SCALABLE DATA ANALYSIS ON MAPREDUCE-BASED SYSTEMS

WANG ZHENGKUI

Master of Computer Science

Harbin Institute of Technology

Bachelor of Computer Science

Heilongjiang Institute of Science and Technology

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

NUS GRADUATE SCHOOL OF INTEGRATIVE SCIENCE AND ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2013

Declaration

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

10/10/2013

WANG ZHENGKUI

10 October 2013

ACKNOWLEDGEMENT

"Let the peace of Christ rule in your hearts, to which indeed you were called in the one body. And be thankful." -Colossians 3:15

This thesis would not have been completed without the support of many people. I would like to reserve this section to express my gratitude to all of them.

First and foremost, I would like to thank my supervisor Professor Kian-Lee Tan. I would like to express my heartfelt gratitude and appreciation for his invaluable guidance and inspiration in this research, his moral support and encouragement during the duration of my Ph.D study. It is a privilege to work under him, and he has set a good example to me in many different ways. His insights and knowledge in this area play an important role in completing this thesis. As a supervisor, he shows me not only how to be a good researcher with rigorous research attitude, but also how to build a good personality with humility and gentleness. All that I have learned from him will be of great influence for my research and my entire life.

Professor Divyakant Agrawal, who has collaborated with me in many of my research works, deserves my special appreciations. He has provided many precious advices during my Ph.D work. His insight of research has inspired me to find a lot of interesting research problems. I would also like to thank him for inviting me to visit the University of California at Santa Barbara, UCSB as a research scholar. That provided me a

good opportunity to meet many professors and researchers in UCSB. I would also like to thank Professor Amr EI Abbadi who has co-hosted me together with Divy in UCSB. I am grateful for his help as well as his guidance during my stay there.

My deep gratitude also goes to Professor Wing-Kin Sung and Professor Roger Zimmermann for being my thesis committee members, monitoring and guiding me in my Ph.D research. I am grateful for their precious time to meet with me for each TAC regular meeting every year. They always provide many precious questions and comments which have inspired me during my research.

I also wish to thank all the people collaborating with me during the last few years: Professor Limsoon Wong, Qian Xiao, Huiju Wang, Qi Fan, Yue Wang, Xiaolong Xu. It was a great pleasure to collaborate with each of them. Their participation further strengthened the technical quality and literary presentation of our papers.

In NUS, I have met a lot of friends who brought a lot of fun to my life, especially, Yong Zeng, Htoo Htet Aung, Wei Kang, Nannan Cao, Luocheng Li, Lei Shi, Lu Li, Guoping Wang, Zhifeng Bao, Xuesong Lu, Yuxin Zheng, Ruiming Tang, Jinbo Zhou, Hao Li, Yi Song, Fangda Wang and all the other students and professors in the entire database labs. I would also thank all of my friends who have made my life much colorful in UCSB, especially, Wei Cheng, Xiaolong Xu, Ye Wang, Shiyuan Wang, Sudipto Das, Aaron Elmore, Ceren Budak, Cetin Sahin, Faial Nawab and many other church friends.

Furthermore, I would like to thank NUS Graduate School of Integrative of Science and Engineering, National University of Singapore for providing me the scholarship during my PhD study.

Last but not least, my deepest love is reserved for my parents Baoren Wang and Suolian Wang, my brother and sister-in-law Xuefa Wang and Feng Yan. They are always supporting, encouraging and loving me. I thank God for blessing me in such a manner to put all of them in my life.

CONTENTS

SUMMARY

Many of today's applications, such as scientific, financial and social networking applications, are generating and collecting data at an alarming rate. As the size of data grows, it becomes increasingly challenging to analyze these datasets. The high computation and I/O cost of processing large amount of data make it difficult for these applications to meet the performance demands of end-users.

