162,000 \text{ bytes} \end{aligned}$$

The threshold that determines whether a map output will be written in memory or to disk during the shuffle phase is

$$maxMapSize = maxMem_{shuf} \cdot \texttt{max\_single\_shuffle\_seg\_fraction} = 790,500 \text{ bytes}$$

Since the size of each map output is approximately $mapOutputSize = 696,450\ bytes$, all 71 map outputs are written in memory. The memory threshold that triggers a merge is given as follows

$$inMemThr = \texttt{mapred.job.shuffle.merge.percent} \cdot maxMem_{shuf} = 2,086,920 \text{ bytes}$$

The number of map outputs that can be written in memory before the merge thread is triggered and spills them to disk is

$$numOutputs^{in-mem}_{merged} = \left\lceil \frac{inMemThr}{mapOutputSize} \right\rceil = 3$$

Since the input consists of 71 map outputs the in memory merge procedure is repeated as many times as the following formula denotes

$$mergeItrs_{in-mem} = \frac{71}{numOutputs^{in-mem}_{merged}} = 23$$

and results in the creation of a file of the following size

$$fileSize^{in-mem}_{merged} = numOutputs^{in-mem}_{merged} \cdot mapOutputSize = 3 \cdot 696,450 \text{ bytes}$$

Moreover, disk merge takes place with a number of iterations determined as follows

$$mergeItrs_{onDsk} = \frac{mergeItrs_{in-mem} - 1}{\texttt{io.sort.factor} - 1} - 1 = 1$$

The in-memory merge iterations are related to the cost of writing the merged files to disk

$$\begin{aligned} costMerge_{in-mem} &= mergeItrs_{in-mem} \cdot numOutputs^{in-mem}_{merged} \cdot mapOutputSize \\ &= (69 \cdot 696,450) \text{ bytes (writing to disk)} \end{aligned}$$

On the other hand, every on-disk merge iteration merges `io.sort.factor` files resulting in a new file of size

$$fileSize^{onDsk}_{merged} = \texttt{io.sort.factor} \cdot fileSize^{in-mem}_{merged} = (30 \cdot 696,450) \text{ bytes}$$

and the cost of each merge iteration on disk is

$$\begin{aligned} costMerge_{onDsk} &= 2 \cdot mergeItrs_{onDsk} \cdot \texttt{io.sort.factor} \cdot fileSize^{in-mem}_{merged} \\ &= 2 \cdot (30 \cdot 696,450) \text{ bytes (reading and writing)} \end{aligned}$$

The total I/O cost of the **shuffle/copy phase** is

$$\mathbf{cost}^{shuf}_{reduce} = costMerge_{in-mem} + costMerge_{onDsk} = (30 \cdot 696,450\ read.) + (99 \cdot 696,450\ writing) \text{ bytes}$$

By the end of the shuffle phase, the files left in memory are

$$outputs^{in-Mem}_{shuf} = N - mergeItrs_{in-mem} \cdot numOutputs^{in-mem}_{merged} = 2$$

and on disk

$$files_{shuf}^{onDsk} = mergeItrs_{onDsk} + [mergeItrs_{in-mem} - (mergeItrs_{onDsk} \cdot \texttt{io.sort.factor})] = 14$$

For the sort phase we have $spillToDsk(files_{shuf}^{onDsk}) \equiv 0$, consequently

$$files_{merged}^{srt} = files_{shuf}^{onDsk} + outputs_{shuf}^{in-Mem} = 16$$

The number of rounds of the merge process is given by,

$$mergeRnds_{srt} = \frac{files_{merged}^{srt} - 1}{\texttt{io.sort.factor} - 1} + 1 = 2$$

The cost of the first round is

$$\begin{aligned} cost_{srt}^{rnd1} &= 2 \cdot [((files_{shuf}^{onDsk} - 1)\%(\texttt{io.sort.factor} - 1)) + 1] \cdot numOutputs_{merged}^{in-mem} \cdot mapOutputSize \\ &\quad + 2 \cdot outputs_{shuf}^{in-Mem} \cdot mapOutputSize \\ &= 2 \cdot (17 \cdot 696,450) \text{ bytes (reading and writing)} \end{aligned}$$

while the final (second) round is associated with as follows

$$cost_{srt}^{final-rnd} = totalInputSize = 49,447,944 \text{ bytes (reading)}$$

The total cost of the **sort phase** is

$$\begin{aligned} \mathbf{cost}_{reduce}^{srt} &= \left(1 - spillToDsk(files_{shuf}^{onDsk})\right) \cdot (r(1) \cdot cost_{srt}^{rnd1} + r(2) \cdot cost_{srt}^{intrm-rnds} + cost_{srt}^{final-rnd}) \\ &\quad + spillToDsk(files_{shuf}^{onDsk}) \cdot cost_{srt}^{spill} \\ &= cost_{srt}^{rnd1} + cost_{srt}^{final-rnd} \\ &= ((17 \cdot 696,450) + 49,447,944 \; reading) + (17 \cdot 696,450 \; writing) \text{ bytes} \end{aligned}$$

The cost of the **reduce phase** is

$$\mathbf{cost}_{reduce}^{rdc} = 0$$

Finally, the **total cost** of the reduce task is

$$\begin{aligned} \mathbf{cost}_{reduce} &= \mathbf{cost}_{reduce}^{shuf} + \mathbf{cost}_{reduce}^{srt} + \mathbf{cost}_{reduce}^{rdc} \\ &= (47 \cdot 696,450 + 49,447,944 \; reading) + (116 \cdot 696,450 \text{ writing}) \text{ bytes} \end{aligned}$$

4.1.2.2 Comparison with real results

Once again, our predicted values are compared to the real ones. Starting with the map task we have

map task theoretical cost		
Phases	**Local Bytes Read**	**Local Bytes Written**
reading	0	0
buffering	0	67,108,929
writing	117,440,616	117,440,616
Total cost	117,440,616	184,549,545
map task real cost		
	Local Bytes Read	**Local Bytes Written**
Total cost	121,632,320	184,520,604

Table 4.9: Challenging experiment-predicted and real cost of the map task

Comparing our results to the real ones we conclude that our formulas successfully coped with the challenging test and once again provided an accurate approximation of the real I/O cost. Slight difference in bytes is attributed to the reasons stated in the simple experiment (Section 4.1.1.2). However, comparing to the corresponding results of the simple experiment presented in Table 4.4, one may notice that in the challenging experiment the model gave more accurate predictions for the bytes written during the map task. Having conducted further experiments, we concluded that this behavior is mainly arbitrary; due to the introduction of more complicated formulas, more roundings took place producing results that were efficient to cover the difference introduced by the checksum bytes which were the reason of the under predicted values in the simple experiment.

