Department of Informatics, University of Zürich

BSc Thesis

Load-Balancing Implementation in Hadoop

Thomas Walter Brenner

Matrikelnummer: 08-928-434

Email: thomas.brenner@access.uzh.ch

August 5, 2013

supervised by Prof. Dr. M. Böhlen and A. Noureldin

University of Zurich[UZH]

Department of Informatics

Abstract

The MapReduce-algorithm is a model that operates on distributed, parallel systems. Hadoop is an implementation of this MapReduce-algorithm. Some applications may produce an imbalance of work on the cluster during the execution. The goal of this thesis is to implement an load-balancing algorithm in the Hadoop framework to sort a list of timestamps. Implemented was an algorithm called TopCluster, which was developed at the universities of Munich and Bozen-Bolzano.

This algorithm gathers locally the necessary information, combines them and produces a distribution of the data in order to avoid skew. In this thesis the TopCluster-algorithm is implemented, modified to meet the necessary requirements and eventually tested with different randomly distributed data.

Zusammenfassung

Der MapReduce-Algorithmus ist eine Modell, um auf verteilten, parallelen System grosse Datenmengen zu verarbeiten. Hadoop ist eine Implementierung dieses MapReduce-Algorithmus. Manche Anwendungen können eine ungleiche Arbeitsverteilung bei der Ausführung hervorrufen. Das Ziel dieser Arbeit ist es, einen Load-Balancing-Algorithmus in das Framework Hadoop zu implementieren, um Zeitstempel zu sortieren. Es wurde ein Algorithmus namens TopCluster implementiert, der an den Universitäten München und Bozen Bolzano entwickelt wurde.

Dieser Algorithmus sammelt die notwendigen Informationen lokal, kombiniert diese und produziert eine Verteilung der Daten um eine ungleiche Arbeitslast auf den einzelnen Instanzen des Systems zu verhindern. In dieser Arbeit wurde der TopCluster-Algorithmus implementiert, modifiziert, um den spezfiischen Anforderungen gerecht zu werden, und mit zufällig verteilten Datensätzen getestet.

Contents

List of Figures

List of Tables

1 Introduction

The MapReduce-algorithm and especially Apache Hadoop [4] are powerful and cost-effective approaches to process large amounts of data in a distributed way. Hadoop offers the user many new advantages like the possibility to handle unstructured data and the possibility to provide stability throughout different platforms and different instances. In Hadoop the load of work is distributed throughout a cluster of instances. Although Hadoop is very powerful, the efficiency stands and falls with the fact, that the load of work is distributed equally in order to provide the best throughput.

Modern MapReduce implementations like Hadoop offer the user a large variety of possibilities to implement user-defined functions. Because of the many possibilities, many situations may occur where the uneven distribution of the data may lead to a longer execution time [8]. One challenge is to evenly distribute the workload among the clusters. There are many examples of scientific data, that is skewed [6] and balancing the load may become necessary. A popular example of skewed data could be the height of patients in medical studies. Another example is the distribution of mountains over the planet. Due to tectonic plate movement mountains are concentrated on several regions whereas other regions are plainer. For some application, these unequal compositions of data may lead to longer execution times. Uneven distribution due to the input data can be detected and prevented. A strategy to balance the load of work can be crucial.

This thesis is about developing and describing an approach to sort a list of timestamps in ascending order [5]. Because timestamps can be distributed unevenly, load-balancing can be useful. The process is shown in figure 1.1.

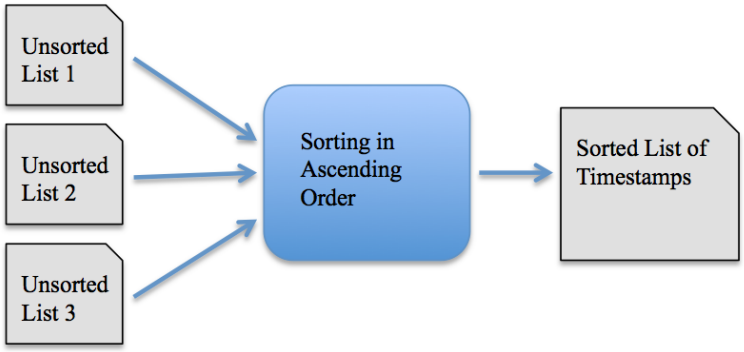

Figure 1.1: Diagram of the Input and Output

The **input data** consists of an unsorted list of timestamps of the form yyyy-mm-dd. The same timestamp can occur multiple times. Also several lists can serve as input, where the different lists are handled as one unsorted list of timestamps.

The **output** of the system is a sorted list of timestamps. This list is separated into different ordered files, which is a restriction of the system that is used to process this data. The files are ordered in ascending order as is the output data therein. Duplicate timestamps are not required and should be eliminated by the system.

The **goal** of this paper is an implementation in Hadoop, which is able to analyze the input data and in case of an uneven distribution, react to avoid skew and distribute the load of work evenly over the reducers. The implementation should deal with different distributions of data. A subgoal is to keep the overhead of this skew-avoiding system as small as possible.

The **scope** of this implementation covers the task to sort a list of individual timestamps in ascending order. This task has to be achieved by implementing the load-balancing algorithm TopCluster [7] in Hadoop [4]. It is assumed, that the processing time for each object is the same and the source of skew is only owed by the uneven distribution of the data. Even if it could be the goal of future projects to implement load-balancing for more general problems, this paper focuses on timestamp processing.

2 Related Work

In this section alternative approaches to the implementation of a load-balancing algorithm in Hadoop are discussed. The first paper in section 2.1 deals with the question, which types of skew can occur and why they occur. The second paper in section 2.2 discusses how the work is distributed evenly among the different instances. The last paper in section 2.3 deals with an approach, which handles skew by evaluating tasks during the execution and redistribute the tasks, which take more time.

2.1 Manifestations of Skew in MapReduce-Algorithms

Kwon, Balazinska and Howe [8] describe various causes and manifestations of skew. It is differentiated between skew that occurs during the map-phase and skew that occurs during the reduce-phase of a MapReduce-algorithm.

2.1.1 Skew during the Map-Phase

The paper identifies three different reasons, why skew occurs during the map-phase. It is assumed that all the mappers process approximately the same number of input values. Therefore skew cannot occur as a result of an uneven distribution of input values.

The first sort of skew is called *Expensive Record*. This sort of skew occurs, if the procession costs of individual input values, depending on the specific map-function, are significantly larger than others.

The second reason is very similar to the first reason. Since in some cases, the same application can be used to process different kinds of input data, skew can occur, because different data takes an unequal amount of time. This sort of skew is called *Heterogeneous Maps*.

The third sort of skew is called *Non-Homomorphic Maps*. Jobs in Hadoop can be executed consecutively. In some cases the data is therefore already mapped by the prior job and the map-phase of the later job is modified to execute a reduce-side task. In cases, where the map-phase executes a job, which is normally executed by the reduce-phase, the same skews as in the reduce-phase can occur. The possible skews in the reduce-phase are discussed in section 2.1.2.

2.1.2 Skew during the Reduce-Phase

During the reduce-phase two types of skew can occur. The first is analogous to the *Expensive Record* in section 2.1.1. This means, the evaluation of the reduce-function for different values is more expensive for one value than it is for another. This problem is even more pronounced

during the reduce-phase, because for a single key, a list of values has to be evaluated. This sort of skew is called *Expensive Input*.

Another sort of skew during the reduce-phase is *Partitioning Skew*. In MapReduce-algorithms the outputs of the map-phase are distributed among reduce tasks via a partitioning logic. This partitioning logic can be implemented either user-defined or using a default hash-partitioning. Whereas the default hash-partitioning in most cases distributes the load of work adequate, user-defined logic can fail to achieve this goal. In both cases skew during the reduce-phase can arise in practice and can lengthen the task execution.

2.1.3 Impact on the Thesis

For the problem of sorting timestamps in order, the three discussed sorts of skew during the map-phase are not relevant. The interesting reason for skew is the *Partitioning Skew*. During this thesis these manifestations can occur and measures are developed to detect and avoid it.

2.2 Efficient Distribution of Data to the Reducers

The task of balancing load with TopCluster [7] is split in two parts. The first task is two acquire the distribution of the data, called the cost model, and the second task is to distribute the data according to the previous acquired cost model. Gufler, Augsten, Reiser and Kemper describe in their paper [6] two strategies how this second task can be executed. This paper is to some extent the sequel to the paper [7] discussed in this thesis.

2.2.1 Fine Partitioning

The first strategy is based on the fact that more partitions than reducers have been created. This allows the system a degree of freedom to distribute them. To distribute the partitions to the reducers, the algorithms takes the biggest partition not yet assigned to a reducer and distributes this partition to the reducer with the smallest load. This step is repeated until all partitions have been assigned to a reducer. This is a relatively simple algorithm to distribute the partitions to the reducers.

2.2.2 Dynamic Fragmentation

Because the fine partitioning strategy discussed in section 2.2.1 may be ineffective in some cases, a second strategy is presented. This strategy takes partitions, which exceed the average partition size by a predefined factor, and splits them into smaller fragments. Because this fragmentation is carried out locally for each mapper, the information is transferred to a central unit. The central unit decides to fragment the partition over the whole system or ignore the fragmentation. If the central unit decides to split the partition up and reduce it by two different reducers the algorithm ensures that the necessary data and information are forwarded to the affected reducers.