Meanwhile, the MapReduce framework has emerged as a powerful parallel computation paradigm for data processing on large-scale clusters. As such, there has been much effort in developing MapReduce-based algorithms to improve performance. However, there remain many challenges in exploiting MapReduce for efficient data analysis. Thus, designing new scalable, efficient and practical parallel data processing algorithms, frameworks and systems for computation intensive analysis and data intensive analysis is the research problem of this thesis.

In this thesis, we explore two extremely important and challenging analyses: Combinatorial Statistical Analysis (CSA, as an representative example of computation intensive analysis to finding the significant objects correlations that is measured by statistical methods) and Online Analytical Processing (OLAP) cubes analysis (as an representative

example of data intensive analysis to materialize the data in support of efficient query response and decision making in data warehousing).

First, we adopt the MapReduce computation paradigm to develop a highly scalable and generic framework with two alternative computation schemes (exhaustive testing and semi-exhaustive testing) for the CSA problem. It is able to distribute the computation task to each processing unit for the analysis with any number of objects with a good load balancing. We also propose new techniques to speed up the statistical testing among different combinations of objects. By incorporating these techniques, our framework obtains great efficiency and scalability towards a large number of objects that none of the existing frameworks are able to achieve. Second, we develop a distributed system, HaCube which is an extension of MapReduce, designed for efficient parallel data cubes analysis for large-scale multidimensional data in traditional OLAP and data warehousing scenario. We propose a generic parallel cubing algorithm to materialize the cube efficiently. We also investigate the view update problem and provide the techniques to update the view when new data is inserted. This, to the best of our knowledge, is the first work to study view maintenance in MapReduce-like environment. Third, we extend the data cubes analysis to a more complex data structure, attributed graphs where both vertex and edge are associated with attributes. Specifically, we propose a new conceptual graph cube model, Hyper Graph Cube, based on the attributed graphs, since the traditional data cubes are no longer applicable in graphs. This is also the first work to develop a MapReduce-based distributed and parallel graph cube materialization solution towards the graph OLAP on large-scale graphs.

We have implemented the above techniques and conducted extensive experimental studies. The experimental results demonstrated the efficiency, effectiveness and scalability of our approaches. We believe that our research in this thesis brings us one step closer towards developing scalable and efficient big data analysis systems.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

The amount of data in our world has been exploding, such as scientific data, industry sales data, finance data, social network data etc. These data resources contain a wealth of information that is of benefit to different communities. A better understanding of these data may help us have a better insight of the world, better target marketing campaigns and perform a better decision making support in industries etc. Analyzing large data sets has become one of the main challenges in various enterprises. Thus, the design of efficient methods for big data analysis has drawn a tremendous attention from both industries and academia recently.

Due to the increasing size of data, analyzing these data becomes quite difficult. The difficulty of analyzing these large-scale data arises because of either the high computation overhead or the high I/O overhead incurred in big data processing. In such a data explosion era, existing techniques developed on a single server or a small number of

machines are unable to provide acceptable performance. Therefore, many studies have endeavored to overcome the limitations of existing techniques to face the challenges arisen by data.

My research aims at developing new techniques towards an efficient and effective large-scale data processing and analysis. Given that the applications could be either computation intensive or data intensive, this thesis studies both of these two categories of applications. Specifically, this thesis explores two extremely important but challenging analyses, combinatorial statistical analysis (CSA) and Online Analytical Processing (OLAP) cubes analysis. The former is a representative example of computation-intensive applications, while the latter represents data-intensive applications.

1.2 Research Problems and Challenges

In this thesis, we propose to exploit parallelism to speed up the data analysis in computation and data intensive applications. Today, we are facing good opportunities to develop scalable data analysis systems. On the one hand, the large amount of computation resources become available to each user, especially benefited from the emergence of cloud computing. Cloud computing has emerged as a successful and ubiquitous paradigm for service oriented computing. The major advantages that make cloud computing attractive are: *pay-as-you-use* pricing resulting in low time to market and low upfront investment for trying out novel application ideas; *elasticity*, i.e., the ability to scale the computation resources and capacity as you need. This provides us powerful computation resources to all users to deploy a real scalable and elastic data analysis system in a large infrastructure.

