TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRONIC AND COMPUTER ENGINEERING

Master Thesis

BucDoop: Bottom Up Computation of Iceberg Data Cubes With Hadoop

Kostantinos V. Tsakonas

Thesis Committee

Associate Professor Antonios Deligiannakis (Advisor)

Professor Stavros Christodoulakis

Professor Minos Garofalakis

Chania, July 2014

Abstract

Big Data analysis has been a key matter during the recent years for the study of various phenomena in various science contexts as well as in business intelligence. Furthermore it appears for good reason to remain in focus for the future. Online Analytical processing methods and Data Cubes need to be further studied in order to reduce time used for efficient data analysis. This study introduces BucDoop, a novel algorithm that exploits the parallelism benefits of Hadoop Map Reduce, for the efficient iceberg data cube creation in reasonable time. BucDoop includes the use of the Bottom Up Computation (BUC) idea in the context of iceberg cube data lattice traversal, managing to reduce the amount of data handled with early pruning architecture and producing the portion of the cube needed for analysis purposes (iceberg problem). Experiments conducted herein present an efficient scalability factor for the creation of the iceberg cube for very big data, by-passing the data explosion and memory constraints problem while using only commodity hardware.

Keywords: OLAP · Data mining · Online analytical processing · Data cube · Iceberg cube · Data aggregation · Bottom Up aggregation · BUC · Bottom Up Computation · Hadoop · Map Reduce

Table of Contents

List of Figures

List of Tables

Acknowledgements

To those people having initiated this journey as well as to those having been part of it, I express my gratitude:

To my daughters, Nelli and Stevi who contributed with their time and make all days better,

To my wife, Irini who is sharing this seeking path with me,

To my parents, Vasilis and Kanella who initiated the sparkle for all seeking, and stand patiently helping all the way, and my sisters Chrysiis and Silia for the encouragement,

To my Advisor, Associate Professor Antonios Deligiannakis, as well as to thesis committee members, Professor Stavros Christodoulakis and Professor Minos Garofalakis, whose guidance, advice and support made this possible,

To candidate PhD student Giannis Flouris who was always patiently there to provide advices and food for thought,

To TUC Cluster administrator Polixeni Arapi whose support was available at all times,

To all Teachers and Professors that contributed to my knowledge trip,

To TUC administration and technical personnel who made it pleasant to work with.

Chapter 1 - Introduction

1.1. General

The gradual introduction of IT technological means in almost every single human activity during the previous decades, led to an exponential increase of the available information. These information come from data collections created over trillions of interactions, from various human-machine or even human-led machine-machine interfaces. Huge amounts of data are produced and exchanged during a single day, and most of them are stored for further exploitation and analysis. The efficiency that the IT systems conferred to human transactions, and the included ability to store and post-analyze this transactions through the systems, created more outcome; many stories to be read. Stories that talk about humans and their behavior. Stories talking about what they did and, if read carefully, probably telling what they will do again.

So the massive demand procured the previous years globally, from either Industrial, Governmental, Academic or Commercial and many other entities, was to read those stories and use them to get answers. From the need for digitization of aids we moved to the need for digitization of efficiency. Questions now are laying there more than ever, giving birth to more questions at the same time they are being answered. Telephone company logs, market place transactions, weather datasets, geological surveys data and thousand more cases, leading to a need for efficiently handling data and extracting useful information from them.

And until not long ago, most of the respective organizations and entities, invoked endless human-hours in order to manage manually to get results concerning their efficiency in their work cycle. In such a case the way ahead was clear and led all the relevant Academic as well as Commercial institutions to define a new strategy in order to cover the actual community needs; Data Analysis.

1.2. Data Analysis

The most commonly known systems where data is stored, which are in use widely for many decades now, is the data bases systems. These systems are based on the Online Transaction Processing architecture which focuses in the significance of availability, speed, concurrency, and recoverability. These data storage tools, do not offer themselves for data analysis. Data that are changed rapidly for the sake of concurrency and speedy performance would be rather not feasible to handle.

On the other hand, analyzing data which represent an obsolete instance would probably lead as to wrong conclusions. What was needed was the ability to have all the information and the means to handle them during analysis.

This sector was covered by the Online Analytical Processing architecture as the philosophy behind handling data for analysis, together with the Data Warehouses which are assigned with the task to provide infrastructure for storing and analyzing vast datasets.

Data Warehouses, focus on the significance of subject-oriented functioning as they are used to analyze a particular subject area such as "sales" of a store, integration ability for storing and analyzing data from multiple data sources, time-variant ability in order to be able to keep historical data useful for analysis, and nonvolatile nature since data stored should be kept as is and not subject to any changes.

Storing data in data warehouses for analysis produces huge amount of information to be analyzed. And by referring to analyzing, procedures much more complex than simple searching queries invocation are involved, in order to discover hidden patterns behind the data, while we do not know that they exist.

These procedures cover the area of data mining, and reside in the space of huge data and big time to analyze them. The reduce of the time needed for analyzing data in an efficient way, are problems handled with the Online Analytical Processing architectures.

Among existing strategies for minimizing processing time, which could be bigger than tens of hours for a query over a typical dataset, is the creation of all possible query answers in advance. With this strategy, analysis personnel can get an answer in real time and support effectively their decision support mechanism. The cost tradeoff in this case is the occupation of exponential storage space. Data Cube operator [1] is a basic tool that materializes this strategy and will be discussed later in this thesis.

1.3. Big Data Processing

As already mentioned above, the explosion in information systems usage over the years, has led to the existence and everyday enrichment of exabytes of data, structured and unstructured, that need to be processed. Many solutions have been presented till today for processing big data. Worldwide, all major Organizations, Companies and many Governments spent time and money in order to improve their intelligence knowledge and systems for gaining a leading place in the market of data processing.

Cloud computing is a relative novelty, that moving towards big data processing, allows the exploitation of a group of computers, in order to spread problems to be solved, and make the best out of already available CPUs. Using cloud computing allowed researchers address problems with unthinkable computational needs that until recently were widowed as non processable.

Enterprises developed solutions for the efficient problem sharing for cloud computing. Google introduced Map Reduce, a model used to simplify computation parallelism and allow users to address more issues concerning big data. Apache, moved one step further materializing Hadoop, an open source platform based on Map Reduce model. The free usage of such tools, allowed users to review many problems not addressed or addressed insufficiently till today, from a new

perspective. More and more issues find their way by using the parallel processing over a cluster.

1.4. Hadoop and Bottom Up Cube Computing

As Hadoop becomes more and more popular in the area of parallel computing, many different types of problems are addressed by this method in order to obtain a result of high computational efficiency. Hadoop developers are also directing their efforts towards including features that will allow different philosophies to exploit the platform.

At this context, efforts are being made in order to transform existing algorithms for use with the Map Reduce model philosophy. The Cube computation has already been addressed by researchers, and strategies like Top-Down Cube computation have been presented.

So far the Bottom Up Cube computation strategy remains to be investigated for the best of our knowledge. Thus, the missing area that this thesis tries to cover is that of the exploitation of the Hadoop Map Reduce platform for the pruning-full bottom up computation of iceberg data cubes.

1.5. Thesis Organization

The rest of this thesis report is organized as follows; In chapter (2) a reference to issues concerning Data Analysis and Online Analytical processing (OLAP) and Data Aggregation is made. The Data Cube operator for answering analysis queries is discussed, as well as some algorithms proposed in order to handle the space and time reduction in Data Cube creation. A thorough discussion on Iceberg Cubes Computing is made together with its various computational techniques with emphasis to the Bottom Up Computing Algorithm. In Chapter (3) we discuss the Hadoop Map reduce model and its architecture, together with its Apache Hadoop implementation, which is widely contributing for handling vast

datasets during the recent years. In Chapter (4) we introduce BucDoop, a novel simple algorithm designed during this thesis research, for exploiting Hadoop's Architecture in the context of Bottom Up Computation philosophy for iceberg cube creation. In chapter (5) we present the experimental results performed over various datasets and parameter variations with BucDoop and finally in chapter (6) we present the conclusions of this thesis and propose future work.

Chapter 2 - Cubes, Iceberg Cubes & Bottom Up Computation (BUC)

2.1. General

Online Analytical Processing (OLAP) is used by enterprise analysts for performing data mining operations over collections, usually of multidimensional data, seeking for various relations and patterns in the processed information. This processing needs to work on a structured collection of data, categorized as to contain all possible involved entities. A data warehouse serves as a database-like collection for OLAP processing, providing the needed infrastructure, in order to store the structured data for analysis. Most usually, an OLTP database fulfills the role of the data origin, by providing to the data warehouses historical data through periodical snapshot instances.

OLAP systems are designed to handle easily and fast the querying over data. They contain two basic types of data: numeric data like quantities, averages and amounts, which are called *measures*, and attribute-like categorizations which are called *dimensions*.

2.2. Data Cube Operator

Trying to retrieve patterns from data, OLAP processing performs group-bys and aggregation over them. Based on this need, and keeping in mind that in the case of data mining over data warehouses we refer to multidimensional and historical data with millions of entries and tens or hundreds of dimensions, aggregating and grouping turns to be an expensive in time job. A proposal that came to solve this time issue was based on the idea of keeping group-by and aggregation results pre-computed: in [1] Grey et al. proposed the Data Cube

operator.

The cube operator proposed a generalization of the 2-dimensional cross-tab operator in the N-dimensional space. The cube treats each of the N aggregation attributes as a dimension of this N-space. The aggregate of a particular set of attribute values is a point in this space.

The creation of the group-by views of the dataset needs a scan of the data each time. Because of the costly multiple scans over the dataset needed, the various solutions proposed directional variations of simultaneously or successional computations of different group-bys. Based on the roll-up and drill-down techniques for examining the succession in group-bys, the lattice that is illustrated in Figure 1 is molded.