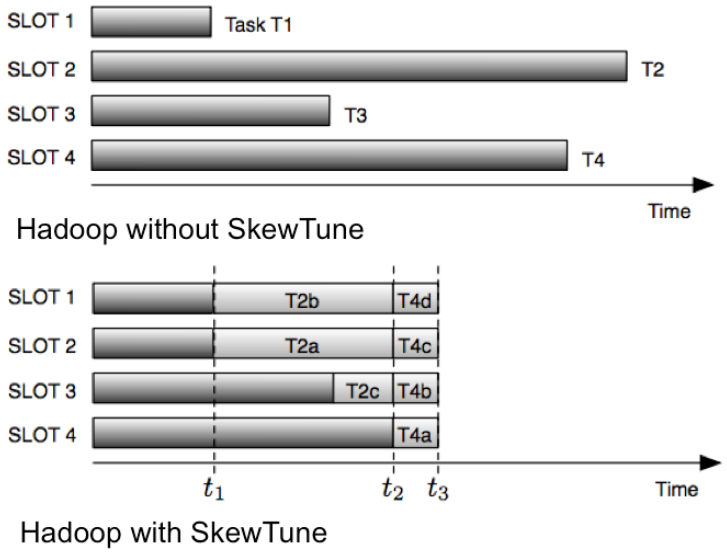

Figure 2.1: Conceptual skew mitigation in SkewTune [9]

2.2.3 Impact on the Thesis

For the specific problem of sorting timestamps both approaches do not offer an adequate solution, because these algorithm cannot ensure that the partitions remain in a sorted order. For the implementation in this paper a different algorithm to distribute the workload was chosen. For many other problems these two approaches provide valuable solutions, which can be implemented relatively simple.

2.3 A Different Approach to handle Skew

2.3.1 Explanation of the Algorithm

Acquiring information about the data distribution and adjust the work according to this information is only one way to deal with skew. Kwon, Balazinska, Howe and Rolia [9] present a completely different way with an implementation of an algorithm in Hadoop, which offers a strategy to balance the load. This implementation is called SkewTune.

A great advantage of SkewTune is, that it is applicable both during the map-phase and during the reduce-phase of the MapReduce-job. The algorithm detects, which tasks take longer than others. If a slot to becomes available the remaining values of the longer tasks are split into sub-tasks and executed by the free slots. Figure 2.1 shows a conceptual load-balancing of the SkewTune-implementation.

2.3.2 Impact on the Thesis

SkewTune represents an alternative to the implementation discussed in this thesis. SkewTune does not differentiate between map- or reduce-tasks. This implementation is again not applicable to the problem of sorting timestamps, because of the same reasons dynamic fragmentation in section 2.2.2 is not. The order of the partitions could not be guaranteed. But for the most cases, SkewTune can be a very useful alternative.

3 Foundation

3.1 MapReduce and Hadoop

To process large data sets with a parallel and distributed algorithm on a cluster the programming model MapReduce can be used. MapReduce allows a big variety of tasks, which can be performed in a relatively simple manner. The model consists of two parts. The first part is a map-function, which filters and sorts the input. The second part, the reduce-function, calculates and summarizes the results in a defined way. Multiple map- and the reduce-functions are independently carried out with small partitions of the data. This procedure allows MapReduce to work in a distributed fashion and therefore to exploit the advantages of parallelism.

Apache Hadoop is an open source implementation of the MapReduce-algorithm. It is a framework that allows the distributed processing of large data sets across clusters of computers [4]. Apache Hadoop is already frequently used by big companies like Yahoo, Facebook or eBay to process large amounts of data they are dealing with. Hadoop supports distributed applications on commodity hardware and doesn't have to use large-capacity computers. The architecture of Hadoop makes the system extremely tolerant towards breakdowns of single nodes in the cluster. Hadoop is relatively simple to use and gives the user a variety of different possibilities to process data in a user-defined manner combined with a very efficient system.

3.2 TopCluster

TopCluster [7] is an algorithm, which collects information about the global distribution of the input data throughout the execution of a MapReduce-algorithm. It offers a strategy to reduce communication and calculation time for the goal to calculate the global distribution of the data. With TopCluster skew can be detected and through the accurate use of this information the skew can be avoided.

3.3 Use Case

To illustrate the possibilities, the strengths and weaknesses of Hadoop and the TopCluster-algorithm, a specific use case consisting of a characteristic dataset and processing task is used. All the examples in the paper will be exercised with the help of this dataset to guarantee consistency.

3.3.1 Dataset

The dataset consists of fifteen timestamps. The timestamps are chosen randomly and lie within the range of 1992-01-01 and 1998-31-12. The associated partitioning function relies on the fact that all the dates are between 1992-01-01 and 1998-31-12. The problems concerning this limitation will be discussed in section 5.4. In table 3.1 the used timestamps are given.

Number	Timestamp
1	1993-04-10
2	1994-02-04
3	1996-01-30
4	1996-09-12
5	1992-01-01
6	1997-03-02
7	1994-11-24
8	1993-07-13
9	1995-10-26
10	1995-04-19
11	1998-12-29
12	1992-03-29
13	1997-12-04
14	1998-05-21
15	1994-08-13

Table 3.1: Unsorted List of Timestamps

3.3.2 Processing Task

To sort timestamps in ascending order Hadoop doesn't need special map- or reduce-functions. In order to explain the functionality of Hadoop it is necessary to have an example, where the map- and reduce-functions effectively operate. As an example for this case, the number of timestamps for each year will be counted. The same set of timestamps as in the previous section 3.3.1 will be used for illustration purposes.

4 The Hadoop Architecture

In this section the default Hadoop implementation and its operating mode will be explained.

Throughout this thesis, it has to be clearly differentiated between the Hadoop Source Code and the Hadoop Program. The **Hadoop Source Code** is the part of Hadoop, which is usually not seen by the end-user. In the Hadoop Source Code the exact process of how the different instances and nodes interact with each other is defined.

The **Hadoop Program** is the code, which consists of the specific map and reduce function and the request to start the Hadoop-Job. The Hadoop Program is sufficient to provide a great variety of powerful algorithms with the definition of the map- and reduce-function. For a regular user usually this variety of options is enough. Through giving the user such a simple way to perform MapReduce-algorithms, without giving him to much possibilities to change the process, Hadoop ensures security of a stable process during the execution and optimal performance.

Necessary information about the structure and the individual classes of Hadoop can be found in the API of Hadoop [2].

In section 4.2 the structure of the Hadoop Program is explained. Section 4.3 gives an overview of the Hadoop Source Code and explains some of the core-mechanisms of Hadoop.

4.1 Conventions

It has to be clearly differentiated between the non-modified, default version, henceforth called **default Hadoop implementation** and the modified version, henceforth called **modified Hadoop implementation**. To implement the TopCluster-algorithm some changes are necessary. In this context implementation means an alteration in the Hadoop Source Code. In this section, the architecture of Hadoop without any alteration is explained. In chapter 5, the modification to the default Hadoop distribution is being illustrated.

4.2 Hadoop Program

The Hadoop Program is divided into two sections. The first section is the definition of the essential functions for MapReduce. Besides the map and reduce functions, there are the optional partitioner and combiner functions, which can be defined by the user. These four functions are the basic modules of the Hadoop-Job. In table 4.1, the four functions are explained.

Module	Functionality	Input	Output
Mapper	The mapper converts a piece of data into a key-value pair.	input object	<key, value>
Reducer	The reducers converts a key and a list of values into the final output of the job.	<key, list of values>	output object
Partitioner (optional to implement)	The partitioner distributes the key-value pairs to different partitions. By default, every partition is reduced by one reducer.	<key, value>	the partition number of the key-value pair
Combiner (optional to implement)	The combiner can optionally combine large lists of values to smaller, for example if only the maximum of a list has to be used for reducing. The combiner is like a small reducer to improve the throughput	<key, list of values>	<key, smaller list of values>

Table 4.1: Four Functions of Hadoop

The second section of the Hadoop Program is the initialization of the Hadoop-Job. This means, store the necessary functions in the JobConfiguration, like the mapper-, reducer- and sometimes a partitioner-class, and afterward start the execution.

To process the use case introduced in section 3.3, a number of mappers and reducers had to be chosen. The fifteen timestamps will be processed by two mappers and three reducers. These specifications are summarized in table 4.2.

Specification	Usage	Value
Number of Mappers	The number of map-tasks that Hadoop performs.	2
Number of Reducers	The number of reduce-tasks that Hadoop performs.	3

Table 4.2: Specification

4.3 Hadoop Source Code

4.3.1 Important Classes of Hadoop

In this section the important classes of Hadoop are explained. With respect to the implementation of TopCluster the JobClient, the JobConfiguration, the JobTracker and for every instance of the system the TaskTracker are important. An overview of the interaction between these classes can be found in section 4.3.2.

4.3.1.1 JobConfiguration

The JobConfiguration is the central class were the settings and information are stored. The object JobConfiguration is given to many classes, in order to assure, that the whole system has access to its information. There are several ways to save the desired parameters in the configuration. Hadoop gives the user the opportunity to save the settings in a XML-file, which can be accessed by Hadoop. Another possibility is to save the settings in the Hadoop Program, for example the map and the reduce function are stored in the JobConfiguration in this way.

4.3.1.2 JobClient

The JobClient is the starting point of every Hadoop-Job. The JobClient tells the JobTracker to start a Hadoop Job.