Regarding the reduce task, the results are gathered in Table 4.10

reduce task theoretical cost		
Phases	**Local Bytes Read**	**Local Bytes Written**
shuffle	20,893,500	68,948,550
sort	61,287,594	11,839,650
reduce	0	0
Total cost	82,181,094	80,788,200
reduce task real cost		
	Local Bytes Read	**Local Bytes Written**
Total cost	81,513,117	81,513,117

Table 4.10: Challenging experiment-predicted and real cost of the reduce task

Comparing our theoretical results to the real ones given by the framework, we notice that our formulas produced a good approximation of the cost. Referring to the simple test, the predictions of this complicated test have a greater deviation than the ones in the simple experiment. This is mainly attributed to the mechanisms that the map/reduce framework deploys during runtime. For example, the in-memory merge iterations that actually took place during the shuffle phase were more than the predicted number of 23. Due to the high number of the received map outputs (71), the framework triggers the merge thread before the in-memory merge threshold is reached to avoid stagnation. As a result, a greater number of bytes are written to disk. This behavior is not deterministic since it depends on the runtime conditions and consequently it cannot be encompassed in our formulas. However, our cost model offered a representative size of the I/O cost.

4.1.3 Conclusion

Both experimental results indicated that our theoretical cost model is able to capture the I/O cost associated with a map and reduce task with a good approximation. Any deviations from the real cost are attributed to minor overhead assigned for file creation purposes and to the runtime behavior of the cluster. Important enough is the assumption we made in both our tests; we assumed that the input file has a uniform distribution of the key values and moreover that the following inequality holds

$$(\texttt{io.sort.factor})^2 \geq (\textit{number of map outputs})$$

The first ensures that all the map outputs received by a reducer are of comparable size while the second guarantees that during merge iterations only unmerged files are being merged. (Refer to Chapter 2 and Section 2.3).

4.2 Join Algorithms

After having evaluated the validity of our cost model, we conducted experiments to evaluate the I/O cost related to each one of the three join algorithms presented in Chapter 3. For each join algorithm we used the cost model to predict the I/O cost of a single map and a single reduce task and compared the predicted values with the real ones resulted from the experiments (Section 4.2.1). Additionally, we conducted further experiments and used our model to predict and compare the total I/O cost of each join algorithm (Section 4.2.2).

4.2.1 Map/Task cost prediction of each join

For the purposes of this section we conducted a series of experiments during which we applied the three join algorithms over the same set of input files. First, we use our cost model to predict the cost of a map and reduce task that took place during each join job and present the results. Finally, we compare our predictions with the real experimental results.

In the experiment two files where used of different sizes but with approximately uniform distribution of the key values. Moreover, to ensure that we could be able to compare the join algorithms, the same number of mappers and reducers were spawned during the application of each algorithm. In a nutshell, three map/reduce jobs took place, one for each algorithm, having the following characteristics

- *file0* has a size of $181,266,670$ bytes and a cardinality of $2,000,000$ records
- *file1* has a size of $458,166,670$ bytes and a cardinality of $5,000,000$ records
- both files have the join key as primary
- $mappers = 10$
- $reducers = 4$

In the following sections. we provide the prediction of our theoretical cost model, regarding a single map and a single reduce task, for each one of the three algorithms.

4.2.1.1 Reduce side join

Each map task reads the input split record by record and the map function passes each record to the buffering phase after having tagged it. The tag overhead is equal to 4 bytes, that is 2 bytes for each tag on the key, value pair. If the map input split consists of X records and the size of each record is b_X, the output of the map function is $(b_X + 4) \cdot X$, that is $4X$ bytes overhead. Consequently, the actual input, determined as *splitSize* in our cost formulas, is increased as follows

$$mapFuncOutput_{RSJ} = splitSize + numOfInputRecords \cdot 4$$

For the application of our cost model we chose a map task that had as its input $splitSize = 67,108,881$ bytes and the incoming records were $numOfInputRecords = 745,309$. The reduce task chosen had as its input $mapOutput = 170,360,091$. Based on the above and after applying the corresponding formulas, our cost model provided the following results

Map Task		
Phases	**Read**	**Written**
reading	0	0
buffering	0	70,090,117
writing	70,090,117	70,090,117
Total cost	70,090,117	140,180,234

Reduce Task		
Phases	**Read**	**Written**
shuffle	0	102,216,054
sort	170,360,091	68,144,036
reduce	0	0
Total cost	170,360,091	170,360,090

Table 4.11: Reduce side join - cost model prediction

Map Task		
	Read	**Written**
Total cost	71,597,558	143,161,670

Reduce Task		
	Read	**Written**
Total cost	170,360,049	170,360,049

Table 4.12: Reduce side join - real cost

4.2.1.2 Reduce side join with Bloom filter

This test consisted of two map/reduce jobs. As we have thoroughly presented in Chapter 3, the reduce side join with Bloom filter algorithm has two steps. During the first step, the Bloom filter is constructed using *file0*. We chose to create the filter using the smallest dataset to limit the size of the constructed filter. This job yielded no I/O cost. The constructed filter was copied to every node and was available to every map task during the second job where the actual join took place. Each map task reads the input split record by record and the map function passes each record to the buffering phase after having filtered and tagged it. The use of a filter reduces the number of records that are tagged and passed to the following phase. This reduction varies among the map tasks; the map tasks that process splits of the *file0* pass all the records to the output while the map taps that process splits of the *file1* discard the ones that fail to pass the filtering, that is, the ones that do not contribute to the join result. Taking into account that both files have the join key as their primary and that different tasks process different file splits, we define the selectivity of the filter as follows

$$bloomSel(m) = \begin{cases} 1 & \text{for} \quad \textit{map tasks that process splits of file0} \\ \frac{2}{5} & \text{for} \quad \textit{map tasks that process splits of file1} \end{cases},$$

The input, denoted as *splitSize* in our cost formulas, is now determined as follows

$$\begin{aligned} mapFuncOutput_{RSJ-f} &= bloomSel \cdot splitSize + (bloomSel \cdot numOfInputRecords) \cdot 4 \\ &= bloomSel \cdot mapFuncOutput_{RSJ} \end{aligned}$$

To estimate the theoretical cost of a map task, we chose a task that processes splits of *file1* so as to underline the reduction in the records yielded by the filter as opposed to the tasks of *file0* that pass all the incoming records. The map task had as its input $splitSize = 67,108,863$ bytes of *file1* and the incoming records were $numOfInputRecords = 730,583$. Regarding the reduce phase of the job, we chose a reduce task among the 4 that were spawned. The predicted and experimental results are provided in Tables 4.13 and 4.14.