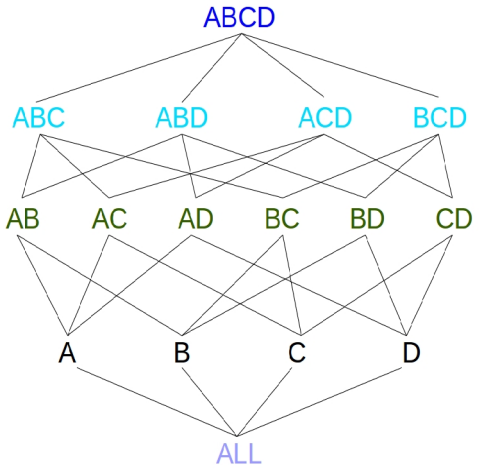

Figure 1: Data Cube Lattice

The lattice derives from a dataset with four dimensions; A, B, C and D. The various group-bys are represented by the respective attribute letters combination. The different colors present the levels with different numbers of group-bys. The lines between the group-bys represent the relation of the group-bys, which are the possible paths that a group-by can derive from, in a parent-to-child relation.

There are three major approaches developed concerning data cube computation: the *top down* computation, represented by Multi Way array aggregation [2], which utilizes shared computation and performs well on dense data sets, *bottom up* computation, represented by BUC [3], which takes advantage of Apriori Pruning and performs well on sparse data sets, and *integrated top down and bottom up* computation, represented by Star Cubing [4], which takes advantages of both and has high performance in most cases.

2.3. Iceberg Cube Problem

Iceberg cubes are similar to conventional data cubes, with the specialized condition that the groups of tuples from the source data set to be materialized into cells of the final cube, are limited to those satisfying a HAVING clause. Given a data set S, an iceberg cube Q on S is defined by a SELECT-GROUP-BY aggregation on S with an additional clause (*iceberg condition*) of the form HAVING AggOp(GB) > minSup , where:

- AggOp is an aggregation operator,

- GB is a group-by of attributes from the fact table S, and

- minSup is a value which defines the minimum support that the group-by has to surpass for being part of the cube.

2.4. Previous Work on Iceberg Cube Computation

To improve the efficiency of the aggregation methods that are the crucial feature of iceberg cube algorithms, four major issues have been considered: lattice search order, specialized data structures, segmentation, and measure computation.

2.4.1. Lattice Search Order

The way that the lattice will be traversed in order to create the cuboids is of major significance to the efficiency of the computation. The three approaches

that have been proposed concerning the search order of the lattice are bottom up, top down, and integrated bottom up and top down. Depending on the search order, various techniques have been proposed to improve efficiency; e.g., Bottom-up aggregation starts by examining aggregations for a single attribute, then pairs of attributes, etc. and thus it can exploit Apriori-like pruning technique to reduce the number of combinations of attributes considered [2],[5]. Apriori-like pruning is based on the anti-monotonic property of a pruning function. "Apriori-like" refers to the Apriori algorithm [6], where this type of pruning was exploited to compute combinations of items efficiently.

Given function f, a value V, and two partitions A and AB such that AB $\subseteq$ A, if $f(A) \leq V$, then $f(AB) \leq V$ is always true, the function f is anti-monotonic [11]. Anti-monotonic functions can be directly used in Apriori-like pruning. Aggregations like Count, Sum of positive values, Minimum, and Maximum are anti-monotonic functions, so they can be used in Apriori-like pruning. For example, for a sub-partition of the group-by A partition, $a_i \in A$, if $f(a_i)$ does not satisfy the minimum support threshold, all less-detailed group-bys containing a_i can be safely pruned; e.g., $a_i b_j$ where $b_j \in$ B a sub partition of group-by B, because they also will not satisfy the minimum support threshold.

Top-down computation examines aggregations for multiple attributes first and then it examines aggregations for fewer attributes. It shares the computation for group-bys related by a prefix relation [7]. Shared computation manipulates calculation of measure values for two or more group-bys during the same pass over the data. Because of the shared prefix between parent and child group-bys, when the child group-by is aggregated, the parent group-by can also be aggregated at the same time. Shared computation is the main idea behind top-down approaches, such as the Top-Down Computation algorithm [8]. Integrated methods have also been proposed. Star-Cubing [4] uses a top-down approach based on the global computation order for all group-bys, while locally, for each group-by, a bottom-up

approach is employed. Pipe'n Prune [9], uses an integrated method which combines features of both bottom-up and top-down aggregation, and thus takes advantage of both Apriori-like pruning and shared computation. The Multi-Tree Cubing algorithm [10] also integrates top-down and bottom-up approaches, and therefore it features both shared computation and Apriori-like pruning.

2.4.2. Specialized Data Structures

Specialized data structures like arrays and trees, are also proposed for efficient calculation [5] [7] [11]. A multi-dimensional array allows an attribute to be associated with an array dimension. The advantage of arrays is the simplicity of performing aggregation and accessing cells. However, if the data cube is sparse, much memory is wasted. To overcome this limitation, Zhao et al. (1997) [2] decompose large arrays into small chunks and load only one chunk at a time into the memory. Shao et al. (2004) [12] first decompose a data cube into subspaces according to the frequency of the values, and then for each subspace, they apply an array-based algorithm.

Various types of trees have been proposed for iceberg cube computation. Xin et al. (2003) [4] proposed star trees and Han et al. (2001) [11] proposed H-trees. The advantages of trees are that they are a compact representation of the data table and the pruning methods can be applied in the trees. The disadvantage of trees is that in the worst case, if the data cube is very dense, the algorithm will exhaust available memory.

2.4.3. Segmentation

Dividing cubes into relatively small segments that can fit into main memory has also been proposed as a solution to the iceberg cube computation. MM-cubing separates frequent and rare attribute values to form subspaces for aggregation [12]. Cross table cubing computes over separates tables in a star-schema, then aggregates the cubes locally, and finally joins the local data cubes to form the

global data cube [13].

2.4.4. Complex Measures

For complex measures that do not satisfy the monotonic or anti-monotonic properties, methodologies for computation has been proposed also. Han et al. (2001) Proposed top-k average as an approximation for the average measure [11]. Since the top-k average is monotonic, it can be used for Apriori-like pruning. Wang et al. proposed combining the ideas, of "divide-and-conquer" and "approximate" in order not be dependent on the specific form of the aggregate function [14]. Thus an approximation that is either monotonic or anti-monotonic can be used for Apriori-like pruning.

2.5. Bottom Up Computation of Iceberg Cube

BUC [3] uses a bottom-up computation where cuboids with fewer dimensions are parents of those with more dimensions. BUC starts by reading the first dimension and partitioning it based on its distinct values. For each partition, it recursively computes the group-bys with the remaining dimensions. The computation along a partition terminates if its count is less than the iceberg condition. This way, bottom-up computation order allows the Apriori-like pruning which is very efficient when the dataset is sparse because of reducing lots of unnecessary computation. Figure 2 shows a BUC processing tree for four attributes.

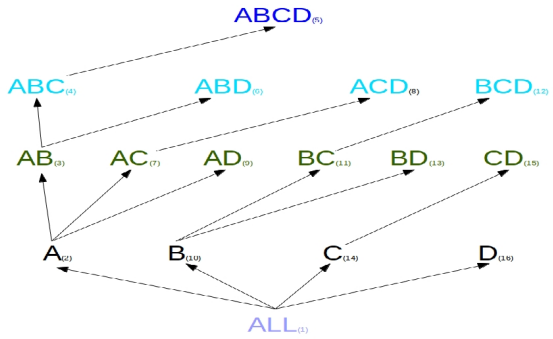

Figure 2: The Bottom Up Computation Cube Lattice

The small numbers beside the group-bys indicate the processing order. BUC first divides the data in partitions based on attribute A and checks the minimum support condition for the first partition a_1. If the condition is not satisfied, BUC prunes any partitions starting with a_1, which include a_1 b_i ($b_i \in B$), a_1 c_i ($c_i \in C$), and a_1 d_i ($d_i \in D$), by stopping recursion. If the condition for a_1 is satisfied, BUC outputs a_1, then recursively moves on to the next group-by, which is AB. It divides the data of partition a_1, based on attribute B, to compute new partitions a_1 b_i, such that $b_i \in B$. If a_1 b_i satisfies the minimum support condition, then the next group-by, ABC, is examined, and so forth. Once the processing for a_1 is complete, BUC continues checking the remaining partitions of A. At this point, all group-bys starting from attribute A have been computed. Next, BUC starts from the remaining attributes B, C, and D and repeats the process for each.

BUC is a divide and conquer strategy, and partitioning is its major cost. BUC can be used to compute either a full data cube or an iceberg cube. Due to its pruning power, BUC works especially well at computing iceberg cubes for sparse database tables as mentioned earlier. Figure 3 presents the original algorithm

Procedure: **BottomUpCube(input,dim)**

Inputs

input::	The relation to aggregate
Dim::	The starting dimension for this iteration

Globals

constant numDims::	The total number of dimensions
constant cardinality [numDims]::	The cardinality of each dimension
minSup::	The minimum number of tuples in a partition for it to be output
outputRec::	The current output record
dataCount[numDims]::	Stores the size of each partition dataCount[i] is a list of integers of size cardinality[i]

Outputs

output::	One record that is the aggregation of input. Recursively, outputs CUBE(dim,…, numDims) on input (with minimum support).

Method

```
1:    Aggregate(input);
2:    if input.count()==1 then
          WriteAncestors(input[0] ,dim); return;
3:    write outputRec;
4:    for d = dim;  d < numDims ; d++   do
5:        let C = cardinality[d];
6:        Partition(input,d,C,dataCount[d]);
7:        let k=0
8:        for i=0; i <C ; i++     do
9:            let c= dataCount[d] [i]
10:           if  c >= minSup then
11:               outputRec.dim[d]=input[k].dim[d];
12:               BottomUpCube(input[k…k+c],d+1);
13:           end if
14:           k+= c;
15:       end for
16:       outputRec.dim[d]= ALL;
17:   end for
```

Figure 3: The original BUC Algorithm [3]

BUC's first step upon each recursive call is to aggregate the entire input (Line 1) and write the result (Line 3). In Line 2 it uses an optimization for skipping the further recursive computation of one Line partitions.

```
1:    Aggregate(input);     //  Places result in outputRec
2:    if    input.count()==1   then //Optimization
```

```
WriteAncestors(input[0] ,dim); return;
3:      write outputRec;
```

Next, it partitions the input for each dimension from the current dimension to the total number of dimensions, according to the specific dimension cardinality (Lines 4-6).

```
4:      for d = dim;  d < numDims ; d++   do
5:              let C = cardinality[d];
6:              Partition(input,d,C,dataCount[d]);
```

On return from Partition(), dataCount contains the number of records for each distinct value of the d-th dimension.