4.3.1.3 JobTracker

The JobTracker can accept a Hadoop Job and distribute the tasks to the different nodes in the system. The JobTracker is started independently to the Hadoop Job and waits for a Hadoop Job. The JobTracker tells the nodes, which tasks they should execute and receives the statement, that the task has ended.

4.3.1.4 TaskTracker

The TaskTracker runs on every node. A TaskTracker receives the tasks (either a map or a reduce task) from the JobTracker and executes them. The tasks are being executed by a virtual Java machine (JVM). This is to make sure that bugs in the user-defined map and reduce functions don't affect the TaskTracker (by causing it to crash or hang, for example) [11]. The TaskTracker informs the JobTracker occasionally that he is still alive and is ready to execute more tasks.

4.3.2 Overview of the Execution

Figure 4.1 gives an overview of how a Hadoop Job is being executed. The shared FileSystem stands representative for the different possibilities to store the necessary data and settings in Hadoop. These can be for example, the JobConfiguration or the Hadoop Distributed FileSystem [3].

4.3.3 Map-Phase

The map-phase has the goal to build a key-value pair for each object in the split of the input data, that was assigned to this mapper. This phase consists of three sub-tasks: The acquisition of a split of the input data, the actual mapping and in the end the preparation of the map output files for further processing. Between the last two sub-tasks, in section 4.3.3.3 the use of the partitioning function is explained. This step is necessary to explain the building of the map

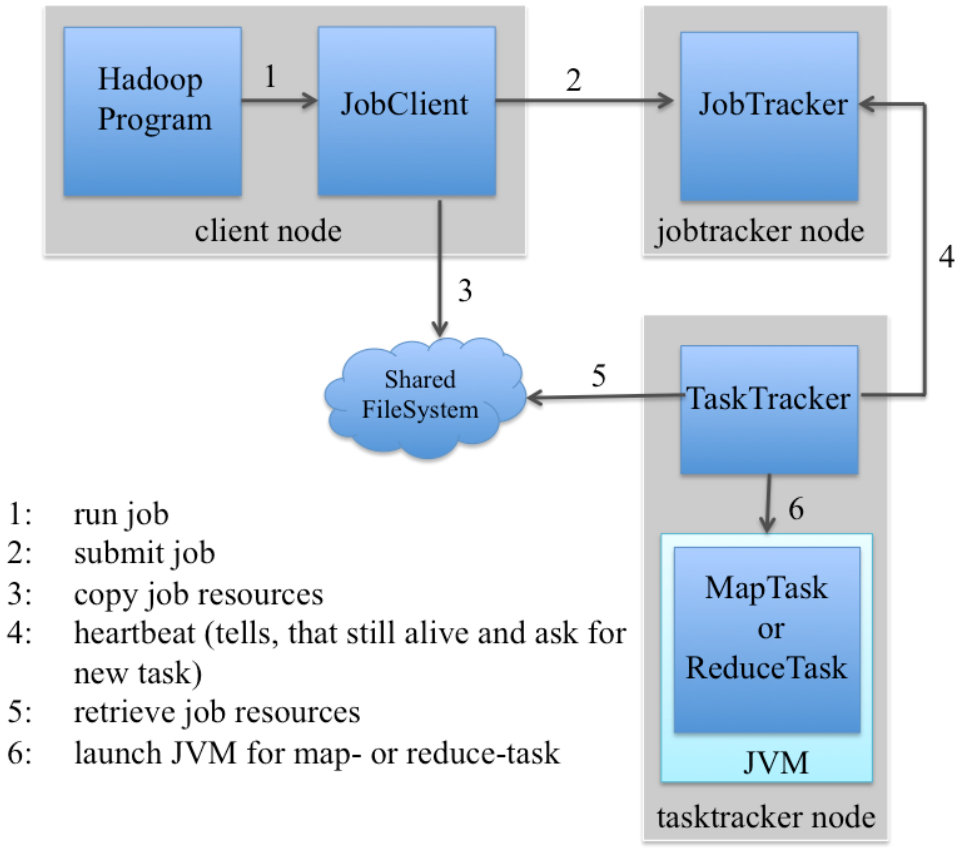

Figure 4.1: The Execution of a Hadoop Job (based on White [11])

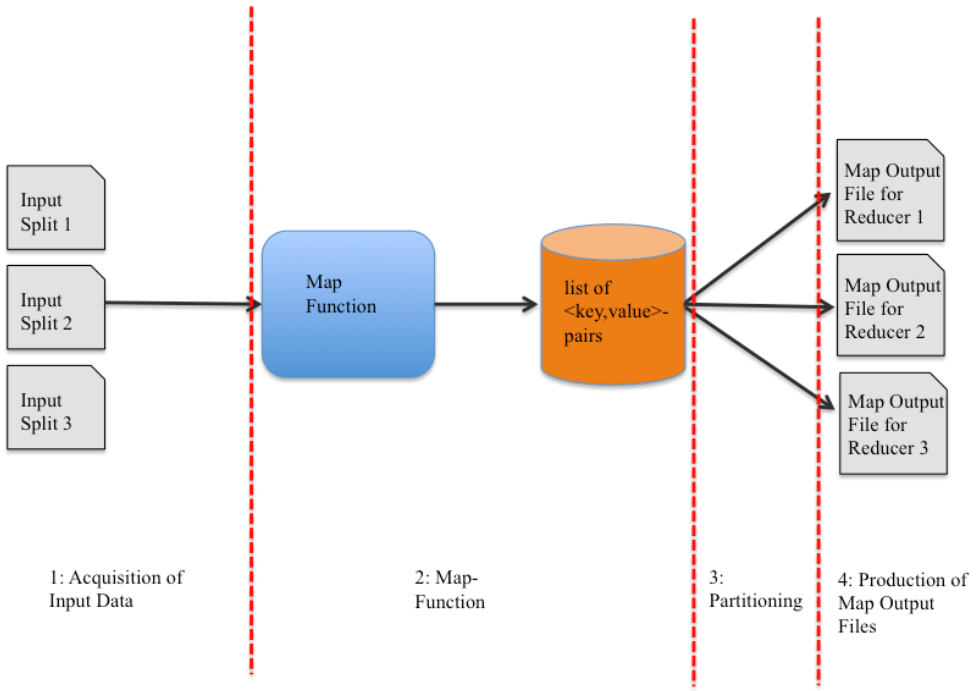

Figure 4.2: Overview of the Map-Phase

output files. A short overview of the four phases is shown in figure 4.2. The more detailed explanation follows in sections 4.3.3.1 to 4.3.3.4.

4.3.3.1 Acquisition of Data

The input of the Hadoop-Job can consist of more than one file. As it is explained in section 4.3.1.4 a Java Virtual Machine is started to perform either a map- or a reduce-task. To map the input data, obviously a map-task is initialized. The number of map-tasks started during a Hadoop-Job depends on the amount of input data. The number of map-tasks is driven by the predefined block size of the input. Up to a certain point, the number of mappers can be manipulated manually. On the one hand, this block size can be altered, on the other hand the number of map-tasks can be increased.

Using the use case in section 3.3, the fifteen timestamps will be mapped by two map-tasks. The first eight timestamps will go to a first mapper (M1), and the timestamps number nine to fifteen will go to a second mapper (M2). The following table 4.3 shows how the timestamps are distributed between the mappers:

Mapper 1 (M1)	
1	1993-04-10
2	1994-02-04
3	1996-01-30
4	1996-09-12
5	1992-01-01
6	1997-03-02
7	1994-11-24
8	1993-07-13

Mapper 2 (M2)	
9	1995-10-26
10	1995-04-19
11	1998-12-29
12	1992-03-29
13	1997-12-04
14	1998-05-21
15	1994-08-13

Table 4.3: Distribution of Timestamps between the Mappers

4.3.3.2 Map-Function

The input data is now located in the correspondent map-task instance. The next step of the map-task is to run the actual map-function, which has been designed in the Hadoop Program by the user. This map-function is deposited in the JobConfiguration and can therefore be accessed through the map-task. In detail, the map-task instantiates an object of the class MapRunner to execute this map-function. The MapRunner fetches the explicit map-function from the JobConfiguration and executes this function on every object of the input data.

This map-function converts a data object into a key-value pair. To illustrate how a map-function is working and is building the key-value pairs, the map-function that is needed to perform the example presented in section 3.3.2 is applied to the data. The key-value pairs are formed out of the input data for M1, where the key in this example is the year, and the value of every object is 1. With an adequate reducer, which will be explained in section 4.3.4.3, the number of timestamps for each year can be evaluated. Table 4.4 shows the output key-value pairs for M1.

Timestamp	Key	Value	Key-Value Pair
1993-04-10	1993	1	<1993,1>
1994-02-04	1994	1	<1994,1>
1996-01-30	1996	1	<1996,1>
1996-09-12	1996	1	<1996,1>
1992-01-01	1992	1	<1992,1>
1997-03-02	1997	1	<1997,1>
1994-11-24	1994	1	<1994,1>
1993-07-13	1993	1	<1993,1>

Table 4.4: Key-Value Pairs for M1

4.3.3.3 Partitioning

A partitioner, respectively the partitioning function, has the duty to divide the key-value pairs into partitions. It decides, which key-value pair is processed by which reducer. As it is explained in section 4.2 the user has the possibility to manually define a partitioner in the Hadoop

Program or to use the default Hash-Partitioner of Hadoop. The decision, which partitioning function should be used, depends on the nature of the problem Hadoop should manage. A partitioning function in Hadoop will produce exactly the same number of partitions as there are going to be reduce tasks.