Map Task			**Reduce Task**		
Phases	**Read**	**Written**	**Phases**	**Read**	**Written**
reading	0	0	shuffle	0	96,832,267
buffering	0	28,012,478	sort	96,832,267	0
writing	28,012,478	28,012,478	reduce	0	0
Total cost	28,012,478	56,024,956	**Total cost**	96,832,267	96,832,267

Table 4.13: Reduce side join with filter - cost model prediction

Map task			**Reduce Task**		
	Read	**Written**		**Read**	**Written**
Total cost	28,454,731	56,884,310	**Total cost**	96,832,213	96,832,213

Table 4.14: Reduce side join with filter - real cost

4.2.1.3 Map side join

To join our files using the map side join algorithm we had to bring them in the right structure; both files had to be partitioned and sorted in the same way. To achieve this, preprocessing was needed. Two additional map/reduce jobs preceded the actual join job. Each one had as its input one file and using the same partitioner and the same number of reducers (4) we brought the input files in the right structure; each file was partitioned in 4 sorted splits. The outputs were used as input to the map/reduce job of the join algorithm. The latter, as we have thoroughly analyzed in Chapter 3 and Section 3.4, requires that no reduce tasks take place and furthermore each map task has zero I/O cost. Referring only to a single map and a single reduce task of the join job we have

Map Task			**Reduce Task**		
Phases	**Read**	**Written**	**Phases**	**Read**	**Written**
reading	0	0	shuffle	0	0
buffering	0	0	sort	0	0
writing	0	0	reduce	0	0
Total cost	0	0	**Total cost**	0	0

Table 4.15: Map side join - cost model prediction

Map task			**Reduce Task**		
	Read	**Written**		**Read**	**Written**
Total cost	0	0	**Total cost**	0	0

Table 4.16: Map side join - real cost

4.2.1.4 Comparison with real results

To facilitate a comparison between the above presented results, we constructed the following tables where all the results are gathered.

Map Task			Reduce Task		
Joins	Read	Written	Joins	Read	Written
RSJ	70,090,117	140,180,234	RSJ	170,360,091	170,360,090
RSJ-f	28,012,478	56,024,956	RSJ-f	96,832,267	96,832,267
MSJ	0	0	MSJ	0	0

Table 4.17: Joins - cost model prediction per task

Map Task			Reduce Task		
Joins	Read	Written	Joins	Read	Written
RSJ	71,597,558	143,161,670	RSJ	170,360,049	170,360,049
RSJ-f	28,454,731	56,884,310	RSJ-f	96,832,213	96,832,213
MSJ	0	0	MSJ	0	0

Table 4.18: Joins - real cost per task

Comparing the results between Tables 4.17 and 4.18 we observe that our cost model achieved a close approximation to the real cost. In general, we notice that the predicted values for the map task deviate from the real ones contrary to the results for the reduce task where the model achieved a really good approximation. In detail, the predicted cost for the map task of each join algorithm is under-predicted by a percentage that fluctuates between 0% and 2%, while for the reduce task this fluctuation is insignificant. This difference is a result of the cost formulas we used for the map task. More precisely, the standard map/reduce framework adds checksums bytes to each created file. The more the created files the greater the bytes overhead. In our cost formula we assumed that the merging algorithm during the writing phase is iterated only once yielding as many intermediate files as the rounds it consists of. However, as we mentioned in Chapter 2 and Section 2.1.3, in the actual implementation, the merging algorithm is repeated as many times as the number of the reducers and thus yields more intermediate files. Consequently, our cost formula failed to predict the additional overhead added by the checksum bytes of the files. Another reason that may have led to greater cost than the predicted one regarding the reduce side join with filter, is the false positive rate of the filter. It is likely that some disjoint records passed the filtering stage increasing the total cost.

Interesting enough is also the difference in the predicted values between the simple reduce side join and the reduce side join with Bloom filter. The predicted cost values of the map task of the former present a greater deviation than the ones of the latter. Once again, this is attributed to the map cost formulas. The greater input size of the reduce side join led to the creation of more intermediate files which, in turn, resulted in more checksum bytes.

We should also state the conclusions we can draw by comparing the cost results of the different algorithms. The map side join is undoubtedly the one that yields the least cost. Comparing the cost of the map and reduce task between the reduce side join and the reduce side join with filter, we conclude that the use of filter greatly reduces the I/O cost. In detail, the cost of the map task is reduced by a factor equal to the filter selectivity. Moreover, there is a great reduction in the cost of the reduce task as well, since the number of redundant transfers during the shuffle phase is reduced. This remark is depicted on both the real and predicted cost.

4.2.2 Total join cost comparison

In the previous section we compared the costs of each task separately. In this section we present the total cost of each join as it is estimated by our cost model and compare it with the real one. For this reason we conducted additional experiments including files of various sizes. We divided the experiments based on the distribution of the key values. In the first series of experiments, we evaluated the cost when the data distribution is uniform, switching to a non-uniform distribution in the second series of experiments. In both series we tested all join algorithms over the same set of input per series. Details are given in Table 4.19.

Experiment details					
Exp.	**Joins**	**file0 card.**	**file1 card.**	**# Mappers**	**# Reducers**
1	RSJ	10^6	$2,5 \cdot 10^6$	6	4
	RSJ-f	10^6	$2,5 \cdot 10^6$	2 — 4	4
	MSJ	10^6	$2,5 \cdot 10^6$	4	0
2	RSJ	$2 \cdot 10^6$	$5 \cdot 10^6$	10	4
	RSJ-f	$2 \cdot 10^6$	$5 \cdot 10^6$	3 — 7	4
	MSJ	$2 \cdot 10^6$	$5 \cdot 10^6$	4	0
3	RSJ	$4 \cdot 10^6$	$10 \cdot 10^6$	20	4
	RSJ-f	$4 \cdot 10^6$	$10 \cdot 10^6$	6 — 14	4
	MSJ	$4 \cdot 10^6$	$10 \cdot 10^6$	4	0

Table 4.19: Experiments

In the case of the reduce side join with filter, we separately denote the number of the map tasks that processed splits of each one of the two input files.

4.2.2.1 Uniform distribution

To estimate the total cost of each join based on our cost model we used the following approach; assuming that all map tasks receive inputs comparable in size, we estimated the cost of a single map task and used it as a representative. The same approach was used for the reduce tasks as well. We calculated the total cost by multiplying each representative cost with the total number of map and reduce tasks correspondingly.