Accordingly in Line 8 BUC iterates through the partitions

```
7:              let k=0
8:              for i=0; i <C ; i++  do              // For each partition
```

If the partition meets minimum support, the partition becomes the input relation in the next recursive call to BUC, which thus computes the Iceberg cube on the partition for dimensions d+1 to the total number of dimensions. Upon return from the recursive call, BUC continues with the next partition of dimension d. Once all the partitions are processed, the algorithm repeats the whole process for the next dimension.

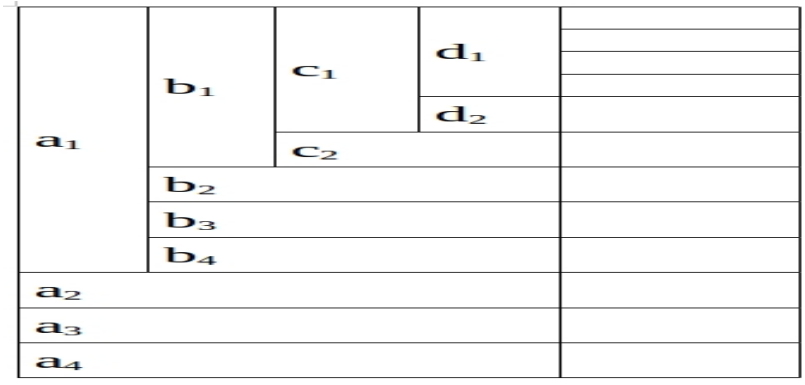

Figure 4: BUC Partitioning [3]

Figure 4 illustrates how the input is partitioned during the first four calls of BUC, in a hypothetical scenario where the A dimension has four distinct values (cardinality=4), the B dimension has also four distinct values (cardinality=4), the C dimension has two distinct values (cardinality=2) and the D dimension has also two distinct values (cardinality=2).

First BUC produces the ALL group-by. Next, it partitions on dimension A, producing partitions a_1 to a_4, and then it iterates on partition a_1. The a_1 partition is aggregated and produces a single tuple for the A group-by. Next, it sorts and partitions the a_1 partition based on dimension B. It iterates on the $\langle a_1, b_1 \rangle$ partition and writes an $\langle a_1, b_1 \rangle$ tuple for the AB group-by. Similarly for $\langle a_1, b_1, c_1 \rangle$ and then $\langle a_1, b_1, c_1, d_1 \rangle$ but this time it does not enter the loop at Line 4. Instead it simply returns only to iterate again on the $\langle \alpha_1, b_1, c_1, d_2 \rangle$ partition. BUC then returns twice and then iterates on the $\langle \alpha_1, b_1, c_2 \rangle$ partition. When this is complete, it partitions the $\langle \alpha_1, b_1 \rangle$ partition on D to produce the $\langle \alpha_1, b_1, D \rangle$ aggregates. Once the $\langle \alpha_1, b_1 \rangle$ partition is completely processed, BUC proceeds to $\langle \alpha_1, b_2 \rangle$.

When a small partition is found, instead of writing for all of the group-bys, BUC simply skips the partition (Line 10) and does not consider any of the partition's ancestors.

```
9:      let c= dataCount[d][i]
10:     if    c >= minsup then    // The BUC stops here
11:           outputRec.dim[d]= input[k].dim[d];
12:           BottomUpCube(input[k…k+c],d+1);
13:     end if
14:     k+= c;
```

We notice here that none of the ancestors can have minimum support so

the pruning can be implemented.

2.6. Considerations on BUC

BUC uses continuous partitioning and sorting. As discussed in [3] if the input does not fit in main memory, the data must be partitioned to disk. A proposed solution was to use a BUC- external implementation initially, and switch to BUC-internal as soon as the partition fits in main memory, since thereafter the further sub-partitioning will provide input that will also fit in memory. Of course, at each successional computation on the next distinct value partition for the same dimension, BUC- external may be needed again, if the new partition does not fit in memory.

Beyer et al, also discussed that the performance of BUC is sensitive to the ordering of the dimensions since the goal of BUC is to prune as early as possible in order to skip unnecessary computations. Thus, for best performance, the most discriminating dimensions based on cardinality and uniformity should be used first during computation. So dimensions with higher cardinality and uniformity should be considered first in order to achieve the most out of the pruning-based functionality of BUC.

Authors of BUC also considered the disadvantage of BUC when having to deal with big and dense inputs which will encounter a very small pruning because of the density and thus a late insertion into memory.

Those considerations will be dealt in the Hadoop-based algorithm where the advantages of BUC can be also exploited, while probable disadvantages are outdistanced by the native parallelization of big data of the Map Reduce model.

Chapter 3 - Map Reduce

3.1. General

The increase in the need for processing vast amounts of data for many real world tasks brought up a requirement for a relatively simple model that would exploit existing ordinary resources for the job. *MapReduce* is such a programming model introduced by Google [15].

The basic idea behind MapReduce model is to use the computational and storage strength of a big number of workstations in parallel, while declutching users from the various details of parallelization, fault tolerance, data distribution and load balancing, that are needed for managing a workload in such a group.

Through this model, the user avoids being involved with parallelization which is handled by the framework of the model, and solely has to define two functions - Map and Reduce. Those two functions will be in fact the core of the user defined job that needs to be performed over the dataset.

The MapReduce framework handles the distributed resources, runs the various tasks in parallel, manages all communications and data transfers between the various parts of the system, and provides redundancy and fault tolerance.

The MapReduce model idea is based on a simple series of actions that synthesize the Map Reduce job. To begin with, Input is read from the file system and is accordingly converted to appropriate Key-Value pairs in a way defined by the user and that will allow their further processing. The Map function processes each input pair and outputs the result (according with the functionality given to the Map function by the user) as zero, one or more intermediate Key-Value pairs. For each distinct intermediate Key, the Reduce function processes all Key-Value pairs with the same Key, and returns a number of final Key-Value pairs (according with

the functionality given to the Reduce function by the user). Finally, the output of the Reduce function is written to the file system as the final Key-Value pairs.

The MapReduce model architecture using the steps described above, acts as a means for guiding data through their path to accomplishing their processing, and allows the user to specify the actual processing that needs to take place.

In every step, any separate Key-Value pair is processed independently of any other data, allowing the processing to take place in any machine that is aware of the user defined computation code that has to take place over it. This computation is the same for any other key value pairs. That way, the different Key-Value pairs, that consist the transformation of the initial dataset, can be distributed amongst a group of machines for further processing. The key idea here is that the user needs to define the way that the data will be transformed into Key-Value pairs in each step of the way, in order to achieve the computation outcome, after being processed by the user defined Map and Reduce functions.

3.2. Hadoop

Hadoop is an open-source framework based on Google's MapReduce model, written in Java and developed by the Apache Foundation.

The execution of a MapReduce program in Hadoop is based on the actions described earlier for a typical Map Reduce job. As thoroughly described in [16], the user uploads input data to the *Hadoop Distributed File System (HDFS)*, which in turn distributes and stores it on the computing nodes for further processing. The input will be split into chunks and each chunk will be processed by a map task. The results will be partitioned into distinct sets, which will be sorted and passed to a reduce task.

The two core components of Apache Hadoop are the Hadoop Distributed File System (HDFS) which covers the sector of data storage handling and the Map Reduce which covers the sector of processing. Both HDFS and MapReduce are

designed in order to be co-deployed and perform as a single cluster and thus provides the ability to move computation to the data.

3.2.1. Hadoop MapReduce Model Infrastructure

Being based on a parallel architecture, the model needs a number of machines to work on. These machines are called nodes. Single machine usage is possible but the advantages of the data distribution and parallel process would vanish, thus such an arrangement would serve only for educational purposes. Nodes can be commodity computers with no special characteristics required, either on the same network or in a wider distributed administrative architecture.

This interconnected group of nodes consist a cluster, which has two types of nodes operating in a master-worker pattern in the abstract sector of HDFS: The *namenode* (the master) and a number of *datanodes* (workers). The namenode manages the file system namespace. It maintains the file system tree and the metadata for all the files and directories in the tree. This information is stored on the *namenode's* local disk. Datanodes are the workers of the platform. They store and retrieve blocks as needed by the framework and are the ones that will perform the computational load as well, in order to benefit from the data locality and avoid the cost of data transfer.

Concerning the abstract sector of Computational processing, there are two types of nodes that allow the job execution process: a *jobtracker* and a number of *tasktrackers*. The jobtracker coordinates all the jobs running on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker. The core components of the HDFS architecture is shown in Figure 5.

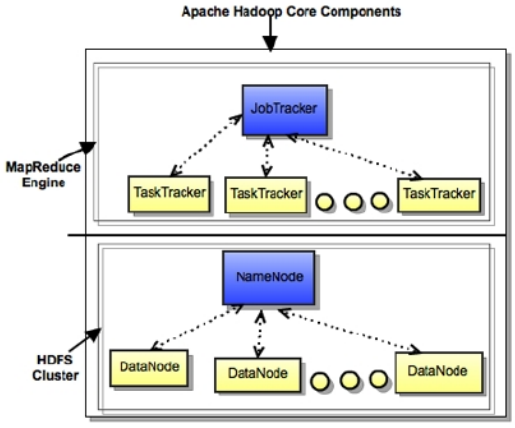

Figure 5: The Hadoop Core Components Architecture [18]

3.2.2. Hadoop MapReduce Data Flow

For every MapReduce job the dataset files are split in chunks from 16 to 128 MB and loaded to the distributed file system. The user initiates the job by specifying the MapReduce program to be executed, and the input path of the dataset in HDFS as well as the output path of the job. The master node sends a copy of the program to every computing node and starts executing the job. The mapping tasks are assigned on many or all of the nodes in the cluster. Each mapper loads the set of files local to the specific machine and processes them, fetching more data from other nodes if necessary and possible as dictated by the master node.