The choice of an appropriate partitioner can be very decisive and game changing. If, for example, a partitioner by accident sends all the key-value to one reducer, the whole data set would be processed by one reducer, while the others would be without work. The partitioning function is therefore responsible for the load distribution throughout the cluster as it already has been discussed in section 2.1.2.

The partitioning function can be individually defined in the Hadoop Program. In contrast to map and reduce functions, Hadoop offers some standard partitioning function, which in many cases satisfy the need of the users. The default partitioning function uses a hash value generated out of the key of the key-value pair. The final number of the partition is this hash value modulo the number of reduce-tasks of the job. This way it is ensured, that no reducer is theoretically without work.

4.3.3.4 Production of the Map Output

After the actual execution of the map-function, the MapRunner-class, which executed the map-function, returns the mapped data to the MapTask-class. The MapTask then builds the map output files. These files are necessary for the data to be interchanged between the different nodes of the system. The map output files are constructed with the help of the partitioning function. Each mapper builds as many files as there are going to be reduce tasks. Now it will be iterated through the key-value pairs. For every entry, the partitioning function is evaluated and the entry is added to the file, where the entry will be reduced in the next phase. The output of M1, following the example from section 3.3.2, where the number of occurrences per year is evaluated, is shown in table 4.5. The Hash-Partitioner is used and the number of reduce-tasks is 3.

File Reducer 1	
Key	**Hash-Value**
1992	0

File Reducer 2	
Key	**Hash-Value**
1993	1
1996	1
1996	1
1993	1

File Reducer 3	
Key	**Hash-Value**
1994	2
1997	2
1994	2

Table 4.5: Map Output Files of Mapper M1

4.3.4 Reduce-Phase

The reduce-phase is the second main part of a MapReduce-algorithm. The goal of the reduce-phase is to evaluate all the key-value pairs with the same key and summarize these pairs to a output object. This phase consists of three sub-tasks: The acquisition of the necessary map output files, the consolidation of the key-value pairs with the same key and the actual reducing.

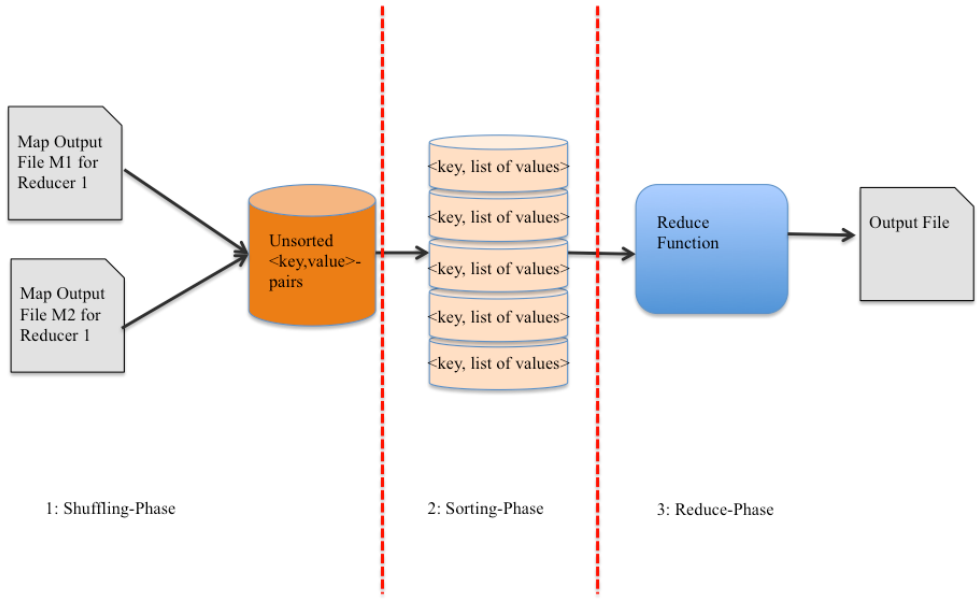

Figure 4.3: Overview of the Reduce-Phase

Similar to the execution of a map task, the TaskTracker starts a Java Virtual Machine, which then performs the reducing. A short overview of the three phases is shown in figure 4.3. The more detailed explanation follows in sections 4.3.4.1 to 4.3.4.3.

4.3.4.1 Acquisition of the Map Output and Shuffling

As it has been discussed in section 4.3.3 , the map output files for all the reduce tasks are built. To perform the reduce-step, it is necessary to acquire all the files for a specific reducer. This phase, the acquisition, is called the shuffle-phase. During the shuffle-phase the reduce task is in a while-loop. It asks the TaskTracker of completed map tasks, and if a map output is found and ready, it is transferred to the reduce task via a buffered stream. The while-loop continues until all the necessary map output files are acquired. Many security methods are implemented to assure that only one map output file per mappers is acquired.

Table 4.6 shows the finished files of the shuffle phase for Reducer 1 (hash value 0):

Key-Value Pairs of Map Output File M1	**Hash-Value**
<1992,1>	0

Key-Value Pairs of Map Output File M2	**Hash-Value**
<1995,1>	0
<1995,1>	0
<1998,1>	0
<1992,1>	0
<1998,1>	0

Table 4.6: Acquired Map Output Files

4.3.4.2 Sorting

The next phase after shuffling is the sorting-phase. During this phase, all the map output files have been acquired, and now have to be merged, sorted and converted to a proper format in order for the reduce function to iterate over the data. The data is not yet in the form the reducer requires it. Until now the data consists only of key-value pairs. The input format for the reducer is of the form <key, list of values>. Therefore the data has to be transformed. The task of transformation and merging, is done in a separate class called Merger.

The Merger-class merges the same keys together across all acquired map output files and constructs the requested format for the reduce function. During this merging process, the keys are sorted in alphabetic ascending order.

To achieve the goal of sorting the timestamps in order, this functionality will be used. After the merging and sorting, the data has temporary structure shown in table 4.7. Note, that the alphabetical ordering of the keys is favored.

Key	List of Values
1992	<1, 1>
1995	<1, 1>
1998	<1, 1>

Table 4.7: Reduce Input

Once the merging and sorting is completed, the keys with the lists of values are in adequate form to be processed by the reduce function. The map output files, which were collected during the shuffling-phase are now not longer required and are deleted.

4.3.4.3 Reduce-Function

To apply the reduce function on the data is the main task. All the phases before are for the preparation of the data in the right form to iterate over all the keys with the reduce function. As discussed in the section 4.3.4.2, all the data from all the mappers are in the form <key, list of values>. The reduce function is a while-loop, which iterates over all the keys. The actual reduce function is provided by the Hadoop Program and is stored in the JobConfiguration.

This actual reduce function, which is provided by a user (there is no default implementation for the reduce function), has a for-loop. This loop typically iterates over all the entries in the list of values. To illustrate the functionality, the evaluation of a year will be shown. Table 4.7 shows the inputs for the Reducer 1 (hash-value 0). This is the specific reduce function:

```
public void reduce(Text key, Iterator<Text> values,
OutputCollector<Text, Text> output, Reporter reporter) {
     int sum = 0;
     for (IntWritable val : values) {
          sum += val.get();
     }
     output.write(key, new IntWritable(sum));
}
```

This specific reduce function adds the values together, and gives in the end the number of occurrences of the specific key. The final output of the reduce functions OutputCollector can be seen in table 4.8.

Key	**Value**
1992	2
1995	2
1998	2

Table 4.8: Output of the Reduce Function

The data has now been processed by the reduce function and is ready to be used as output of the job. Parallel to the reduce function of the reducer, the output file of Hadoop is written. The object OutputCollector directly writes the key-value pair of the reducer to the file. It is important to know, that each reducer produces an individual output file. By default the output files are not merged.

5 TopCluster in Hadoop

This section explains the main part of this paper: the TopCluster-algorithm and its implementation in Hadoop. In section 5.1 the original TopCluster [7] will be explained. Section 5.2 discusses the modifications that are necessary to adapt TopCluster to Hadoop and the problem of sorting timestamps. Then sections 5.3 to 5.8 show step by step the modifications of the Hadoop Source Code.

TopCluster is an algorithm, that is adaptable to different MapReduce implementation. To adjust the algorithm to Hadoop and to the specific problem of sorting timestamps, some specifications have to be changed. In this paper, it will be differentiated between the **orginal TopCluster-algorithm** and the **modified TopCluster-algorithm**. The modified TopCluster-algorithm is the result of this thesis and solves the problem of sorting timestamps.

5.1 Original TopCluster-Algorithm

TopCluster [7] is one approach to deal with the balancing of processing load due to skewed data. As discussed in section 2 there exist other approaches to handle the skew. The method, described in the paper about TopCluster, is a straightforward one. The goal of TopCluster is to compute the distribution of the data in the fastest and most cost-effective way possible and with this knowledge distribute the load of work evenly throughout the different nodes of the system.

5.1.1 Definitions

A **cost model** is a model of the data, which represents the costs to process it. A cost model has the dimension of the costs, which can be for example the quantity, the data volume or the complexity of the key-value pairs. During this thesis the cost model is built out of the quantity of the key-value pairs. The goal of TopCluster is to form a cost model of the data to process. With the help of this cost model the data can be distributed in a way that all the reducers have to process the same "costs".