Taking into account the assumption that the key values have a uniform distribution, we expect that the multiplication of the representative reduce cost with the total number of reducers will provide a good approximation of the real total reduce cost. As far as the total map cost is concerned, in the case of the reduce side join the cost of one map task was used as a representative while in the case of the reduce side join with filter we estimated two representative map costs, one for each participating file, due to the differentiation introduced by the filter. Finally, in the case of the map side join, there was no I/O cost related to either the map or the reduce tasks of the join. However, since the map side join requires that the input files are structured in a certain way, we estimated the total cost of preprocessing needed to restructure the files. We used our cost model to estimate the costs of the representative tasks and the results are depicted in Tables 4.20, 4.21 and 4.22 along with the corresponding real costs per task.

Predicted Cost per Task				**Real Cost per Task**			
Join	**Prep.**	**Map**	**Reduce**	**Join**	**Prep.**	**Map**	**Reduce**
RSJ		210,170,169	168,694,412	RSJ		214,608,794	168,694,400
RSJ-f		210,382,167 /84,098,385	95,306,600	RSJ-f		214,924,082 /84,682,676	95,306,540
MSJ	sort0	133,450,056	45,484,192	MSJ	sort0	136,470,961	45,484,180
	sort1	201,326,601	116,209,554		sort1	205,811,255	116,209,518

Table 4.20: Exp.1 - cost per task

Predicted Cost per Task				**Real Cost per Task**			
Join	**Prep.**	**Map**	**Reduce**	**Join**	**Prep.**	**Map**	**Reduce**
RSJ		210,270,351	340,720,181	RSJ		214,759,228	340,720,098
RSJ-f		210,270,351 /84,037,434	193,664,534	RSJ-f		214,759,228 /85,339,041	193,664,426
MSJ	sort0	201,326,643	92,630,658	MSJ	sort0	205,819,476	92,630,666
	sort1	201,326,736	234,085,385		sort1	204,785,343	234,085,396

Table 4.21: Exp.2 - cost per task

Predicted Cost per Task				**Real Cost per Task**			
Join	**Prep.**	**Map**	**Reduce**	**Join**	**Prep.**	**Map**	**Reduce**
RSJ		210,072,312	684,769,694	RSJ		214,469,952	684,769,674
RSJ-f		210,104,385 /84,033,516	390,208,678	RSJ-f		214,517,057 /85,610,468	390,208,474
MSJ	sort0	201,326,649	186,934,574	MSJ	sort0	205,788,689	186,934,514
	sort1	201,326,817	469,831,932		sort1	205,777,258	469,831,464

Table 4.22: Exp.3 - cost per task

Once again we notice that our model achieved a good approximation of the cost per map and reduce task. Noticeable deviations are observed only in the map tasks and attributed to the reasons we thoroughly presented in the previous section. Interesting enough is the observation made when comparing the map costs (either the predicted or the real ones) of each join algorithm for each one of the experiments; referring to each join algorithm separately, we can see that the map cost values are comparable in size in each experiment. As the cardinality of the files increases when we switch from one experiment to the other, the framework does its best to avoid inducing greater I/O cost on each node; it invokes more map tasks to process the input. This is also depicted in Table 4.19. As a result, job load is distributed across more tasks and the I/O cost related to each map task is prevented from being increased.

Based on these representative costs per task and by using the multiplication approach, we estimated the total cost of each join. The relevant results are presented in Tables 4.23, 4.24 and 4.25 along with the corresponding real cost as it was provided by the Map/Reduce framework.

Total Predicted Cost per Join				**Total Real Cost per Join**			
Join	**Maps**	**Reduces**	**Total**	**Join**	**Maps**	**Reduces**	**Total**
RSJ	1,261,021,014	674,777,648	1,935,798,662	RSJ	965,571,317	674,766,728	1,640,338,045
RSJ-f	757,157,874	381,226,400	1,138,384,274	RSJ-f	503,242,317	381,215,900	884,458,217
MSJ	1,072,206,516	646,774,984	1,718,981,500	MSJ	970,256,120	646,766,776	1,617,022,896

Table 4.23: Exp.1 - total cost per join

Total Predicted Cost per Join				**Total Real Cost per Join**			
Join	**Maps**	**Reduces**	**Total**	**Join**	**Maps**	**Reduces**	**Total**
RSJ	2,102,703,510	1,362,880,724	3,465,584,234	RSJ	2,044,476,871	1,362,866,800	3,407,343,671
RSJ-f	1,219,073,091	774,658,136	1,993,731,227	RSJ-f	1,115,125,070	774,670,856	1,889,795,926
MSJ	2,013,267,081	1,306,864,172	3,320,131,253	MSJ	1,960,488,991	1,306,866,824	3,267,355,815

Table 4.24: Exp.2 - total cost per join

Total Predicted Cost per Join				**Total Real Cost per Join**			
Join	**Maps**	**Reduces**	**Total**	**Join**	**Maps**	**Reduces**	**Total**
RSJ	4,201,446,240	2,739,078,776	6,940,525,016	RSJ	4,108,971,767	2,739,066,908	6,848,038,675
RSJ-f	2,437,095,534	1,560,834,712	3,997,930,246	RSJ-f	2,301,364,687	1,560,829,542	3,862,194,229
MSJ	4,026,535,332	2,627,066,024	6,653,601,256	MSJ	3,940,955,039	2,627,066,884	6,568,021,923

Table 4.25: Exp.3 - total cost per join

Based on the above tables, we can state some interesting conclusions. In general, the predicted total cost is greater than the real one. As it can be noticed, this deviation is mainly attributed to the predicted total map cost which exceeds the corresponding real one. Based on the inability of our cost model to predict the checksum bytes (as already stated in previous analysis in Section 4.2.1.4) added during each map task, one would expect the total predicted map cost to be less than the real one. When we multiplied the representative map cost with the number of map tasks, we silently assumed that every map task has a cost of comparable size. However, no all map tasks process the same size of input; when the file is split, the last split of each file can contain fewer records and consequently the corresponding map task yields less cost. As a result, the multiplication approach managed to make up for the inability of our formulas to predict the checksum overhead leading to slightly over-predicted values. However, the cost model achieved to provide a sufficient prediction of the I/O cost induced by each join algorithm.

On the other hand, the use of input files with uniform distribution of the key values, ensured that all reduce tasks receive inputs of comparable size and thus the multiplication of the representative reduce task cost with the number of the reduce tasks efficiently approached the real total reduce cost. For a better comparison between the predicted and the experimental values we plotted the cost of each join as a function of the total cardinality of the input files used in the three experiments. The plots are depicted in Figures 4.2, 4.3 and 4.4.