When the mapping phase has completed, the intermediate Key-Value pairs that were output of the map phase must be exchanged between machines in order to group all values with the same key to a single machine which will act later as a reducer. The reduce tasks are spread across the same nodes in the cluster as the mappers, and process all the pairs assigned to them. When the Reduce phase is finished its output is being written to HDFS. The Hadoop MapReduce Data Flow is shown in Figure 6.

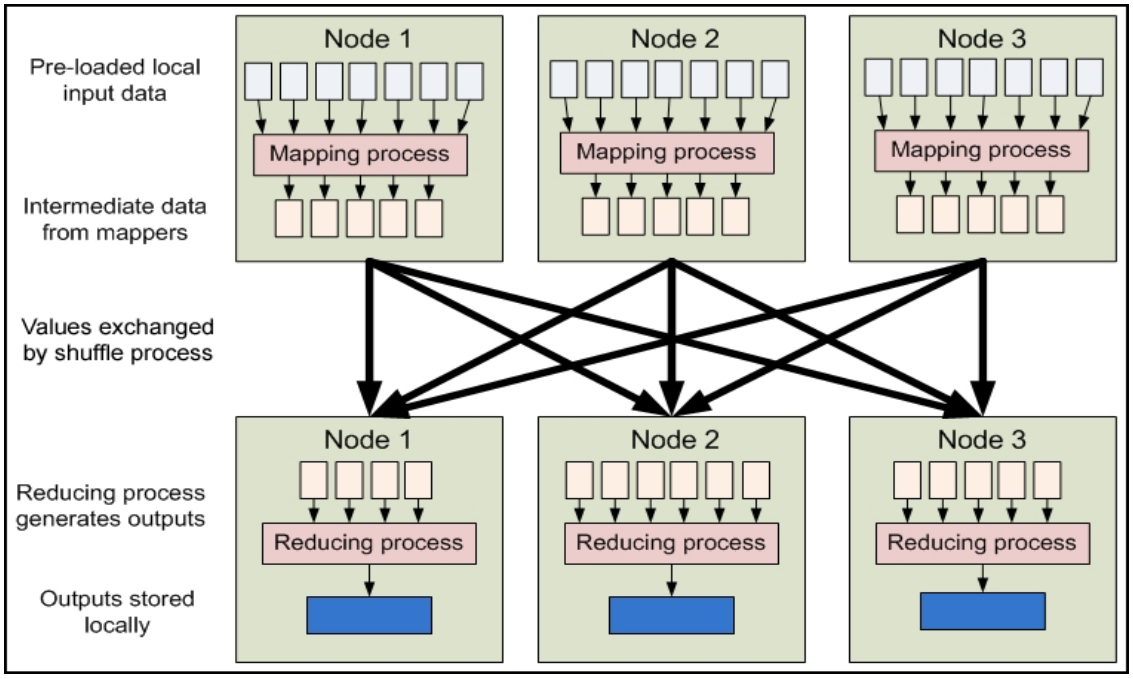

Figure 6: Hadoop MapReduce Data Flow [19]

3.2.3. Hadoop MapReduce Work Flow

As already mentioned earlier, one of the big benefits of using Hadoop Map Reduce to process big data in a cluster is that the user will not have to be involved in any way with cluster setup and parallelization issues as well as fault tolerance. The user will have to define the computation that will take place in the Map and/or in the Reduce phases, as well as the way that the input should be read for the sake of the computation needed.

To this end the user will have to deal with the following entities:

Input Handling

Input: The input files reside in HDFS after user uploading. They will be used as input to the Map Reduce job. The size of the files usually is very big and thus initiating the need to invoke Hadoop.

InputFormat: Refers to the way that the input files will be split and read in order to be useful to the job computation. The InputFormat will read all files in a directory and divide them into one or more InputSplits. Hadoop provides various inputFormat classes for handling different kinds of data such as TextInputFormat, SequenceFileInputFormat etc. Some details about inputFormats provided are presented in Table 1. The default InputFormat is the *TextInputFormat*. This InputFormat treats each line of input file as a separate record.

InputFormat	Description	Key	Value
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into key, val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

Table 1: InputFormats Provided by Hadoop

InputSplits: An InputSplit is a chunk containing a portion of the dataset that will be handled by a single map task in the MapReduce program. As a result, a MapReduce program includes several tasks. Map tasks may involve reading a whole file or only a part of a file. By default, the FileInputFormat break a file up into 64 MB chunks which is the size of blocks in HDFS for the sake of minimizing seeking cost. This way the InputFormat defines the list of tasks that consist the mapping phase since each map task is assigned to a single input split. The tasks are

assigned to the nodes in the file system where the input file chunks are physically resident.

RecordReader: TheRecordReader is responsible for loading the data from file system by converting them into key, value pairs suitable for reading by the Mapper.

Processing Data

Mapper: The Mapper is the main user defined function, which is responsible for transforming the Key/Value pairs that have been read with the above mentioned steps with the use of the computational functionality programmed by the user. Together with the functionality that the user will define in the reducer, the map tasks will contribute their share in the data processing. Depending on the computational needs of each MapReduce job over some data, the mapper could involve no actual computational functionality. For each input Key/Value pair, a map task will be called, and after the proper process it will produce an intermediate Key/Value pair that will be the map output for passing to the reduce phase.

Combiner: Combiner is a processing step which can be optionally used for minimizing data transfer between Map and Reduce phases. The Combiner runs locally on any machine that performed a mapper phase task and receive as input all data emitted by the Mapper instances on the given node. Given the problem that allows for efficient usage of a suitable combiner, the output from the Combiner will be significantly smaller than the output of numerous map outputs before being combined. This output is sent to the Reducers, instead of the output from the Mappers.

Reduce: The Reducer function is responsible for creating a reduce task for each pair consisted of a Key and a list of values associated with that Key. The input for the reduce task is a result of the grouping performed over all the Values coming from many different mappers output pairs, which are associated with the

specific Key assigned to the reduce task. It is the second main user-defined function, which complements the processing over the data.

Data exchange during Processing

Partition & Shuffle: The Partition function is responsible to gather all Values associated with a Key, that are coming from different mappers, and transfer them to the node that is assigned for the reduce of the associated Key. The process of moving map outputs to the reducers is called *shuffling*.

Sort: The set of intermediate keys on a node is automatically sorted by Hadoop framework before they are presented to the Reducer.

Output handling

OutputFormat: The Key/Value that result by the reduce processing are written to the output in a way that is defined by the *OutputFormat*. Each Reducer writes a separate file in a common output directory. Details about provided OutputFormats are given in Table 2. The default *OutputFormat* is TextOutputFormat, which writes output pairs on individual lines of a text file.

OutputFormat:	Description
TextOutputFormat	Default; writes lines in "key /value" form
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Disregards its inputs

Table 2: OutputFormats Provided by Hadoop

RecordWriter: The record Writer is the means through which the output is actually written to the output file system. The Components of a Map Reduce job are illustrated in Figure 7.

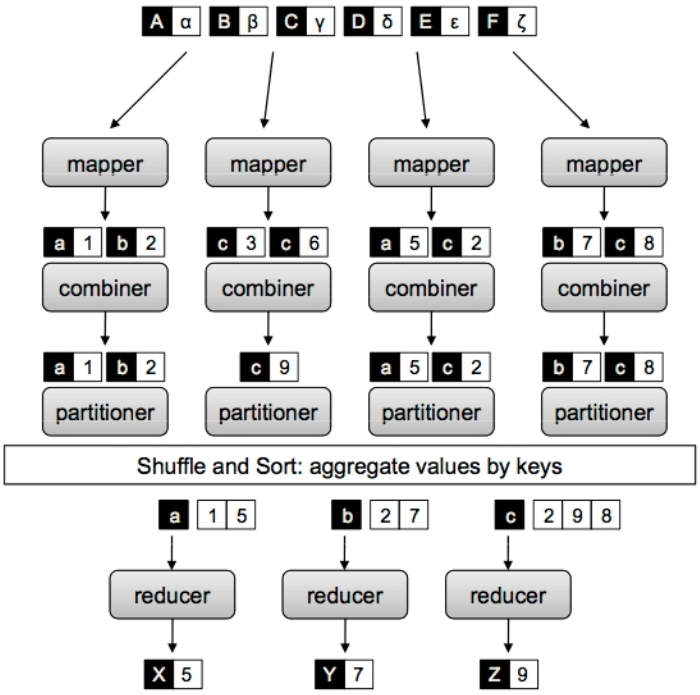

Figure 7: Hadoop MapReduce Components flow [20]

Chapter 4 - Bottom Up Computation of Iceberg Cubes with Hadoop

4.1. General

In this chapter we will present BucDoop, an algorithmic approach for creating the iceberg cube with Hadoop. For that purpose, we will analyze the architecture of the iceberg cube Bottom Up Computation, and adjust it to the Map Reduce philosophy.

4.2. Work Phases

We will start by recalling the basic phases that BUC uses, which are:

-partitioning the input relation starting with single attribute group-bys,

-pruning partitions that are not covering the iceberg condition and

-continue partitioning supported group-bys bottom-up-wise.

Considering that the input relation is very big, partitioning and sorting the dataset according to the successional dimensions as performed in the original BUC algorithm is not feasible. Therefore the new algorithm handles the input dataset initially unsorted.

4.3. Adding Hadoop Philosophy

For managing the partitioning phase at each stage, the built in feature of grouping by based on the Key of Key/Value pairs in Hadoop, was used to achieve this kind of functionality. The specific internal feature of the Map Reduce model, allows for the partitioning of huge datasets during the first pass, with no limitations having to do with memory issues, since every single record read can be converted in an appropriate Key/Value pair, and simply rely on the Hadoop framework to

perform the proper partitioning by its internal hash based partitioning mechanism, and forward the new partition to one of the reducers. The size of each new partition, which can normally exceed memory size when dealing with big data, will also be unproblematic since the Hadoop Distributed File System will manage its storage and allow the reducer to handle it smoothly.