All the timestamps are processed by mappers. Each mapper produces a **local histogram**, which is the local image of the cost model in Hadoop. The heading of these local histograms are integrated to form a **global histogram**, which represents the distribution of the data throughout the whole system.

The **controller** represents the central unit of TopCluster. The global histogram is managed and calculated by the controller. The controller for this implementation is integrated in the JobTracker of Hadoop.

5.1.2 How the Algorithm works

The TopCluster-algorithm has been designed to provide an algorithm, which is able to create a cost model of the data. The goal of TopCluster is to keep the management overhead of this model as small as possible. Skew may as well appear during the map-phase of the algorithm, but this case shall be neglected. It is assumed, that the skew and the uneven workload can occur more likely during the reduce-phase of Hadoop. For the problem of sorting timestamps *Partitioning Skew* [8] as it has been discussed in section 2.1.2 is most likely to appear. The principle of TopCluster is, that it gathers information during the map-phase of the MapReduce-algorithm to avoid skew during the reduce-phase.

The map tasks are computed in a distributed way. Every instance has to collect a local histogram at first. All the local histograms are sent to a central point, where out of the local histograms of each mapper a global histogram is built. With the help of this global histogram the distribution is known and the system can take measures to balance the load. Because the data processed with MapReduce can be very large, the size of these local histograms may increase heavily and become infeasible for large scale data [7]. To face this problem the TopCluster-algorithm offers a methodology to approximate the global histogram and reduce the transmissions.

To reduce transmission costs, not the complete local histogram is sent to the controller, but only the header of the local histograms. The header histogram consists only of the partitions with a size bigger than a threshold value τ. There exist two approaches to define the threshold value τ. The first approach is, that τ is calculated dynamically and individually by each mapper and follows the formula: $\tau = (1 + \epsilon) * \mu_i$ where ϵ is a multiplication factor chosen by the user and μ_i is the average number of entries per partition. For the implementation a value for ϵ of 10% has been chosen. The second approach TopCluster [7] offers is not dynamic and a global τ_{global} is defined by the user. Every local histogram has a local τ_{local}. This τ_{local} is calculated by the division of the global τ_{global} by the number of mappers. In general the definition of τ manually can be difficult. For the use of TopCluster in Hadoop this approach to define τ is not advised, because necessary parameters like for example the size of the input data is not known, and the dynamic approach has been chosen.

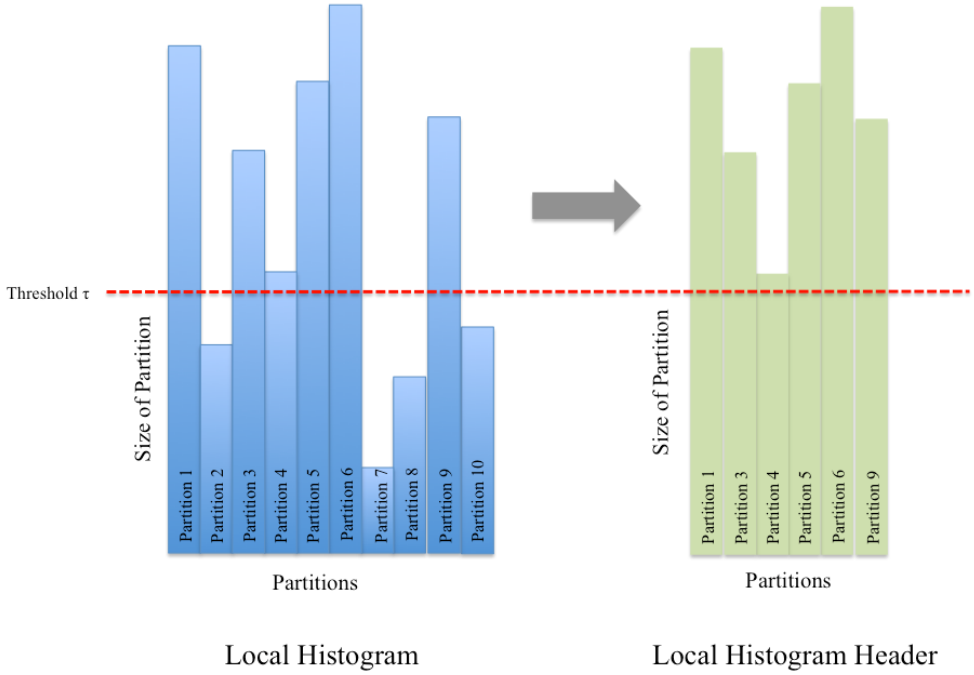

Figure 5.1: Conversion of Local Histogram to Local Histogram Header

In addition to these values above a certain threshold τ, the headers collect a presence indicator, for all the partitions which size is bigger than zero. With the help of these headers of the histograms, the overhead to process the histograms can be reduced significantly.

All the local histograms are combined together to form the global histogram. Because not the complete local histograms, but only the headers of the local histograms are collected by the controller, the main part of TopCluster is an algorithm to generate the global histograms out of these headers.

TopCluster builds temporary a lower and an upper bound histogram. In the lower bound histogram all the existing values of the headers are added and where no values exist, the lower bound histogram is filled with zeros. In the upper bound histogram also the existing values of the headers are added, but where no values exist and the headers carry a presence indicator for the partition, the smallest value of this specific header is added. If the presence indicator is negative, also the value zero is added to the upper bound histogram.

If these lower and upper bound histograms have been built, the global histogram is calculated out of the average of these two histograms.

5.1.3 Advantages and Disadvantages of the original TopCluster-algorithm

One advantage of TopCluster compared to Hadoop without load-balancing is the improved distribution of load, if there exists a skew. Another advantage is that TopCluster is very flexible. New cost models can be introduced and can be implemented without deeper knowledge

of architectural subtleties of the MapReduce-implementation.

To the contrary, if the data is evenly distributed, TopCluster produces an overhead, which isn't necessary. TopCluster has no possibility to notice before the map-phase, how the data is distributed and if a load-balancing is needed at all.

5.2 Modified TopCluster-Algorithm

This section deals with the modifications to the original TopCluster-algorithm, which are required to implement it in Hadoop to sort a list of timestamps. Two alterations are necessary to the original TopCluster: the increase of the partitions and the alteration in counting partitions instead of keys.

5.2.1 Increase of the Number of Partitions

The straightforward approach to implement TopCluster in Hadoop would be, to count the number of occurrences per key and then redistribute the keys to the existing partitions in a way, that the load is balanced. Because the global histogram and the final distribution overall is not calculated until all mappers are done, a redistribution of the key-value pairs to other partitions would be costly because all the key-value pairs had to be visited again. Therefore another approach to distribute the work has to be chosen.

Hadoop, as explained in section 4, is constructed in a way that the number of reducers and the number of partitions are equal. It follows the paradigm that every partition is reduced by one reducer. One way to balance the load is by increasing the number of partitions, where the number of the reducer remains constant. Another theoretical approach would be to redistribute some keys to other partitions, but the architecture of Hadoop denies this possibility. By the increase of the number of partitions, more than one partition could be sent to one reducer and the work load could be distributed in the way, the global histogram defines it. The altered number of partitions is determined by the number of reducers multiplied by a fineness variable. The minimum value for the fineness variable is 1, one partition for each reducer. The maximum of partitions is given by the number of different input values. A higher fineness variable on the one hand would improve the load-balancing, but on the other hand would produce more overhead for calculation purposes.

For demonstration purposes the fineness and therefore the number of partitions for the use case are specified in table 5.1.

Specification	Usage	Value
Number of Reducers	The number of reduce-tasks, that Hadoop will perform.	3
Fineness	The constant, that defines the fineness of the load-balancing	2
Number of Partitions	The number of partitions, that the data can be distributed to. This number is defined by the multiplication of the number of reduce-tasks with the fineness.	3 * 2 = 6

Table 5.1: Specification

5.2.2 Counting the Partition-Size

Where the default TopCluster-algorithm counts the number of occurrences of each key, it is sufficient for the modified TopCluster-algorithm to evaluate only the size of the partitions. Under the assumption, that duplicate timestamps are not being desired as output of the system, every key appears only one single time. Balancing according to the number of occurrences would not be successful, since this number would always be one. To avoid this problem, the modified version counts the number of timestamps per partition. To guarantee the keys are always counted to the same partition, for the calculation of the partition for the histogram and the calculation of the partition for a key-value pair as it is described in section 4.3.3.4 the same partitioning function is applied.

The two different alternatives are demonstrated with the help of the timestamps of the use case in section 3.3. To demonstrate the functionality of the original TopCluster-algorithm, the map-function, which takes the year of a timestamp as key-value, is being chosen. The original TopCluster-algorithm counts the number of occurrences per key. The global histogram of the original version is shown in table 5.2.

Entry in Global Histogram	Number of Entries
1992	2
1993	2
1994	3
1995	2
1996	2
1997	2
1998	2

Table 5.2: Global Histogram Entries for original MapReduce-algorithm with key = year

The modified TopCluster-algorithm in contrast does not count the number of occurrences per key, but the size of the partitions. To illustrate the functionality of this modified version, the timestamps of the demonstration dataset in section 3.3 are partitioned by a partitioner, which

separates the timestamps in two intervals. The first partition is for the timestamps between 1992-01-01 to 1995-12-31 and the second partition is for timestamps between 1996-01-01 and 1998-12-31. The global histogram for the modified TopCluster-algorithm is shown in table 5.3:

Entry in Global Histogram	Number of Entries
Partition 1	9
Partition 2	6

Table 5.3: Count of Elements in one Partition

The fact, that TopCluster counts the size of the partitions instead of occurrences of a specific key, reduces the work of TopCluster on the one hand, and on the other hand does not lower the efficiency of TopCluster. In section 7, where further research is discussed, the extension to a more general approach will be mentioned.