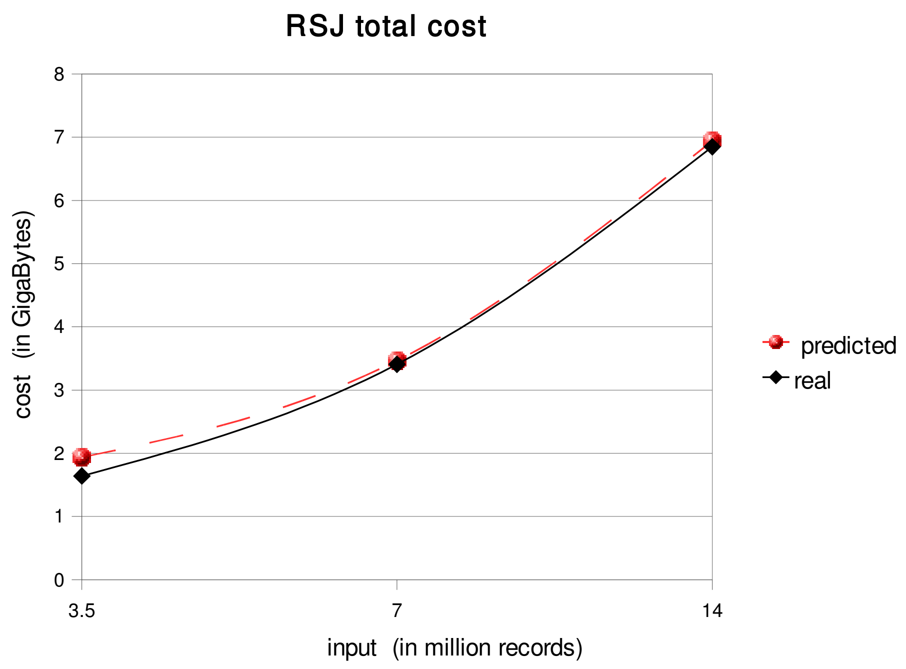

Figure 4.2: Predicted and real cost of RSJ

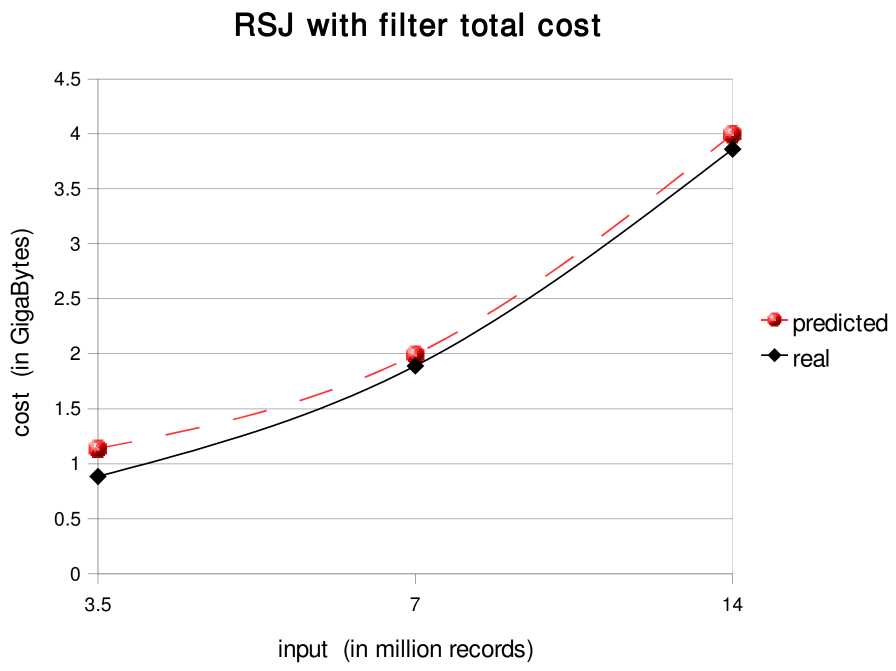

Figure 4.3: Predicted and real cost of RSJ-f

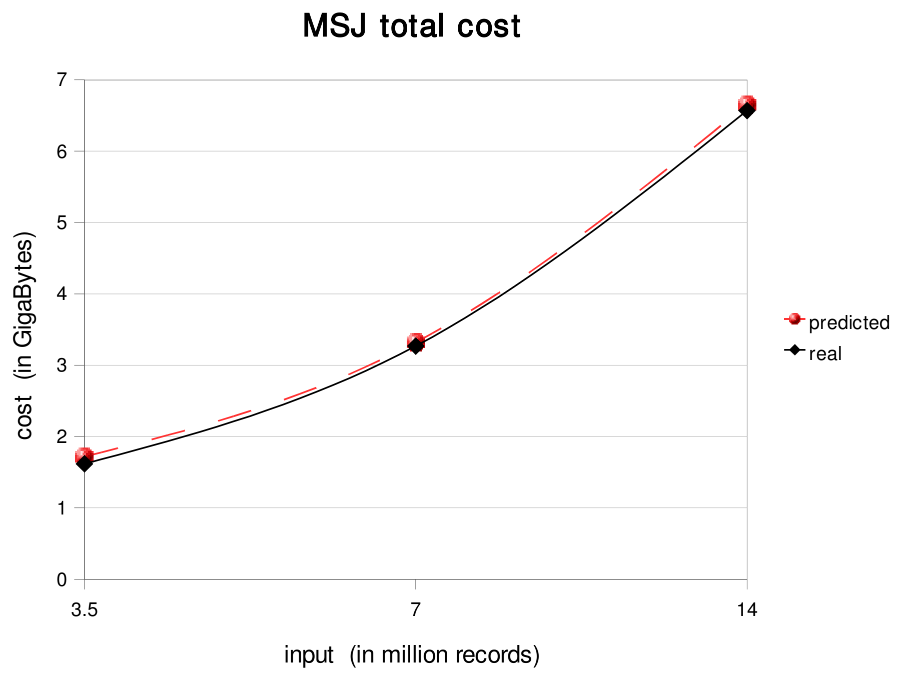

Figure 4.4: Predicted and real cost of MSJ

The above plots helped us notice two interesting points, additional to the ones we stated during the analysis of the results presented in Tables 4.23, 4.24 and 4.25.