Concerning the iceberg threshold support count of the partitions' tuples, this functionality was laid on the reducer side, which groups the assigned partition and performs the counting. The reducer needs to scan the partition twice; once for counting partition support and performing aggregation at the same time, and a second time for passing succeeding tuples to the next processing phase.

For maintaining the partitions scans to one, we exploit the initial scan for producing all possible next phase group bys and forward them to the appropriate partition/reducer. As a result, each read will produce D-G+1 Key/Value pairs, where D is the dimensionality of the dataset and G is the position of the last grouped by attribute in the processed input record during the last executed job.

In a hypothetical example, we have D=3 with dimensions A, B, C and one measure M.

Let <a1, b1, c1, m1> be the first input tuple, in the mapper of the first Map Reduce job. The job is the first job executed, which means that the position of the previous job processed group-by is G=0. So the mapper will create 3-0+1=4 new records with the following group-bys:

Group By A: <a1, all, all, m1>

Group By B: <b1, all, m1>

Group By C: <c1, m1>

Group By All: <all, m1>

As a following step, the mapper needs to pass the created group-bys to the reducer for proceeding to the phase of assembling the various partitions with records coming from other mappers. In order to provide the ability to the reducers, after counting the support of the assembled partition, to reform the initial information and create the next phase group-bys, the mapper will form the Key/Value pairs in a pattern that will allow the minimal exchange of only useful data between mappers and reducers.

We notice in the above presented group-bys, that the information which need to be transferred to the reducers for further computation does not include all dimensions; e.g. for passing the values for the group-by B, we do not need to pass the values for A. therefore we skip the value for A but we transfer the values following B, because they will be needed in the next group-by processing job for dimension C. To that end the mapper needs to create a key representing the group-by B and a value representing the next dimensions that need to be transferred to the next jobs, in case that the group-by B is found to be frequent in this jobs reduce phase, as well as the measure which will be used in the aggregation phase. We also notice that in order to obtain the minimum possible data transfer, we need to avoid transferring intermediate dimension information which are not participating in the processed group-by; for example while processing group-by AC, we should avoid sending information concerning intermediate dimension B.

In our case we choose to reduce the amount of data transfer, which will result a great benefit for every non-useful byte skipped when dealing with millions of records. We thus need to use an efficient solution in order to avoid confusing keys' grouping for different group-bys with the same values; e.g. we should distinguish the key representing the group-by AB with participating integer values of A=1 and B=1, from the key representing the group-by AC with participating integer values of A=1 and C=1, and make sure that the AB group-by <key, value> pair will end up in a different reduce task from the AC group-by <key, value> pair

even if they both have the same participating integers <1,1> in their key.

This functionality was obtained using a customized type of complex keys, which include an integer array for holding the participating dimension integers, accompanied by a byte array used as a bitmap of flags denoting the participating group-bys. The byte array capacity is only as big as it needs to be, in order to hold the bitmap flags for the number of the data set dimensions; e.g. for 6 dimensions and a single dimension group-by representation only one byte will be needed for the bitmap and only one integer for the participating dimension, instead of a full sequence of 6 integers and their delimiters or a respective sequence of ASCII characters that would be needed otherwise.

Finally the next dimensions that are not participating in the currently processed group-by as well as the measure, which are information that cannot be skipped, are transferred to the next phases contained in a customized type of complex value, holding an integer array and a float point value respectively.

For the smooth handling of the composite keys and values transfer between nodes, in order to avoid plural serialization/deserializations needed during the jobs, the implementation that we used initially reads an ASCII - character dataset during the first map reduce job, and accordingly all other data exchanges are performed in binary format.

For our example case the mapper will generate the following after receiving the first input tuple <a1, b1, c1, m1>:

Group By A:	Key1.1=<1, 0, 0><a1>	Value1.1=<b1, c1><m1>
Group By B:	Key1.2=<0, 1, 0><b1>	Value1.2=<c1><m1>
Group By C:	Key1.3<0, 0, 1><c1>	Value1.3=<><m1>
Group By ALL:	Key1.4<0, 0, 0><>	Value1.4=<><m1>

Let us assume in our example that we utilize a Map Reduce cluster with two mappers and two reducers. Let us also assume that the first input tuple that we examined already was the input split that was stored in the node of the first mapper, and thus was assigned for process to that mapper. Finally let us assume that the iceberg condition for minimum support is minSup=2.

Now, let <a2, b1, c2, m2> be the first input tuple of the input split that is stored in the node of the second mapper, and so it will be processed by the second mapper. The second mapper will generate the following:

Group By A:	Key2.1=<1, 0, 0><a2>	Value2.1=<b1, c2><m2>
Group By B:	Key2.2=<0, 1, 0><b1>	Value2.2=<c2><m2>
Group By C:	Key2.3<0, 0, 1><c2>	Value2.3=<><m2>
Group By ALL:	Key2.4<0, 0, 0><>	Value2.4=<><m2>

After the completion of the mapping phase, the partitioning phase will assign the intermediate Key/Value pairs to the two existing reducers. During this phase, among the other exchange of data, pairs <Key1.2/Value1.2> and <Key2.2/Value2.2> will end in the same reducer node (let that be reducer 1) and in the same reduce task for processing. The reducer will count the partition size and check if the minimum support condition is satisfied, while at the same time it will aggregate the measures. In the paradigm case the support is 2 and satisfies the condition.

Reducer 1 will then append the succeeding croup-by (ALL,b1,ALL,m1+m2) in the final iceberg cube which is stored in HDFS, and forward the partition with the two Key/Value pairs to the job output in HDFS, which will act as input for the next bottom-up wise group-bys; AB, AC and BC.

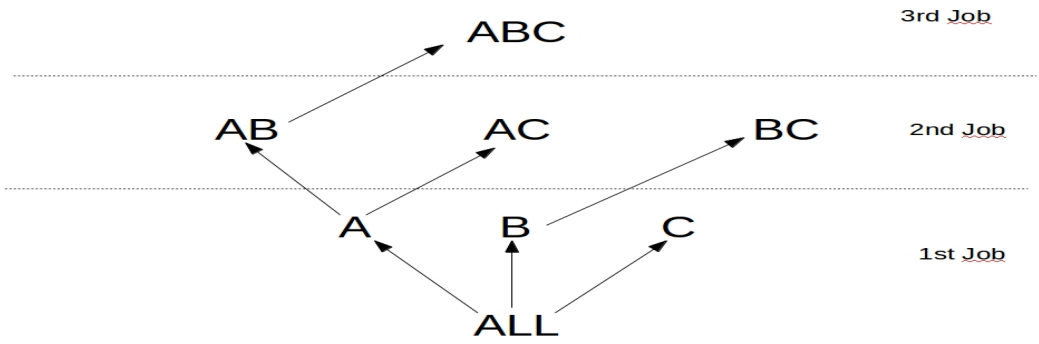

Figure 8: BucDoop Job Flow For The Iceberg Cube Computation

This way, the Hadoop framework will have to initiate in maximum D successive Map Reduce jobs, for computing all possible group-bys. Figure 8 illustrates the way that the cube lattice is traversed by the Map/Reduce jobs, in order to create the iceberg cube.

4.4. BucDoop Algorithms

The algorithm that is used in the Jobtracker set up is presented in Figure 9. This algorithm controls the work flow of Map Reduce job iterations that need to be scheduled for the iceberg cube to be created. With the proposed architecture, each Mar Reduce job uses as input the sufficiently supported group-bys that were output during the previous phase.

The proposed solution uses two different algorithms for the Mappers; Mapper_A for converting raw input records into Key/Value pairs and Mapper_B which converts Key/Value pairs of the previews job group-by into Key/Value pairs for the exactly next less detailed group-by for processing (e.g. from group-by A create group-by AB, AC, AD etc.).

Procedure: BucDoopJob(input)

Inputs

input:: The relation to aggregate

Outputs

output:: The succeeding GroupBys of the initial raw Fact Table with their respective aggregates (iceberg cube)

Method

```
1:  Run MapReduce Job
2:      Run Mapper_A
3:      Run Reducer
4:  End MapReduce job
5:  input ← output;
6:  delete(output);
7:  While (!input.isEmpty)
8:      Run MapReduce Job
9:          Run Mapper_B
10:         Run Reducer
11:     end MapReduce job
12:     input ← output;
13:     delete(output);
14: end While
```

Figure 9: Algorithm Used in The Jobtracker Setup

The algorithm gets as input, the relation to be processed. In Lines 1-4 it initiates a Map Reduce job, where the algorithm for mapper_A is used, for reading

the raw data and converting them to Key/Value pairs. After the completion of the Map phase, the grouped-by output partitions are sent to the Reduce phase, where they will be counted and aggregated. The sufficient partitions will be outputted for the next phase processing and their aggregate will be appended to the iceberg cube.

In Lines 5 and 6, the algorithm is defining the previous stage output as input for the next stage. This way, it exploits the HDFS namespace administration efficiency of Hadoop which is being performed by the Namenode, and does not need to move any data.

In lines 7-14, the algorithm iterates over successional Map Reduce jobs execution, where the output of the previous job is used to produce the exactly next less detailed group-by for processing. The algorithm that is used for the first Mapper is presented in Figure 10.

Procedure: Mapper_A (input)	
Inputs	
input::	A chunk of the relation to aggregate (raw fact table tuples) residing in HDFS
Globals	
constant numDims:	The total number of dimensions
Outputs	
output::	numDims+1 total output records with format <Key,Value> for each input record

```
Method
1:      Prefix ← "";
2:      Suffix ← input.getAllInputAttributes;
3:      Mes ← input.getInputMeasure;
4:      for i=0 to numDims with step 1
5:          Prefix ← Suffix.pollFirstElement;
6:          Key ← Prefix;
7:          Value ← new Value(Suffix,Mes)
8:          Output.Write(<Key,Value>)
9:      endfor
```

Figure 10: Algorithm Used for the Mapper of the 1st Map Reduce Job

In Lines 1-3, the algorithm initializes variables. In Line 2 it assigns the input record attributes to the variable Suffix and in Line 3 it assigns the input record measure to the variable Mes. In Lines 4-9, it iterates over the dimension space in order to produce all possible single group-bys; it polls first suffix element and assign it to the variable Prefix which represents the group-by that will be processed in the Reducer. In Lines 6-8, it builds the Key/Value pairs and writes them to the HDFS output.