5.2.3 Distribution

The paper about the TopCluster-algorithm [7] only deals with the collecting of the information and the creation of the global histogram. To implement a load-balancing mechanism in Hadoop, another algorithm has to be introduced, which converts the global histogram into a distribution of the partitions for the specific reducers. As already discussed in section 2.2 different approaches to solve this problem exist. Due to the specification, that all the partitions have to remain in order, a new algorithm has been applied.

The distribution-algorithm calculates the average load of one reducer by dividing the number of keys by the number of reducers. Then the first partition is filled with partitions until the reducer has to process more than this threshold value of timestamps. After that the next reducer is filled with partitions until it reaches this threshold. This is carried out until all partitions are distributed.

5.3 Implementation of the modified TopCluster-Algorithm

In section 5.4 the implementation of the new partitioner, as it is necessary to increase the number of partitions is discussed. The section 5.5 explains how the local histogram is built during the map-phase. Section 5.6 deals with the building process and calculation of the global histogram. The alteration of the reduce-phase is shown in section 5.7. The section 5.8 discusses the problem of Hadoop already starting with some reduce-tasks and the influence on the modified TopCluster-algorithm. In the end in section 5.9 some of the difficulties of the implementation are shown.

In this paper the changes are not being explained on a source-code-level. The important steps are explained and are referenced to the source code. The alterations in the code are noted in a general form. This simplifies the effort to find the changes in the source code. The

form of the comments in the code have the following form "TopCluster-<class>-<sequential number>". An example in the class Test would be:

```
...
//TopCluster-Test-0001
topcluster.add(0);
//EndTopCluster-0001
...
```

All the changes to implement the TopCluster-algorithm were made in the package *org.apache.hadoop.mapred*. Only the new partitioner can be found in *org.apache.hadoop.mapred.lib*. Therefore the class names are given without the exact name of the package.

The source code can be accessed and downloaded under the address: *http://www.ifi.uzh.ch/dbtg/research.html*.

An implementation of the modified TopCluster-algorithm in the Hadoop Program is not possible. The Hadoop Program, as mentioned in section 4.2, consists only of the definition of the four modules and the initialization of the job. The implementation needs deeper access to specific functions of Hadoop and the architecture has to be modified to satisfy the requirements.

In section 4.3.1.1 it already has been discussed, that the general information and settings are stored in the JobConfiguration. The modified TopCluster-algorithm has as well some settings and information and the JobConfiguration serves the algorithms as the storage of these information. The settings can be stored in topcluster-site.xml as a XML-File. In topcluster-site.xml three important properties can be defined as shown in table 5.4.

Property	**Definition**
mapred.topcluster.granularity	The fineness of the partitions (this value will multiplied by the number of reducers and produces the number of partitions)
mapred.topcluster.startdate	The lower bound
mapred.topcluster.enddate	The upper bound

Table 5.4: Elements of topcluster-site.xml

The most important value is the mapred.topcluster.granularity. It determines the number of partitions Hadoop creates and therefore the fineness of the load-balancing.

In Hadoop the different instances have to communicate. This means, that the JobClient has to communicate with the JobTracker, and the JobTracker with the TaskTrackers. In general the various instances inform about success and progress of the different tasks. In the modified Hadoop implementation it will be necessary to send the headers of the local histograms via these communication-protocols. For big Hadoop-clusters, these histograms could become very large, therefore only the headers of the histograms (as discussed in section 5.1) will be exchanged between the TaskTrackers and the the JobTracker.

5.4 New Partitioner

An important part of the modified Hadoop implementation is the partitioning function. A new partitioner for the modified Hadoop implementation has been built as the class TopClusterPartitioner (TopCluster-TopClusterPartitioner-0060) in the package *org.apache.hadooop.mapred.lib*. This function is a key to the successful implementation. In order to equally distribute the data between the reducers, the number of partitioners has to be bigger than the number of reducers as it already has been mentioned in section 5.2.1. The partitioner to sort the timestamps calculates the partition. It is essential for the implementation that the calculation of the partition during the building of the local histograms and the calculation of the effective partition during the building process of the map output files are the same. This new partitioner is able to do this task and can not be overwritten by the user's Hadoop Program (TopCluster-JobClient-0200). Given is an algorithm to calculate the partition of a timestamp:

```
int getPartition(Date input) {
     int numberOfPartitions;
           //defined in topcluster-site.xml
     int rangeOfPartitions;
           // How many days each partitions covers,
           // calculated before
     int numberOfDaysBetween;
           // Number of Days between start- and enddate,
           // calculated by lower and upper bound
     int diff; //days between the date and startdate
     int border = 0;
     int partition = -1;
     while(diff >= border &&
           diff < (numberOfDaysBetween+1)) {
           border += rangeOfPartitions;
           ++partition;
     }
     return partition;
}
```

With the help of the demonstration dataset in section 3.3, the partitions have been built with start date 1992-01-01, end date 1998-12-31 and six partitions (3 (number of reducers) * 2 (fineness) = 6):

Number	Startdate	Enddate
1	1992-01-01	1993-03-02
2	1993-03-03	1994-05-02
3	1994-05-03	1995-07-02
4	1995-07-03	1996-08-31
5	1996-09-01	1997-10-31
6	1997-11-01	1998-12-31

Table 5.5: List of Partitions

To build these intervals it is necessary for the implementation to define the range in which all the timestamps are located. The user can define a lower bound and an upper bound in topcluster-site.xml as the properties *mapred.topcluster.startdate* and *mapred.topcluster.enddate*. It is important for all the timestamps to be between these bounds, otherwise Hadoop can't handle them.

5.5 New Map-Function

First the partitioner and the local histogram are initlialized with the specific data from the JobConfiguration (TopCluster-MapRunner-0003). Important information is for example the fineness, the number of reducers and the number of mappers. With this information the local histogram can be built (TopCluster-MapRunner-0001).

5.5.1 Collecting the Data for the Local Histogram

During the process of mapping, the map-function iterates over all the values and maps them. After each mapping step for one key-value pair, this pair is added to the corresponding bar in the local histogram (TopCluster-MapRunner-0004). The partition is calculated by the new partitioning function as explained in section 5.4. If all the values are processed by the mappers, the local histogram already has been filled entirely with the necessary data.

5.5.2 Local Histogram Header

The next step is the calculation of the header of the local histogram and the transmission to the JobTracker. This calculation is handled by the MapRunner-class, where already the information of the local histogram is kept. The task however is initialized by the MapTask-class. After the successful execution, the headers are sent to the JobTracker to be administrated further (TopCluster-MapTask-0040).

The calculation of the header is handled by the MapRunner-class (TopCluster-MapRunner-0002). An object LocalHistogramMini (TopCluster-LocalHistogramMini-0025) is created, which represents the header of the local histogram. The header is built analogously to the explanation in section 5.1 with a dynamic threshold and $\epsilon = 0.1$.

5.6 Building the Global Histogram

After all the mappers have finished with their tasks, the JobTracker has collected all the local histograms and is ready to calculate the global histogram. The global histogram gives the system the knowledge about the distribution of the data across it. The global histogram is produced in the class GlobalHistogram (TopCluster-GlobalHistogram-0090).

The calculation of the global histogram follows the rules already explained in section 5.1. Briefly it is about calculating a lower and an upper bound histogram. These temporary histograms represents the estimated maximum and minimum of the partition sizes, because only the headers of the local histograms are collected, and therefore not all information is processed. The global histogram is the result of the average between lower and upper bound histogram. This global histogram is calculated after all the mappers have finished (TopCluster-GlobalHistogram-0091). The global histogram for the use case is shown in table 5.6.

Partition	Number of Timestamps
1	2
2	3
3	3
4	2
5	2
6	3

Table 5.6: Global Histogram

After the global histogram is built, the next step is to calculate how the partitions will be distributed to the reducers. A function in the GlobalHistogram-class (TopCluster-GlobalHistogram-0092) builds this distribution. The requirements can only be met through keeping the order of the partitions. To calculate the distribution, the number of key-value pairs is counted. This number is divided by the number of reducers, which produces the average load of a reducer. The algorithm to distribute the partitions, now starts at reducer 1 and adds so many partitions, until the number of key-value pairs is even or bigger than the average work load of the reducer. In the example of the use case, the average work load of a reducer is 15/3 = 5 key-value pairs per reducer. The partitions are distributed to the reducers as shown in table 5.7.

Reducer	Allocated Partitions	Number of key-value pairs
1	1,2	5
2	3,4	5
3	5,6	5

Table 5.7: Distribution

The last task after the calculation is to tell the JobTracker, that the global histogram has been calculated and therefore the distribution is ready. Now the reducers can start their work, because they know, which partitions will be processed by them.

5.7 New Reduce-Function

Some changes in the reduce task had to be implemented as well. The first change is a loop for the reduce task to wait for the calculation of the global histogram to finish (TopCluster-ReduceTask-0500). The JobTracker is asked every three seconds, if the mappers all have finished. If they did, a function is initiated, which acquires the start and the end partition of the partitions that will be processed by the reducer (TopCluster-ReduceTask-0510). For example reducer 1 gets the start partition 1 and the end partition 2 as it was calculated in table 5.7. The individual reduce tasks get their individual start and end points as it shown in the table 5.8.