- When the cardinality of the inputs is low, as it is in the case of the first experiment, the predicted values provided by our cost model presented greater deviation from the real cost values contrary to the cases of larger cardinality (experiment 2 and 3). Recall that in each experiment we constructed a representative cost of the map tasks. That is, we used the input split of a certain experimental map task and by applying our cost model over this input size, we created a representative map cost. Next, we multiplied this cost with the number of map tasks and found the total map cost. Moreover, recall that since each file is divided in splits of predefined size, in every case there is a split of each file that includes less records than the others. Consequently, the map task that processes it yields less I/O cost. Taking this into account and considering the total number of map tasks spawned during a job, our approach leads to a more representative map cost when more map tasks are spawned, that is, when the cardinality of the participating files is higher.
- Based on Figure 4.3, we observe that the predicted cost for the reduce side join with filter has the greatest deviation from the real cost value in comparison to the other join algorithms. This is mainly attributed to the selectivity of the filter considered when constructing the representative map cost. During the application of our formulas, the selectivity was determined as the ratio of the file records that contribute to the join results

to the total number of records included in the file. Thus we obtained two values, one for each file and using the multiplication approach, we assumed that each map task (that processes records of the same file) is related to the same selectivity value. However, these values referred to each file in total, while the actual selectivity value was slightly different in each map task since it processes only a part (split) of the total file. This assumption led to the deviation we noticed.

The above mentioned points, along with the ones stated during the analysis of the results in Tables 4.23, 4.24 and 4.25, justify the deviations of the predicted values. However, these are only minor when compared to the total I/O cost and do not degrade our cost model. In every join algorithm, the predicted curve follows the same trend as the actual cost curve strengthening our claim that the cost model offers a good approximation of the total cost.

To facilitate a comparison between the three algorithms we plotted the cost curve of each join algorithm in the same plot. This is shown in Figures 4.5 and 4.6.

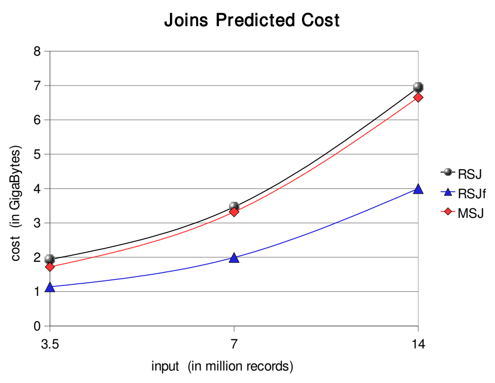

Figure 4.5: Joins predicted cost

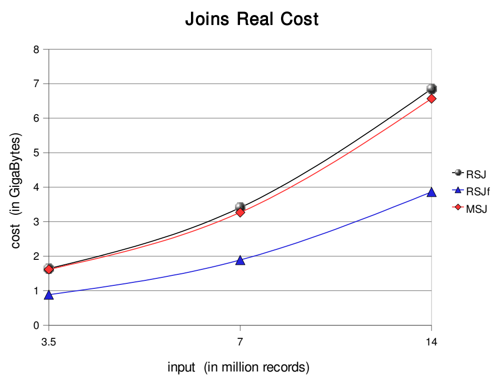

Figure 4.6: Joins real cost

Based on the above figures, our cost model managed to identify the right relation between the cost of the different join algorithms. In case the input files are fully sorted and partitioned the map side join has no I/O cost and is undoubtedly the best choice for the user. However, if the input files are not appropriately structured, then the cost of the map side join is increased by the cost of the preprocessing leading to a total cost that is comparable to the cost of the simple reduce side join. This is depicted in both figures, where we can see that the cost curve of the map side join is below the curve of the reduce side join. Moreover, the reduce side join with filter yields the least cost; depending on the selectivity of the filter the cost curve of the reduce side join with filter has an upper bound equal to the cost of the simple reduce side join (which is the case when all the records of each file pass the filtering stage, that is, they are joint records). The lower the selectivity of the filter is (fewer records pass to the reduce phase), the lower the cost curve of the join lies.

4.2.2.2 Non-uniform distribution

When the distribution of the key values is non-uniform, the reduce tasks can receive map outputs that are not of comparable size. Estimating the cost of a reduce task that has an input of a specific size and using this cost as a representative under the multiplication approach, would result in a total cost that deviates a lot from the real cost. To prevent this, a representative reduce cost is estimated as follows; the total size of data received by the reducers is divided by the total number of reduce tasks and the result is used as input to our cost formula. On the map side, the representative cost is calculated in the same way as in the uniform distribution and once again we use the multiplication approach for the estimation of the total cost.

In tables 4.26 and 4.27 we present the results of our cost model as far as the representative costs are concerned. We applied the three join algorithms over the same set of input files presented in Table 4.19 but this time the distribution of the key values was non-uniform.

Predicted Cost per Task			
Join	**Prep.**	**Map**	**Reduce**
RSJ		210,382,167	168,283,352
RSJ-f		210,382,167 /25,255,599	89,333,347
MSJ	sort0	133,450,056	45,484,778
	sort1	201,326,694	115,800,016

Predicted Cost per Task			
Join	**Prep.**	**Maps**	**Reduces**
RSJ		210,191,499	338,882,756
RSJ-f		210,191,499 /7,003,751	118,424,413
MSJ	sort0	201,326,775	92,630,666
	sort1	201,326,643	232,249,538

Table 4.26: Exp1 & 2 - non Uniform distribution - predicted cost per task

Predicted Cost per Task			
Join	**Prep.**	**Map**	**Reduce**
RSJ		210,100,632	683,183,680
RSJ-f		210,215,481 /10,541,785	264,987,532
MSJ	sort0	201,326,649	186,931,910
	sort1	201,326,445	468,250,366

Table 4.27: Exp.3 - non Uniform distribution - predicted cost per task

Based on the above representative costs, we estimated the total cost of each join algorithm. The predicted and actual values of the join algorithms during each experiment are listed in Tables 4.28, 4.29 and 4.30.

Total Predicted Cost per Join				**Total Real Cost per Join**			
Join	**Maps**	**Reduces**	**Total**	**Join**	**Maps**	**Reduces**	**Total**
RSJ	1,262,293,002	673,133,408	1,935,426,410	RSJ	963,123,038	673,133,412	1,636,256,450
RSJ-f	521,786,730	357,333,388	879,120,118	RSJ-f	438,230,817	357,333,388	795,564,205
MSJ	1,072,206,888	645,139,176	1,717,346,064	MSJ	967,804,861	645,133,448	1,612,938,309

Table 4.28: Exp.1- non Uniform distribution - total cost per join

Total Predicted Cost per Join				**Total Real Cost per Join**			
Join	**Maps**	**Reduces**	**Total**	**Join**	**Maps**	**Reduces**	**Total**
RSJ	2,101,914,990	1,355,531,028	3,457,446,014	RSJ	2,033,493,634	1,355,531,028	3,389,024,662
RSJ-f	679,600,754	473,697,652	1,153,298,406	RSJ-f	710,484,230	473,697,650	1,184,181,880
MSJ	2,013,266,826	1,299,520,816	3,312,787,642	MSJ	1,949,508,138	1,299,531,064	3,249,039,202