It is obvious that the algorithm is "multiplying" the size of the input by D times. This is not a downside, since the examination of the all the possible group-bys that are available, after the completion of processing of the previous dimension group-bys and the "removal" of the previous dimension from our calculations, would be necessary in any case. The original BUC algorithm, after finishing with the current processed dimension group-bys, "removes" the current dimension and

continues processing the group-bys of the subsequent dimensions. In our proposed Map Reduce solution, the input is scanned once, and all possible group-bys of that phase are generated. This way we minimize the Map Reduce jobs and their respective input scans. The Mapper_B algorithm (Figure 11) is very similar in logic to that of Mapper_A, with the difference that in includes functionality for converting input Key/Value pairs into next step Key/Value pairs.

Procedure: Mapper_B (input)	
inputs	
input::	Chunk of Records with format <Key,Value>
currentDim::	The dimension which has been grouped-by in the previous job
Globals	
constant numDims::	The total number of dimensions
Outputs	
output::	numDims-currentDim+1 total output records with format <Key,Value> for each input <Key,Value> pair
Method	
1: Prefix ← input.getKey;	
2: Suffix ← input.getValue.getSuffix;	

```
3:      Mes ← input.getValue.getMeasure;
4:      for i= currentDim to (numDims-1) with step 1
5:              Prefix ← Prefix.add(Suffix.pollFirstElement);
6:              Key ← Prefix;
7:              Value ← new Value(Suffix,Mes)
8:              Outpt.Write(Key,Value)
9:      end for
```

Figure 11: Algorithm Used for the Mapper of the 2nd and Rest Jobs

The algorithm assigns values in variables in Lines 1-3, and creates new group-bys in Lines 4-9. Instead of creating a new prefix as Mapper_A does, it extracts from the input the old prefix, and creates the new group-by, adding a new group-by dimension member, which will be one of the remaining dimensions of the input which are iterated in Lines 4-9.

For reason of clarification here we need to recall that in practice, since what is imported and exported during the phases, representing the dimensions, is a group of integers (as commonly used in practice, mapping of the dimension members to integers is necessary in order to improve storage efficiency), the use of the bitmap mentioned earlier in this chapter is the tool which helps us to distinguish the group-by that a set of integers refers to; e.g., <1,1,0><5,7> and <1,0,1><5,7> have the same participating integers (5 and 7) but their preamble clarifies that the first refers to group-by AB and the second refers the group-by AC.

Concerning the Reduce phase, a single reduce algorithm is used for all jobs. The algorithm that is used for the Reducers is presented in Figure 12.

```
Procedure: Reducer
inputs
```

input::	Key, Iterator over a list of Values
icebergFile::	The HDFS file that stores the created cube

Globals

constant numDims::	The total number of dimensions
constant minSup::	The minimum number of tuples in a partition for it to be output

Outputs

Output::	One output record with format <Key,Value> for each record in input list
outputGroupBy::	One output record with format <GroupBy,Aggregate> if the input groupBy count exceeded minSup condition during reduce job

Method

```
1:    Aggregate ← 0
2:    Count ← 0
3:    While (input.hasNext)
4:          Count++;
5:          Aggregate += input.getValue.getMesure;
6:    end While
7:    If (Count>=minSup)
```

```
8:              outputGroupBy ← concat(Key,Aggregate);
9:              IcebergFile.append(outputGroupBy);
10:             While (input.hasNext)
11                      Output.Write(input.next);
12:             end While
13:     end If
```

Figure 12: Algorithm Used for the Reducer of All Jobs

The algorithm initializes variable values in Lines 1-2. In Lines 3-6 it iterates over the input values and counts them, while in the same time it aggregates the measures. In Line 7, the iceberg condition satisfaction is checked; if the condition is not satisfied, the reduce task will not provide further output neither for the iceberg cube directly, nor for the next processing jobs. In that point, the pruning of the insufficiently supported partitions is performed. If the iceberg condition is satisfied, the algorithm will produce a record for the final iceberg cube (Line 8) and append it to the cube (Line 9). It will also forward the sufficiently supported tuples of this partition to the job output (Lines 10-12), in order to be used by the next job, and contribute to the calculation of the less detailed group-bys.

Chapter 5 - Experimental Evaluation

5.1. General

For the experimental evaluation of the proposed algorithms, an implementation in Java was performed. Java SE 6 was used and NetBeans IDE 7.1 platform for the actual coding. The version of Hadoop Map-Reduce library used was 0.20.2. The initial evaluation of the code was performed in a Hadoop pseudo-distributed mode cluster with one machine. The rest of the experimental evaluation was conducted in the Technical University of Crete Softnet lab Hadoop cluster.

5.2. Cluster Configuration

The Hadoop configuration of the cluster is based on version 1.0.3. The cluster consists of 17 nodes, one configured as master namenode and 16 slave nodes. The node machines are of two different types. The first type, master node and 12 slave nodes, runs Ubuntu Linux operation system release 9.0.4 with Intel Xeon 4 core CPU, model X3323, at 2.50GHz and 4 GB of RAM. The second type, 4 slave nodes, runs Ubuntu Linux operation system release 10.04.4 with Intel Xeon 8 core CPU, model X3440, at 2.53GHz and 4 GB of RAM. All machines' hard drives have 500 GB capacity with read and write speed at 7200 rpm.

The Hadoop configuration allows each map or reduce task to occupy one core of CPU from the machine and 1 GB of RAM. The aforementioned configuration allows a machine to run 4 parallel tasks at any given time allowing a Map-Reduce job to have up to 64 parallel tasks for all 16 nodes. The HDFS has capacity of 7,43 TB.

5.3. Evaluation Data

In order to evaluate the performance of the algorithms, testing in various data sets was performed. To this end, data sets had to be created to fulfill the needs for different parameters' evaluation. The data sets created were ASCII character text files divided in lines, and each line represented a tuple. The tuples included integer maps for the various dimension members, separated with tab delimiters. For the evaluation, one float value measure was used. The data distribution in the datasets for the main portion of the experiments was uniform. A small number of experiments were conducted with data not following a uniform distribution and very big cardinalities, in order to check the behavior of the algorithm with sparse datasets.

5.4. Evaluation Parameters

Data cube computation is by default a space consuming operation. Techniques for drastically minimizing space consumption have been proposed by researchers, e.g. Dwarf presented with [17]. Still one of the main key factors concerning data cube computation, which limits enterprise analysts' job, is the time consumption. Time consumption was the key factor examined during this experimental evaluation of the proposed algorithmic solution.

The performance of the algorithm over the cardinality variation as well as the minimum support variation was performed initially. Next, the variation of the used reducers' number was checked, and finally, the scalability of the algorithm was evaluated over increasing size data sets by increasing the number of dimensions or the number of the tuples.

Since the proposed algorithm architecture includes successional Map Reduce jobs in order to compute the iceberg cube, where at each stage/job the size of the input differs greatly according to the output of the previous job, the number of

mapper nodes used was not forced, and was left up to Hadoop to use appropriate number of mappers each time. At each experimentation phase, the evaluated parameter was varied while all other job parameters were kept fixed.

5.5. Experiments Discussion and Results

As a first step for analysis, a small data set size was used, in order to determine the effect of the cardinality change over time consumption. Besides the small data set of 100K tuples that was processed, all the other fixed parameters used inserted a relative small time consuming factor; iceberg condition of minimum support used was 10 and 20 tuples in two different series of experiments, the configuration used almost all the reducers of the cluster for the job (r=15) and the dimensionality used was of 5 dimensions. The cardinalities used varied in a range of 20 to 30, for the purpose of representing a typical cardinalities' range close to real life data. The illustrated in Figure 13 results formed greater time consumption for smaller cardinalities. The result was expected since data population of each distinct value increases for small cardinalities and the partitions that have no sufficient support are less; thus less pruning is achieved for small cardinalities and the calculations' time increases. A pick point was recognized in cardinality of 30 distinct values. The variation of time consumed for minimum support of 20 tuples was very small, procuring that support equal or bigger than 20 would provide no interest for our further experimentation. The next group of experiments helped us clarify the correctness of this second observation.

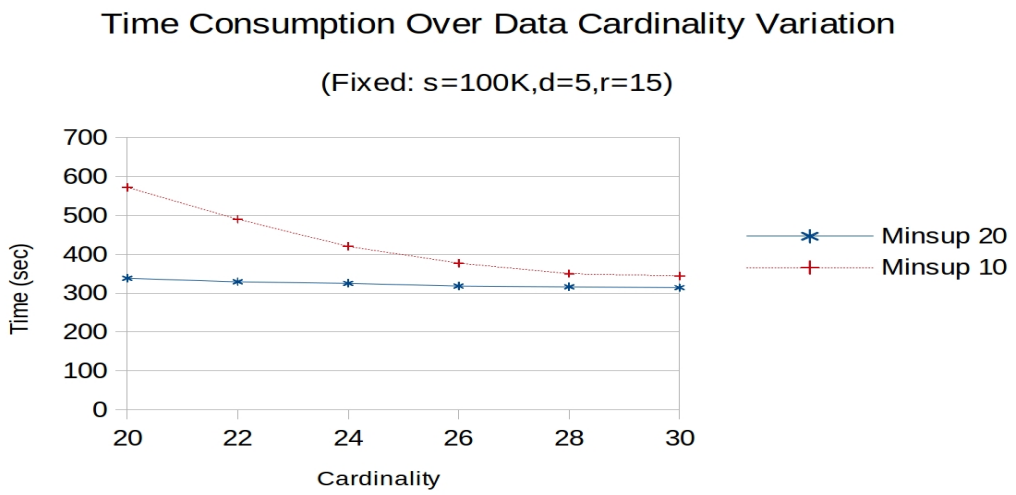

Figure 13: Time Consumption for Various Cardinalities

Next, in order to check minimum support variation effects, we used the same data set size of 100K tuples, and kept reduced the time consumption effect of all other fixed parameters; used 5 dimensions, used a cardinality of 30 that presented the lower time consumption in the previous experiments and again used almost all the reducers of the cluster (r=15). As indicated in our first group of experiments, this second group of experiments presented a stable calculation time for minimum support over 20 tuples. The experimental results presented a drastic alteration of computational behavior for minimum support up to 10 tuples. This behavior was expected since for the full cube computation (minSup=1) no pruning takes place and calculation time is the biggest possible, while for minimum support increasing the pruning also increases and the calculation time decreases. The behavior of the algorithm during these experiments is illustrated in Figure 14.