On the other hand, MapReduce (MR) has emerged as a powerful parallel computation paradigm for data processing on large-scale clusters. It becomes a very popular and

attractive platform due to its *high scalability* (scale to thousands of machines), *good fault tolerance* (automatic failure recovery), and *ease-of-programming* (simple programming logic). More importantly, the MR framework has been integrated with the cloud so that each user can easily deploy their MR-based algorithms to the cloud with low expense.

Based on this, we are able to develop real scalable data analysis systems by adopting MR as the data processing engine over the large-scale cluster. However, it is non-trivial to develop such MR-based data analysis operators. A naive data processing solution over MR may be very costly. Thus, the research problem, in this thesis, is to explore the efficient big data analysis techniques over the MR computation paradigm. Since the analysis could be either computation intensive or data intensive, we tackle the problems for both of these categories of analyses in this thesis.

1.2.1 Computation Intensive Analysis

Computation intensive analysis involves high computation overhead where parallelizing these computation tasks will reduce the total data processing time. In this thesis, we take the combinatorial statistical analysis as an example to explore a practical parallel solution for computation intensive applications.

Combinatorial Statistical Analysis (CSA) plays an important role in finding the significant correlations that are typically measured by statistical methods among different objects. Finding such correlations between multiple objects may help us better understand their relationships. Intuitively, CSA evaluates the significance of the associations between a combination of objects by adopting the statistical methods, such as χ^2 test. Due to the power of the statistical methods, CSA has been widely used in many different applications to find the associations between objects, especially in scientific data analysis.

As an example, CSA is used in epistasis discovery to determine the association

among a combination of Single Nucleotide Polymorphisms (SNPs) that cause complex diseases(e.g. breast cancer, diabetes and heart attacks)[47][74][90][80][81].

From a computational point of view, finding significant associations is very challenging. On the one hand, scientists typically do not want to miss any answers. As such, the widely adopted solution is to exhaustively enumerate all possible combinations of a certain size, say k, in order to find all statistically significant associations of k objects [51]. Given n objects, there are $C(n,k) = \frac{n!}{(k!(n-k)!)}$ combinations to evaluate.

On the other hand, the cost for computing statistical test to evaluate the association significance of one combination is high. As such, for a large number of combinations, it will take a very long time to complete the processing.

Thus, the research problem we have here is how to build a scalable, practical, efficient and effective parallel cloud-based *C*SA computation framework on MR. In particular, an efficient and effective scheme must address two challenges:

1. Given the large number of combinations, they must be distributed evenly across the processing units; otherwise, the unit with a significantly bigger load will become a bottleneck. Therefore, a distribution scheme that balances the load should be developed. Meanwhile, the solution should be able to scale well towards a large-scale data analysis.

2. At a particular unit, we also have a large number of combinations being allocated, each of which requires an expensive statistical test. The naive strategy of processing these tests independently is inefficient. Instead, a scheme that can minimize the computation for efficient statistics testing should be designed.

1.2.2 Data Intensive Analysis

Besides the computation intensive analysis, we also want to study the processing of data-intensive applications. In such applications, the computation difficulty is not the main bottleneck but high I/O overhead incurred by the large volume of data. Decision

support systems that run aggregation queries over data warehouse is an example.

OLAP data cubes [31] are one such critical technology that has been used in data warehousing and OLAP to support decision making. Given n dimensions, data cubes normally precompute a total of 2^n **cuboids** or group-bys, where each cuboid or group-by captures the aggregated data over one combination of dimensions. Each of such cuboid can be stored into a database as a view to speed up query processing.

There are two key operations in data cube analysis. The first is **data cube materialization** where the various cuboids are computed and stored as views for further observation and query support. The second is **data cube view maintenance** where the materialized views are updated when new data is inserted. Both these operations are computationally expensive, and have received considerable attention in the literature [7][95][96][44] .