Table 4.29: Exp.2- non Uniform distribution - total cost per join

Total Predicted Cost per Join				**Total Real Cost per Join**			
Join	**Maps**	**Reduces**	**Total**	**Join**	**Maps**	**Reduces**	**Total**
RSJ	4,202,012,640	2,732,734,720	6,934,747,360	RSJ	4,099,488,897	2,732,734,860	6,832,223,757
RSJ-f	1,408,877,876	1,059,950,128	2,468,828,004	RSJ-f	1,589,863,440	1,059,950,128	2,649,813,568
MSJ	4,026,530,124	2,620,729,104	6,647,259,228	MSJ	3,931,487,193	2,620,734,860	6,552,222,053

Table 4.30: Exp.3- non Uniform distribution - total cost per join

The results indicate that our predictions are a good approximation of the real cost. More precisely, almost all our cost values are only slightly over-predicted than the real ones. As in the case of the uniform distribution, we attribute this deviation to the partially wrong assumption that all map tasks process inputs of comparable size. Moreover, the accurate approximation of the total reduce cost suggests that using the mean as the input to our cost formula of the reduce task led to a representative cost which rendered the multiplication by the number of reduce tasks applicable. Interesting enough are the results of the reduce side join with filter. In Table 4.28 the predicted total cost is higher than the real one while in Tables 4.29 and 4.30 is lower. This behavior highly depends on the non-uniform distribution of the key values in combination with the selectivity of the filter; we determined the later as the percentage of the total records (of a file) that participate in the join. We also considered that the representative map task has a filtering stage during which the number of records that pass to the next phase is equal to the records included in the input split multiplied by the selectivity. In the case of the uniform distribution this multiplication results in an accurate approximation of the output records in every map task. However, in the case of non-uniform distribution, the actual number of the records that pass the filtering stage in each map task can differ not only from each other but also from the representative one. Consequently, depending on the distribution of the key values in each split, the total predicted map cost can be slightly greater or lower than the real one.

In Plots 4.7 and 4.8 we depict the cost of each join as a function of the total cardinality of the inputs and provide a visual comparison between the predicted and the real values.

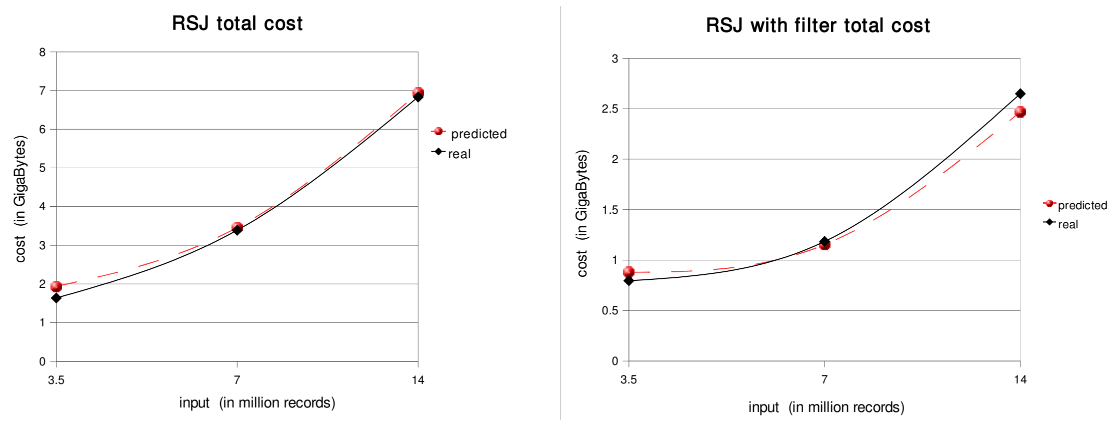

Figure 4.7: Predicted and real cost of RSJ and RSJ-f for non-uniform distribution

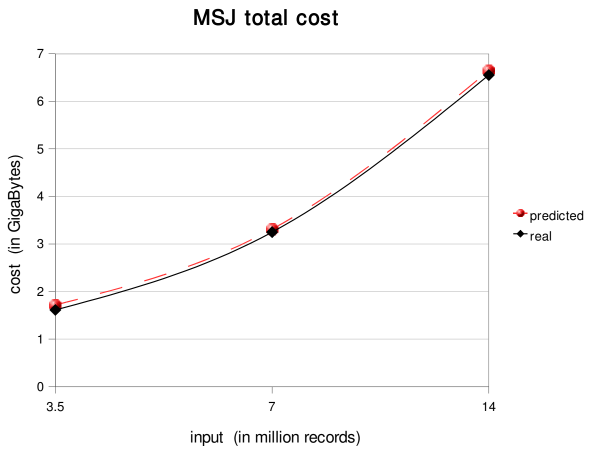

Figure 4.8: Predicted and real cost of of MSJ for non-uniform distribution

The above plots offer additional evidence that our predicted values provide an accurate approximation of the real cost values. All the predicted curves follow the same trend as the actual cost curve. Moreover, the predicted curves of the reduce side and the map side join lie only a bit higher than the corresponding real curves. On the other hand, the non-uniform distribution of the key values caused the predicted curve of the reduce side join with filter to oscillate over and below (but always close to) the real cost curve. To facilitate a comparison between the three algorithms we plotted the cost curve of each join algorithm in the same plot. This is shown in Figure 4.9.

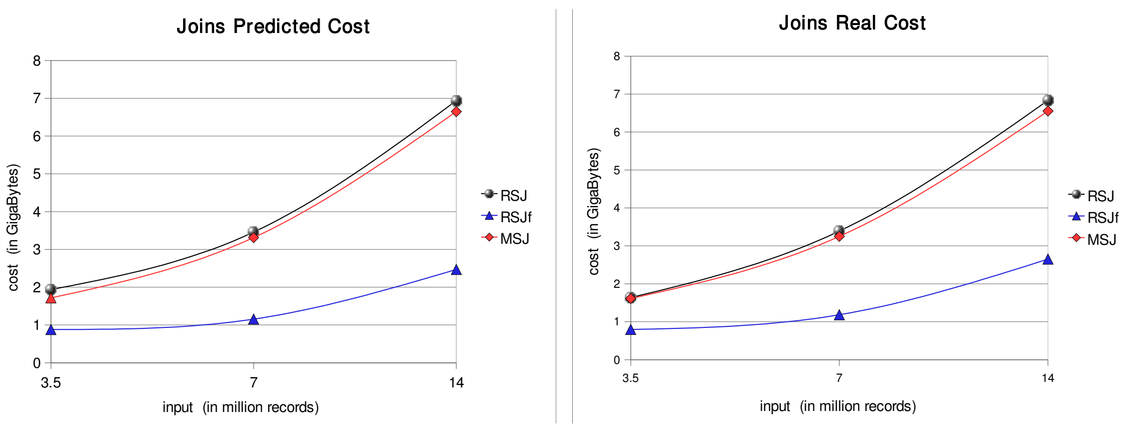

Figure 4.9: Joins predicted and real cost for non-uniform distribution

Comparing the plots in Figure 4.9 we conclude that the predicted curves exhibit the same trend as the real ones indicating that our cost model is robust against non-uniform distribution. A comparison between the three curves leads to the same conclusions as the ones stated during the analysis of the uniform distribution; the cost of preprocessing renders the map side join as expensive as the simple reduce side join, while the reduce side join with filter is the cheapest.