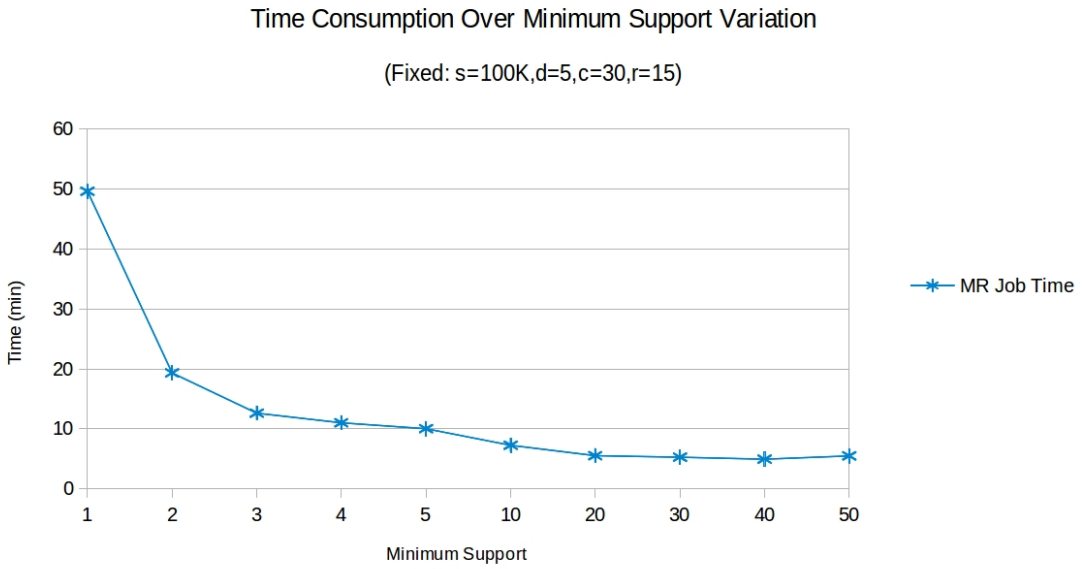

Figure 14: Time Consumption for Various Minimum Supports

Accordingly, a number of experiments was performed in order to clarify the -expected- behavior of reducer nodes' number; The same data set of 100K tuples was used as input, with all other fixed parameters set to low consuming effect numbers; dimensionality set to 5, minimum support to 10, and cardinality to 30. As expected, the influence of the reducers number was big; with almost all the nodes of the cluster (r=15), we achieved the best performance (Figure 15). Although the effect was not linear, there couldn't be recognized a pick point, and it is assumed that for the exponentially growing computation of the data cube, an even bigger cluster size would perform better. The results of this experiment in a closer look present the reduction of time in half when using triple reducers. As the Hadoop architecture normally leads to the reduction of time in half when the reducers are doubled, we observed here the influence of our choice not to use all the mappers possible and increase the parallelization, but let the system decide the number of mappers in each stage.

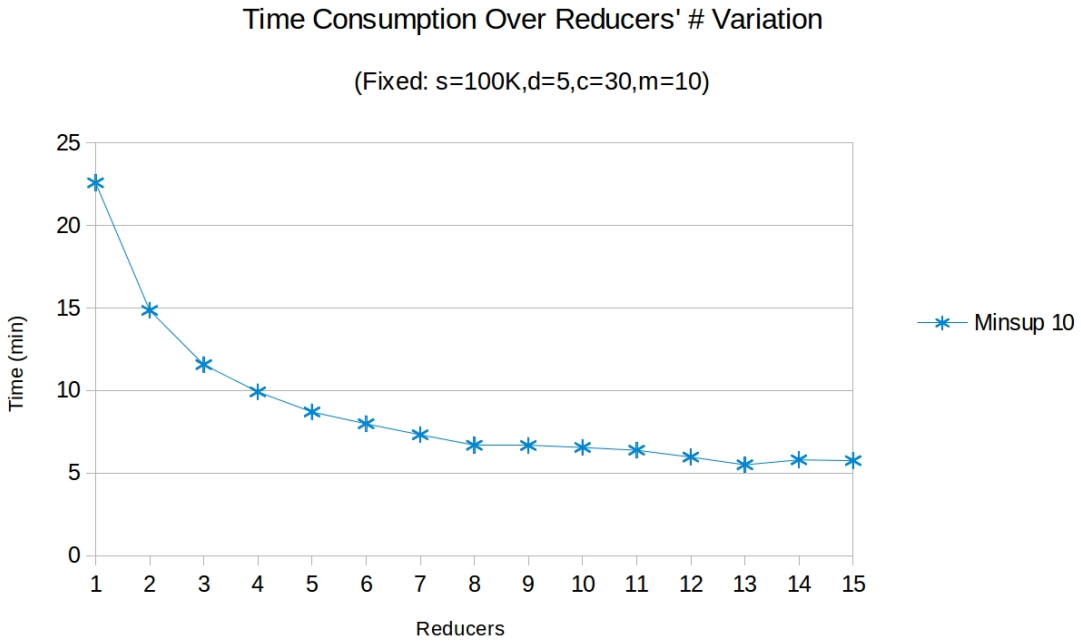

Figure 15: Time Consumption for Variation of Reducers' Number

Having experimented over the parameters that do not affect the size of the input, and reached a conclusion of the parameter value areas that these futures behave well, we proceeded on experimenting with the dimensionality and the number of records of the input, which affect the input size. First we performed a set of experiments over dimensionality, keeping the rest of the parameters fixed (cardinality=30, minimum support=10, reducer nodes=15, 100K tuples). We observed a very good response of the algorithm which handled scaling smoothly concerning job execution time. The results of these experiments are illustrated in Figure 16.

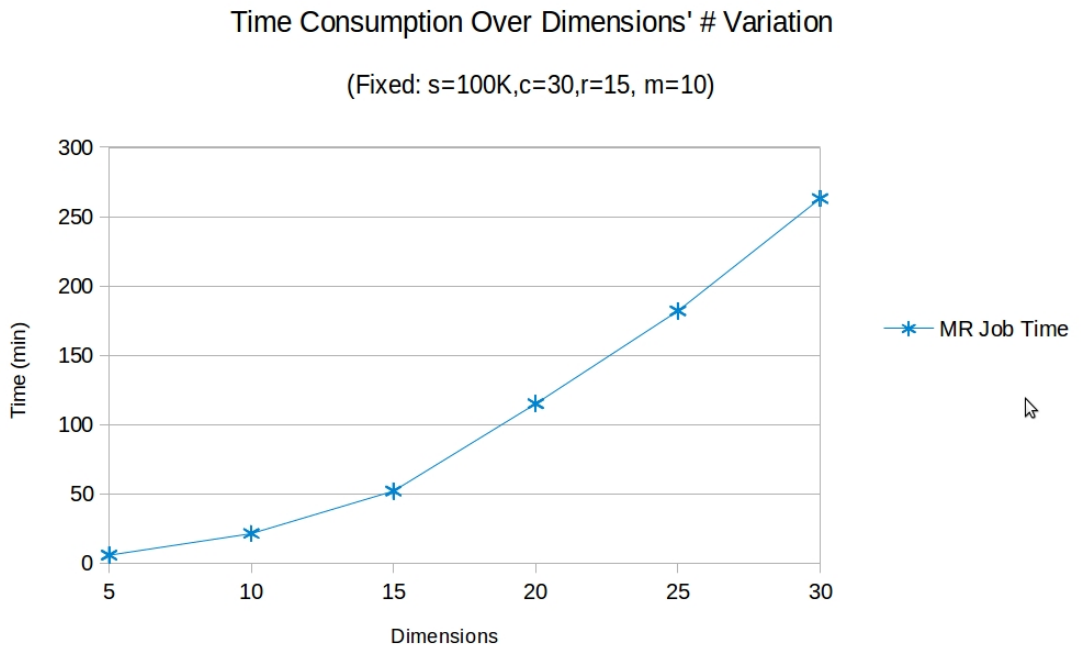

Figure 16: Time Consumption for Variation of Dimensions

Next, we experimented using increasing number of tuples to produce a scalability challenge for the algorithm; We kept all other parameters fixed in their good behavior area (cardinality 30, reducer nodes 15, dimensions 5) and experimented with data sets of 500 thousand to 3 million tuples in two groups of experiments with minimum support of 5 and 10 tuples; As a result we observed a high scaling ability. The results of these experiments are illustrated in Figure 17. The increased consumption of time when dimensions number or dataset size became big, presented a clear image of the curse of dimensionality, but our algorithm managed to handle the effect smoothly.