Therefore, in this thesis, our research problem is to deploy an efficient and scalable data cube analysis system targeting on a large amount of data over the MR-like computation paradigm. To design such a distributed system, the main challenges can be summarized as follows:

1. Given n dimensions in a relation, there are 2^n cuboids to be computed to materialize the cube. An efficient parallel algorithm to materialize the cube faces two sub-challenges: (a) Given that some of the cuboids share common dimensions, is it possible to batch these cuboids to exploit some common processing? (b) Assuming we are able to create batches of cuboids, how can we allocate these batches or resources so that the load across the processing nodes is balanced?

2. View maintenance in a distributed environment introduces significant overheads, as large amounts of data (either the materialized data or the base data) need to be read, shuffled and written among the processing nodes and distributed file system (DFS). Moreover, for non-distributive measures, recomputation is necessary to update the views.

It is thus critical to develop efficient view maintenance methods for a wide variety of frequently used measures.

Furthermore, we extend the OLAP cubes analysis to a more complex structured data, attributed graphs where both the vertex and the edge are associated with attributes. The attributed graph has been widely used to model the information networks. Attributed graphs become quite ubiquitous due to the astounding growth of different information networks such as the Web and various of social networks(e.g. Facebook, LinkedIn, RenRen).

Obviously, these attributed graphs contain a wealth of information. Analyzing such information may provide us an accurate and implicit insight of the real world. For instance, analyzing the relationship (edge) information in a social network may help us to better understand how users interact with each other among different communities.

However, the traditional OLAP cubes are no longer applicable to graphs, since the edges(relationship information) have to be considered in graph warehousing. The traditional data cubes only aggregate the numeric value based on the group-bys and are unable to capture the structural information.

In order to conduct graph OLAP, a new conceptual graph cube model has to be designed on a graph context further. And then, to support large graphs, there is a need to develop a parallel graph cube computation algorithm such that it is practical and scalable enough in processing the large graphs we are facing today.

1.3 Contributions of This Thesis

To solve the research problems aforementioned, we propose several new algorithms, frameworks and systems in this thesis. Our main contributions are summarized here.

In the first part of this thesis, we propose a MR-based framework, *COSAC* -COmbinatorial Statistical Analysis on Cloud platforms for the CSA problem. Our contributions

are:

- We propose an efficient and flexible object combination enumeration framework with good load balancing and scalability for large scale of datasets using the MR paradigm. The enumeration of combinatorial objects takes an important role in computer science and engineering [64]. We develop schemes for enumerating the entire set of objects (Exhausitive Testing) as well as a subset of the set (Semi-exhaustive Testing). Our framework is useful beyond scientific data processing; it is suited for any applications that need to enumerate the objects set.

- We propose a technique for efficient statistics analysis using IRBI (Integer Representation and Bitmap Indexing) which is both CPU efficient with regard to statistics testing, and storage and memory efficient. Statistics methods have been widely used as powerful tools in many different applications, e.g. data mining, machine learning. The approach we adopted in our thesis can be a promising solution to speed up the statistical testing.

- We propose an optimization technique based on the sharing of computation to salvage computations that can be reused during statistical testing with significant performance savings, instead of conducting the testing for each combination independently.

- We implement the framework and conduct extensive experimental evaluation. The results indicate that our framework is able to conduct analysis in hours where the task normally took weeks, if not months. To the best of our knowledge, non of the existing framework has such a computation capability.

In the second part of this thesis, to develop a scalable parallel data cube analysis platform on big data, we develop a distributed system, `HaCube`, integrating a new data

cubing algorithm and an efficient view maintenance scheme. Our main contributions in this work are as follows:

- We present a distributed system, `HaCube`, an extension of MR, for data cube analysis on large-scale data. `HaCube` modifies the Hadoop MR framework while retaining good features like ease of programming, scalability and fault tolerance. It also builds a layer with user-friendly interfaces for data cube analysis. We note that `HaCube` retains the conventional Hadoop APIs and, thus, is compatible with MR jobs.