4.3 Discussion

In this section we conducted a series of experiments that challenged the validity of our cost model and enabled us to compare the three join algorithms; the *reduce side*, the *map side* and the *reduce side with filter* join. Our conclusions are the following

- We extracted a cost model that can successfully predict the I/O cost related to a map and a reduce task. Slight deviations from the real cost value are mainly attributed to the inability of our model to predict the overhead due to checksum bytes added by the framework during the creation of intermediate files.
- The cost model can efficiently used to predict the I/O cost of a total Map/Reduce job by multiplying the predicted I/O cost of a representative map and reduce task with the total number of the corresponding tasks. In this case the predictions are associated with an error mainly attributed to the multiplication approach. However, this error does not prevent the model from predicting cost values really close to the real ones.
- Additionally, the model is robust to non-uniform distribution of the key values.
- The optimization we introduced to the simple reduce side join by adding a filter on the map side, dominated over the simple reduce side and map side join. By reducing the total number of records that are fetched by the reduce tasks, the reduce side join with

filter induced the least cost. In any case, the simple reduce side join appears to be the most expensive. The only case in which the map side join is a better choice is when the input files are already sorted and partitioned in the same way. Otherwise, it results in a I/O cost comparable to the one induced by the simple reduce side join

Chapter 5

Conclusions and Future Work

5.1 Overview

In this project we dealt with the recently introduced Google's Map/Reduce programming model. The latter has two basic join implementations: the *reduce side* join and the *map side* join. Existing literature includes a great deal of studies that engage these join implementations to join datasets of huge size and their results are evaluated in terms of execution time and success. This project's motivation comes from the absence of research that compares the two join algorithms in terms of the I/O induced on the participating nodes. Having no access to the Google's Map/Reduce implementation, we used the corresponding open source implementation of the Hadoop project. We structured our research plan by first introducing a theoretical cost model to predict the I/O cost related to a single map and reduce tasks and then using this cost model as a tool to predict the total cost of each join implementation. To evaluate our proposed model and compare the join implementations, we set up a Hadoop cluster and conducted a series of diverse experiments. Furthermore, we further challenged the two standard join implementations by introducing an optimized version of the *reduce side* join that suggests the use of a Bloom filter on the map tasks.

5.2 Conclusions

In general, this project was completely successful. Our proposed cost model was correct and it was found to provide an accurate approximation of the real I/O cost associated with a single map and a single reduce task. We also found that it can be successfully used to predict the total cost of a Map/Reduce job. Based on this remark, we used it to estimate the total I/O cost related with each one of the three join implementations. The results revealed that, in general, our optimized version of the reduce side join, claims the least I/O cost. The only case that the map side join outperforms in terms of I/O cost, is when the participating datasets are appropriately structured. If this is not the case, the required preprocessing renders it as costly as the simple reduce side join. In this point we should mention that our cost model is tailored to the particular Map/Reduce implementation provided by the Hadoop project and in case the base Hadoop implementation changes, the equations that our cost model consists of will, naturally, have to reflect that. However, we believe that it managed to provide an insight into the I/O cost issue of the general Map/Reduce framework in such an extent that the results provided by its use can be sufficiently used to draw general conclusions.

The experiments in Chapter 4 are divided in experiments that applied the cost model to predict the cost of a single map and reduce task and experiments that used the model to estimate and compare the total cost of the three join implementations. The former yielded results that underlined the validity of the suggested cost model. The latter proved that it can be successfully used to predict the total cost of the join algorithm with values very close to the real ones and furthermore, they stressed out its robustness by testing cases of uniform and non-uniform distribution of the key values. For all the cases, the three join algorithms were compared and the I/O cost analysis showed that the reduce side join with filter is, in general, the least expensive suggesting, at the same time, that the use of metadata of the input datasets leads to optimized behavior.

5.3 Future Work

Having discussed the conclusions drawn by our work, our attention can turn to what else may be done in the future. One interesting direction would be that of introducing generality to our cost model: a cost model deprived of the limitations introduced by certain Map/Reduce implementations (as Hadoop's implementation) would be more representative of the I/O cost induced by the general Map/Reduce dataflow.

Another area that could be studied further is the memory-buffer cost of the map task introduced by the Bloom filter: our optimized reduce side join suggests that a Bloom filter is loaded in-memory during the map phase of the join algorithm. However, the size of the filter is proportional to the size of the input datasets. Having a dataset of a very large size can lead to a prohibitive size of filter which in turn can have an impact on the overall performance of the Map/Reduce process.

The construction of the Bloom filter based on one of the datasets to be joined and the promising results of the reduce side join with Bloom filter showed that the exploitation of statistics regarding the input datasets can lead to a general increase in performance. One direction of great interest would be that of finding ways to extract useful information from the datasets and employ it during the join process. For instance, an alternative to the Bloom filter could be the sampling of the key space and use of the results for the construction of a filter of acceptable size.

Bibliography

[1] Apache hadoop framework. http://hadoop.apache.org/.

[2] Hive. web page. http://hadoop.apache.org/hive/.

[3] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Alex Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *VLDB*, 2009.

[4] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *VLDB '08*, 2008.

[5] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. *SIGMOD*, 2007.

[6] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. *NIPS*, 2006.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI '04*, pages 137–150, December 2004.

[8] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. *SIGMOD '08*, 2008.

[9] A. Pavlo, A. Rasin, S. Madden, M. Stonebraker, D. DeWitt, E. Paulson, L. Shrinivas, and D. J. Abadi. A comparison of approaches to large scale data analysis. *SIGMOD '09*, 2009.

[10] Jason Venner. *Pro HADOOP*. Apress L. P., Berkeley, California, first edition, 2009.