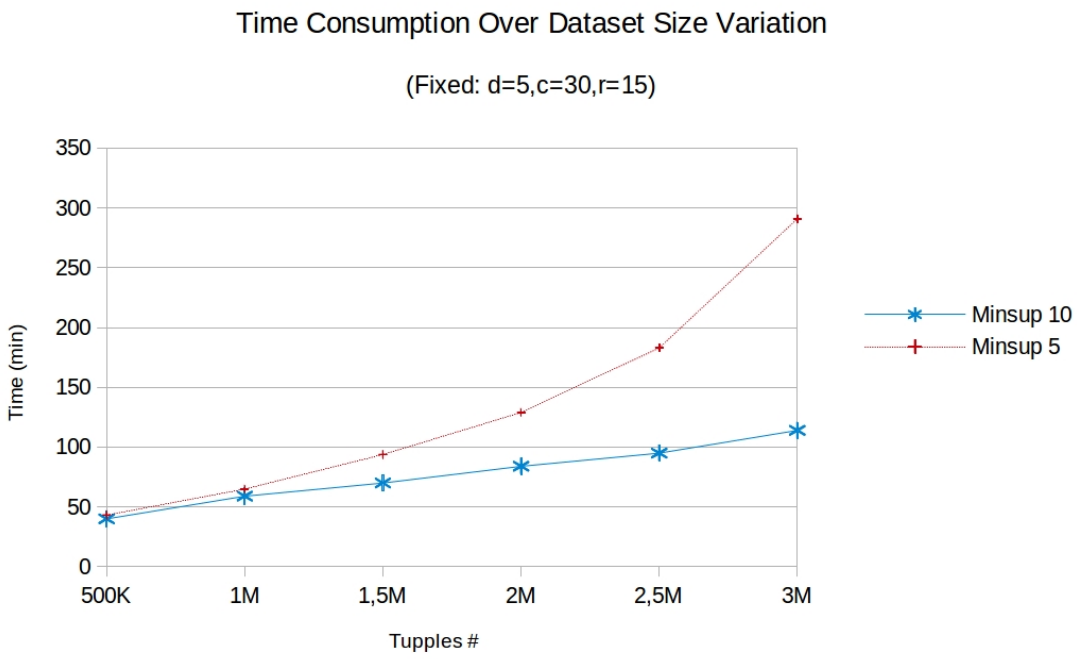

Figure 17: Time Consumption for Variation of the Number of Tuples

Finally, we experimented with data sets containing data not following a uniform distribution, with high cardinalities, in order to observe the behavior of the algorithm with sparse data. The results of these experiments presented a very good behavior of BucDoop with sparse data as depicted in Figure 18. Throughout the experimentation phase we observed that no matter the number of dimensions, the calculation time was mostly consumed in the first 3 jobs where the single, double and triple attribute group-bys took place. In the case of sparse data with big cardinalities the calculation time was mainly consumed in the first job. In comparison with the results presented earlier in Figure 13 with dense data, we observed that the performance was dominated by the data set size and the minimum support and not the sparse distribution of the data, since the data needed a single pass over them and the sparseness of the data led to high pruning and consuming time only in the first job during our experiments with 5 dimensions.

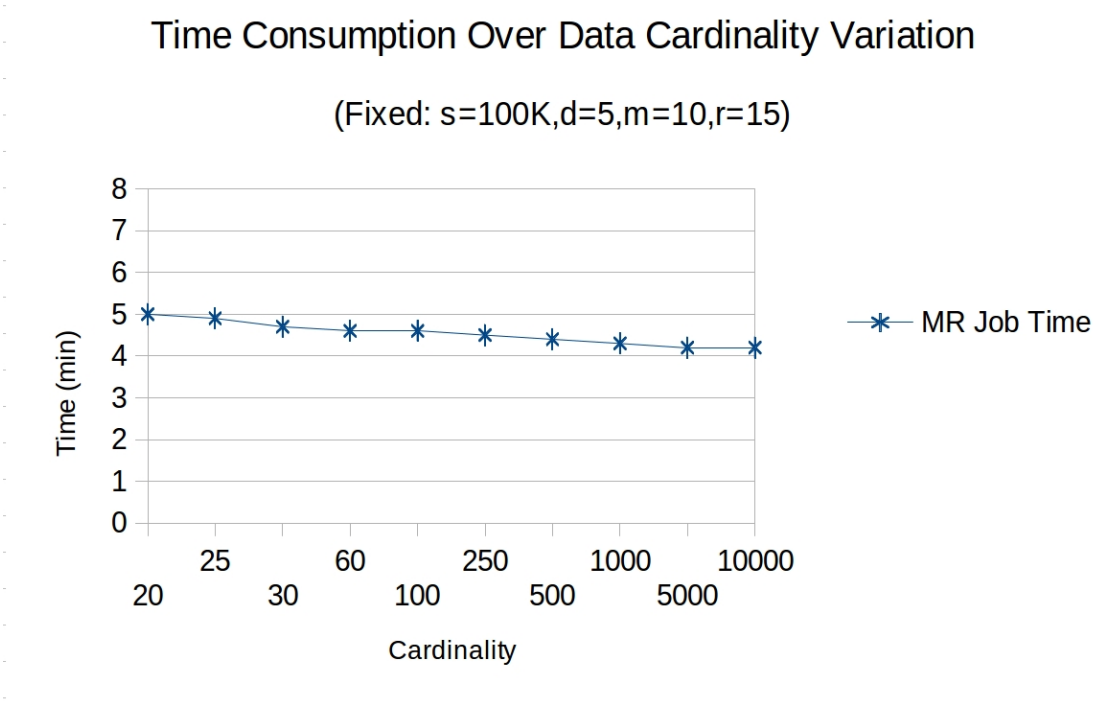

Figure 18: Time Consumption for Various Cardinalities

As a general observation on the overall performance the algorithm obviously exploited the advantages of both BUC efficiency and Map Reduce model scalability. The algorithm handled the processing of the various datasets that were tested in reasonable times and presented smooth scalability while the dataset sizes used were increasing

Chapter 6 - Conclusions-Future Work

6.1. Conclusions

In this Thesis report we made an effort of providing a solid and efficient proposal for the Bottom-Up-Computation of iceberg cubes with Hadoop Map Reduce platform. The former proposed ideas for iceberg cube computation were presented shortly and their main conclusions were taken into account. We also discussed the philosophy behind the distributive model of Hadoop Map reduce. An effort to efficiently implement the Bottom Up Computation high-performance characteristics and to adjust them to Hadoop was made.

As depicted in the previous chapters, bottom up computation, which is an efficient algorithm for computing sparse iceberg or even conventional cubes, can be further improved by the implementation of distributed philosophy of the Map Reduce model. The disadvantages that dense and big inputs present with BUC can be handled by this model which includes native processing of the main characteristics of BUC; Partitioning and Sorting.

6.2. Future Work

The possibility of further minimizing the data scans for cases of vast data sets like the ones this proposal tried to cover will certainly be beneficial. Although the data exchange through the proposed algorithms and tested implementation, are as minimized as possible, and the inputs need maximum two scans to be processed, the expected size of data does provide big cost even with a single scan.

An idea which could be investigated would be that of adding functionality to the proposed algorithms, in order to provide pre-computed partition metrics to the reducer. For example, the appropriate use of a combining function that would send

count and aggregate metrics for the processing partition to the reducer, so that they can be exploited before scanning the data, would further minimize the scans to only one.

References

[1] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F. and Pirahesh, H. (1997). Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), pp.29—53.

[2] Zhao, Y., Deshp, e, P. and Naughton, J. (1997). An Array-based Algorithm for Simultaneous Multidimensional Aggregates. *SIGMOD Rec.*, 26(2), pp.159--170.

[3] Beyer, K. and Ramakrishnan, R. (1999). Bottom-up Computation of Sparse and Iceberg CUBE. In:ACM. pp.359—370.

[4] Xin, D., Han, J., Li, X. and Wah, B. (2003). Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. pp.476—487.

[5] Ross, K. and Srivastava, D. (1997). Fast Computation of Sparse Datacubes. In: *Morgan Kaufmann Publishers Inc.*. pp.116—125.

[6] Agrawal, R. and Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. In: *Morgan Kaufmann Publishers Inc.*. pp.487--499.

[7] Agarwal, S., Agrawal, R., Deshp, e, P., Gupta, A., Naughton, J., Ramakrishnan, R. and Sarawagi, S. (1996). On the computation of multidimensional aggregates. 96, pp.506—521.

[8] Findlater, L. and Hamilton, H. (2003). Iceberg-cube algorithms: An empirical evaluation on synthetic and real data. *Intelligent Data Analysis*, 7(2), pp.77--

97.

[9] Chen, Y., Dehne, F., Eavis, T. and Rau-Chaplin, A. (2005). PnP: Parallel And External Memory Iceberg Cubes. In: *IEEE Computer Society*. pp.576-577.

[10] Li, X., Hamilton, H., Karimi, K. and Geng, L. (2009). The Multi-Tree Cubing algorithm for computing iceberg cubes. *Journal of Intelligent Information Systems*, 33(2), pp.179--208.

[11] Han, J., Pei, J., Dong, G. and Wang, K. (2001). Efficient computation of iceberg cubes with complex measures. 30(2), pp.1--12.

[12] Shao, Z., Han, J. and Xin, D. (2004). MM-Cubing: Computing iceberg cubes by factorizing the lattice space. pp.213--222.

[13] Cho, M., Pei, J. and Cheung, D. (2005). Cross table cubing: Mining iceberg cubes from data warehouses. pp.461--465.

[14] Wang, K., Jiang, Y., Yu, J., Dong, G. and Han, J. (2005). Divide-and-approximate: A novel constraint push strategy for iceberg cube mining. *Knowledge and Data Engineering, IEEE Transactions on*, 17(3), pp.354--368.

[15] Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In:*USENIX Association*. pp.10—10.

[16] White, T. (2012). *Hadoop: the definitive guide*. 3rd ed. Beijing: O'Reilly Media / Yahoo Press.

[17] Sismanis, Y., Deligiannakis, A., Roussopoulos, N. and Kotidis, Y. (2002). Dwarf: Shrinking the petacube. pp.464--475.

[18] Hortonworks Inc. (n.d.). 1. Apache Hadoop core components - Getting Started Guide. Retrieved July 6, 2014, from http://docs.hortonworks.com/HDPDocuments/HDP1/HDP-1.2.0/bk_getting-started-guide/content/ch_hdp1_getting_started_chp2_1.html

[19] Java J2EE Tutorials : MapReduce. (n.d.). Retrieved July 6, 2014, from http://www.j2eebrain.com/java-J2ee-mapreduce.html

[20] MapReduce Patterns, Algorithms, and Use Cases | Highly Scalable Blog. (n.d.). Retrieved July 6, 2014, from http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/

[21] Lin, J. and Dyer, C. (2010). *Data-intensive text processing with MapReduce*. 1st ed. [San Rafael, Calif.]: Morgan & Claypool Publishers.

[22] Ng, R., Wagner, A. and Yin, Y. (2001). Iceberg-cube computation with PC clusters. 30(2), pp.25--36.