Reducer	Start Partition	End Partition
1	1	2
2	3	4
3	5	6

Table 5.8: Start and End Partition

The reduce task acquires the necessary map output files (TopCluster-ReduceTask-0530). Until now, the reduce task was allowed only to import one map output file per mapper, but with this alteration, it now will acquire all the necessary files. Another little alteration had to be made concerning the temporary names of the map output files (TopCluster-ReduceTask-0520), because now there exist more than one file per mapper.

It has to be mentioned, that some security mechanisms responsible for the limitation to acquire exactly one file per mapper had to be disabled throughout the whole ReduceTask-class.

5.8 Slowstart of Reduce-Phase

It is a paradigm of Hadoop, that not all the map tasks have to be finished before the system initializes the first reduce tasks. This mechanism is called slow start. Obviously no reduce task can finish before all the map tasks are finished. But with the help of the slow start the first two phases of the reduce-phase, shuffling and sorting, can already start. In some cases, this functionality allows Hadoop a higher throughput. [11]

For the implementation of the TopCluster-algorithm, this slow start can not be applied. The reduce tasks need all the map tasks to have finished and the global histogram to be calculated before they can start. It is advised to shut this slow start mechanism down, by setting the property *mapred.reduce.slowstart.completed.maps* in the mapred-site.xml to 1. Otherwise some reduce tasks already start and keep waiting, due to the modified TopCluster-implementation, until the global histogram has been calculated and block resources in the mean time.

5.9 Challenges during the Implementation

During the implementation of TopCluster in Hadoop some challenges occurred:

- A first challenge to modify the Hadoop Source Code was to compile and execute the code properly. Hadoop uses another framework called Ant [1] to compile the source code. This framework had to be installed.
- Another challenge was at the beginning the transmission of the headers of the local histograms as an object. The object had to be serialized into an array of bytes, transmitted and deserialized in the JobTracker.
- The acquisition of more than one map output file per mapper was a challenge. The system only allowed one file per mapper and in instance of the ReduceTask. A large number of security measures had to be deactivated. It was a challenge to shut down only the minimum number of measures in order to prevent the highest standard of security.
- The impossibility of the combination of map output files due to issues with a checksum feature of the Hadoop Distributed File System had been a problem as well. For a time it had been an idea, not to deal with all the map output files individually, but merge the files to one file. This idea had to be discarded due to issues with this checksum feature.

6 Testing the Implementation of the modified TopCluster-Algorithm

6.1 Coefficient of Variation as Measurement Category for Skew

To measure the quality of the implementation, a measurement indicator to evaluate the even distribution of the workload is being introduced. In this thesis the coefficient of variation σ_r is used to measure the skew. σ_r is the relative standard deviation. This means the standard deviation divided by the expectation E. In this section the expectation E is the theoretical average load of one reducer. The coefficient of variation therefore tells that 68.3% of the values are in an interval $[E - \sigma_r, E + \sigma_r]$ [10]. A decreasing coefficient of variation σ_r means that the load is distributed more evenly. In figure 6.1 two examples of distributions are shown, one shows a distribution with an uneven distribution and a high coefficient of variation, the other one an even distribution with a low coefficient of variation.

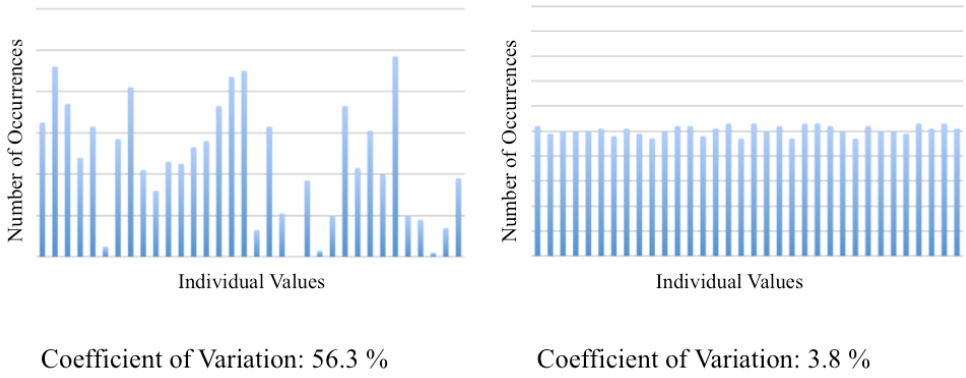

Figure 6.1: Two Examples of different Distributions

6.2 Approach

The quality of the equal distribution of the data using the modified Hadoop implementation is being tested. The test data has been produced by the class DataBuilder in the folder *TopClusterTools*. DataBuilder can generate different distributions in order to test the the functionality and the quality. The number of peeks and the size of the peeks can be varied to test different data distributions. The timestamps are in the range of 1992-01-01 to 1998-12-31. For testing

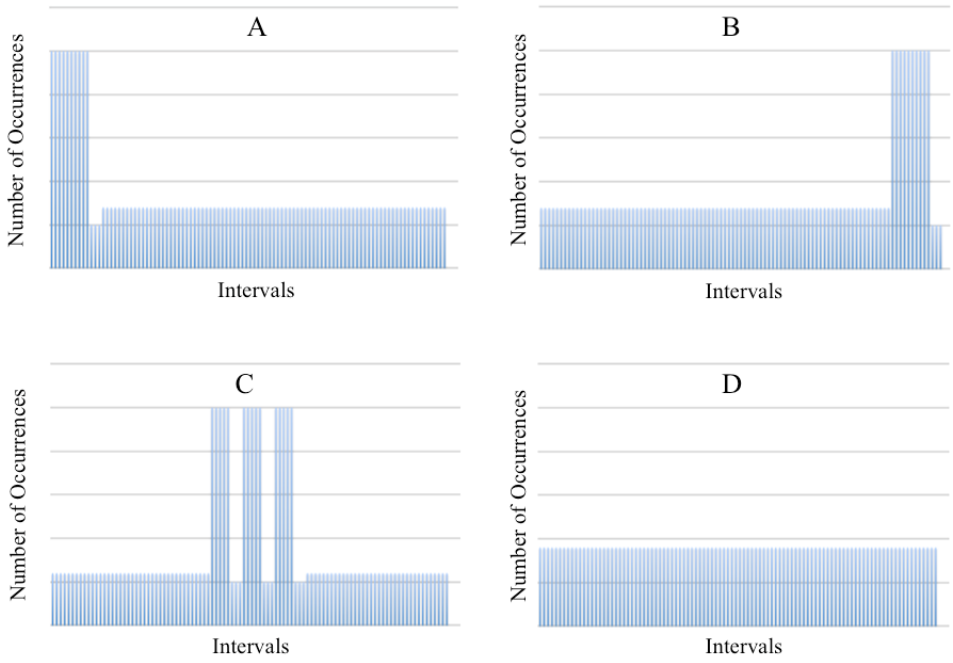

Figure 6.2: The four tested Distributions

purposes, the timestamps are being supplemented by an individual, random value. This was made to ensure that each timestamp is unique. The random value has no effect on the proper functionality of the modified Hadoop implementation. The test has being conducted by a cluster with one instance. For every job the coefficient of variation σ_r has been calculated in order to evaluate the quality of the implementation.

To illustrate the efficiency of the load-balancing of Hadoop, the processing of four different distributions with the modified Hadoop implementation is going to be evaluated. Some of the parameters, like fineness and the number of reducers, are being varied as well. The four different distribution are shown in figure 6.2 and named A,B,C and D. Table 6.1 shows the coefficient of variation for these four distributions.

Distribution	**Coefficient of Variation**
A	60.6%
B	60.4%
C	76.2%
D	0%

Table 6.1: Start and End Partition

The goal of this evaluation is to demonstrate, that the modified Hadoop implementation lowers the coefficient of the variation and therefore improves the load-balancing. The distribution D has been chosen to test the modified implementation with an already equal distribution of data.

6.3 Test Results

In table 6.2 the results of the test are shown. The test was conducted with 1'000'000 timestamps. The data was mapped by two mappers.

Distribution	**Number Of Timestamps**	**Number of Reducers**	**Fineness**	**Number of Partitions**	**Theoretical Average Load of one Reducer**	**Coefficient of Variation** σ_r	**Execution Time in s**
A	1'000'000	4	4	16	250'000	**33.2**	72
B	1'000'000	4	4	16	250'000	**51.9**	82
C	1'000'000	4	4	16	250'000	**53.3**	68
D	1'000'000	4	4	16	250'000	**25.0**	72
A	1'000'000	8	4	32	125'000	**19.4**	136
B	1'000'000	8	4	32	125'000	**44.4**	108
C	1'000'000	8	4	32	125'000	**51.2**	109
D	1'000'000	8	4	32	125'000	**9.6**	125
A	1'000'000	4	20	80	250'000	**11.5**	78
B	1'000'000	4	20	80	250'000	**4.9**	82
C	1'000'000	4	20	80	250'000	**11.2**	81
D	1'000'000	4	20	80	250'000	**4.9**	84
A	1'000'000	8	20	160	125'000	**12.4**	129
B	1'000'000	8	20	160	125'000	**5.4**	136
C	1'000'000	8	20	160	125'000	**17.6**	152
D	1'000'000	8	20	160	125'000	**9.0**	145
A	1'000'000	4	100	400	250'000	**3.3**	112
B	1'000'000	4	100	400	250'000	**1.9**	104
C	1'000'000	4	100	400	250'000	**2.8**	100
D	1'000'000	4	100	400	250'000	**1.1**	112
A	1'000'000	8	100	800	125'000	**3.8**	171
B	1'000'000	8	100	800	125'000	**1.6**	173
C	1'000'000	8	100	800	125'000	**3.3**	189
D	1'000'000	8	100	800	125'000	**1.5**	179

Table 6.2: Standard Deviation in Percent to Average Load

6.4 Conclusion

The results in section 6.3 show that the modified Hadoop implementation with the TopCluster-algorithm solves the problem stated in section 1. For data with skew, the standard deviation could be reduced significantly in comparison to not using the extension.

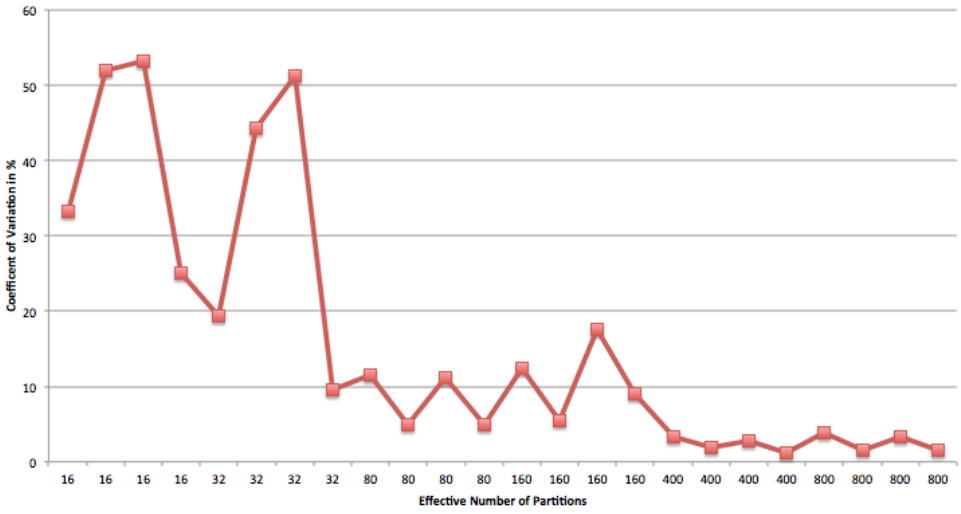

Figure 6.3: Coefficients of Variation in relation to the Number of Partitions

The test results show that with an increasing number of partitions the load-balancing becomes more precise. The coefficients of variation are summarized in figure 6.3. It can be observed, that the initial distribution has an influence on the quality of the load-balancing, but the number of partitions has a greater effect. With the increase of the number partitions the quality of the load-balancing increases as well, but on the other hand can become a bottleneck, if the management overhead is too large.

Because the Hadoop Job has been executed on one instance, the execution time is only partly significant. As it can be observed, all the execution with eight reducers take longer than the executions with four reducers. This is, because the system has an overhead to execute an additional reduce task. But still it can be seen, that with the increase of fineness, the execution time increases along, because of the communication and calculation overhead of TopCluster.

6.5 Comparison to the Default Hash-Partitioner

In the sections 6.1 to 6.4 the reduction of the coefficient of variation by using the modified Hadoop implementation was tested. In this section it is experimented with a worst case scenario, where the data is constructed in a way that all of the key-value pairs are reduced by only one reducer. As already discussed in section 4.3.3.3, Hadoop uses by default a partitioning function, which classifies the key-value pairs by a hash-value. It can be created a dataset in a way, that all timestamps have the same hash-value and are therefore sent to the same reducer.

This experiment was successful. If the generated data is a list of unsorted timestamps, the default Hadoop implementation sends all the timestamps to one reducer, which means a coefficient of variation of 200% for 4 reducers. The experiment was conducted with all the four distributions introduced in section 6.2 and the result was always the same. All the timestamps were sent to only one reducer. The results are shown in table 6.3.

Distribution	**Load Reducer 1**	**Load Reducer 2**	**Load Reducer 3**	**Load Reducer 4**	**Coefficient of Variation**
A	1'000'000	0	0	0	200%
B	1'000'000	0	0	0	200%
C	1'000'000	0	0	0	200%
D	1'000'000	0	0	0	200%

Table 6.3: Distribution with all timestamps having Hash-Value 0

This experiment should show, that for all sort of problems skew may occur. TopCluster can provide a solution to detect skew and avoid it. The implementation in this paper only deals with load-balancing a list of timestamps. In section 7 the extension to a more general level is discussed. The implementation presented in this thesis is a first step for a general implementation of TopCluster in Hadoop.

7 Further Research

In this chapter two fields of further research questions is discussed. The first section deals with the generalization of TopCluster. The second section deals with the extension of the cost model.

7.1 A general TopCluster Implementation

The implementation of TopCluster has been adjusted to the problem of sorting timestamps as stated in the introduction of the paper. Of course a more general implementation of TopCluster could be very useful and could be an interesting question for further work. The problem of data skew can occur for many different problems. A further research question could be, how the algorithm could be extended to a more general use. By a generalization the use of the implementation would increase. In the next section the alteration to the current implementation are illustrated in a general way.

Basically only two major alterations have to be made: the use of the partitioning function and a new algorithm to calculate, which reducer will process which partitions. The calculation of the global histogram could be kept as it is implemented now.

7.1.1 Alteration to the Partitioner

To generalize TopCluster, the implementation has to be extended to accept all possible partitioners. It is important to know, that the default partitioners, which Hadoop offers, always produce as many partitions as there are reducers. To use TopCluster generally, it would be important to ensure, that the partitioner produces enough partitions to give TopCluster the possibility to distribute them equally. This means, that the partitioner produces more partitions than there are reducers.

7.1.2 Problems of the Partitioner

There may occur severe problems to implement TopCluster for a general use. The source of these problems is the use of the partitioner. It is important to understand, that the implemented partitioner (TopCluster-TopClusterPartitioner-0060) was necessary, because of the goal to sort the timestamps. For a more general use, it may be necessary for the user to define an individual partitioning function and not using the default partitioning function provided by Hadoop. Because the number of reducers and the number of partitions won't be equal, as it is for the default Hadoop implementation, the user has to be aware of this fact. He only then can create a partitioner, which respects the necessities of TopCluster. The default Hadoop implementation

faces the same problem. The user has the possibility to define a partitioner, which produces problems, because the number of partitions is not equal to the number of partitions. Therefore the user can define a partitioning function with a number of partitions, that isn't fit for Hadoop. Until now, Hadoop has no security measures to prohibit this fact. A general implementation of TopCluster does not augment this problem, but it has to be mentioned, that this problem can be severe, and could make TopCluster useless, if the partitioner is implemented wrong.

7.1.3 Alteration to the Distribution

Because the partitions had to remain in order, the distribution algorithm had been adjusted to this specific problem (TopCluster-GlobalHistogram-0091). The implementation in this paper depends on the fact that the partitions stay in order. If this specification is obsolete, a more efficient algorithm could be implemented to distribute the partitions to the reducer. There exist more complex and efficient algorithms. Gufler et al. [6] provide some solutions to deal with the distribution problem as it has already been discussed in section 2.2. In addition, other algorithms, which concerns the specific location of the big partitions could be very useful and spare time, which is required to copy the big partitions to the right reducer.

7.2 Extension of the Cost Model

An interesting question also is a change of the cost model. During the implementation, only the number of key-value pairs have been calculated. This is satisfying for problems, where the calculation of every key-value pair is about the same. But there may occur tasks for Hadoop, where the complexity of the task varies between different key-value pairs. A new cost model could help TopCluster or similar load-balancing algorithms to become more efficient.

Bibliography

[1] Apache. Welcome. 12. July 2013. 17. July 2013. <http://ant.apache.org/>.

[2] Apache. Hadoop 1.0.4 API. 15. July 2013. <http://hadoop.apache.org/docs/r1.0.4/api/overview-summary.html>.

[3] Apache. *HDFS Architecture Guide.* 6. March 2013. 15. July 2013. <http://hadoop.apache.org/docs/stable/hdfs_design.html>.

[4] Apache. *Welcome to Apache™ Hadoop®!.* 10. July 2013. 15. July 2013 <http://hadoop.apache.org/>.

[5] Brenner, Thomas. Constant Interval Extraction using Hadoop. University of Zurich in course Vertiefung. March 2013.

[6] Gufler, Benjamin, et al. "HANDLING DATA SKEW IN MAPREDUCE."

[7] Gufler, Benjamin, et al. "Load balancing in mapreduce based on scalable cardinality estimates." *Data Engineering (ICDE), 2012 IEEE 28th International Conference on.* IEEE, 2012.

[8] Kwon, Y., et al. "A study of skew in mapreduce applications." *Open Cirrus Summit* (2011).

[9] Kwon, YongChul, et al. "Skewtune: mitigating skew in mapreduce applications." *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* ACM, 2012.

[10] Papula, Lothar. *Mathematische formelsammlung: Für ingenieure und naturwissenschaftler.* Vol. 6. Springer DE, 2009.

[11] White, Tom. Hadoop: the definitive guide. O'Reilly, 2012.