- We show how batching cuboids for processing can minimize the read/shuffle overhead to salvage partial work done for efficient data cube materialization.

- We propose a general and effective load balancing scheme $\mathcal{LBCCC}$ (short for Load Balancing via Computation Complexity Comparison) to ensure that resources are well allocated to each batch. $\mathcal{LBCCC}$ can be used under both HaCube and MR frameworks.

- We adopt a new computation paradigm, MMRR (`MAP-MERGE-REDUCE-REFRESH`), with a local store under `HaCube`. `HaCube` supports efficient view updates for different measures, both distributive such as SUM, COUNT and non-distributive such as MEDIAN, CORRELATION. Thus, this is able to support more applications with data cube analysis in a data center environment. To the best of our knowledge, this is the first work to address data cube view maintenance in MR-like systems.

- We evaluate `HaCube` based on the TPC-D benchmark with more than one billion tuples. The experimental results show that `HaCube` has significant performance improvement over Hadoop.

In the third part of this thesis, we further tackle the graph OLAP problem where a new graph OLAP model and a parallel solution over the attributed graphs have been proposed. Our main contributions in this work are in the following aspects:

- We propose a new conceptual graph cube model, Hyper Graph Cube, to extend decision making services on attributed graphs. Hyper Graph Cube is able to capture queries in different categories into one model. Moreover, the model supports a new set of OLAP Roll-Up/Drill-Down operations on attributed graphs.

- We propose several optimization techniques to tackle the problem of performing an efficient graph cube computation under the MR framework. First, our self-contained join strategy can reduce I/O cost. It is a general join strategy applicable to various applications which need to pass a large amount of intermediate joined data between multiple MR jobs. Second, we combine cuboids to be processed as a batch so that the intermediate data and computation can be shared. Third, a cost-based optimization scheme is used to further group batches into bags (each bag is a subset of batches) so that each bag can be processed efficiently using a single MR job. Fourth, a MR-based scheme is designed to process a bag.

- We introduce a cube materialization approach, MRGraph-Cubing, that employs these techniques to process large scale attributed graphs. To the best of our knowledge, this is the first parallel graph cubing solution over large-scale attributed graphs under the MR-like framework.

- We conduct extensive experimental evaluations based on both real and synthetic data. The experimental results demonstrate that our parallel Hyper Graph Cube solution is effective, efficient and scalable.

The works in this thesis have resulted in a number of publications, more specifically, [80], [81] and [77], [79] and [78].

1.4 Thesis Outline

The thesis is organized as follows. In Chapter 2, we first provide the preliminaries on MapReduce and then review the related works. For the CSA problem, we focus on work related to epistasis discovery. We also review the existing data cubes analysis techniques including three classic cubing approaches and parallel computation solutions, and the graph OLAP works.

We then present our proposed COSAC framework for combinatorial statistical analysis in Chapter 3. In this chapter, we demonstrate how to use MR to develop a highly scalable and efficient framework that parallelizes the computation tasks in the computation intensive analysis.

Chapter 4 introduces a distributed system, HaCube, designed for an efficient parallel data cube analysis on the traditional relational data. This chapter shows how MR can be extended to support traditional data cubes analysis. We will also introduce the system architecture of HaCube, a new cubing algorithm for cube materialization as well as the new view maintenance strategies in HaCube.

In Chapter 5, we present our proposed Hyper Graph Cube model and a MR-based cube computation framework. We also introduce other graph OLAP operations and challenges in graph OLAP.

Finally, we conclude this thesis and discuss some future research work in Chapter 6.

CHAPTER 2

RELATED WORK

In this chapter, we first introduce the preliminaries of MapReduce. Then, we focus on some related works. More specifically, we first present some related work on the Combinatorial Statistical Analysis-CSA problem. In particular, we focus on existing works on epistasis discovery. Then, we review the most closely related works on existing data cube processing and graph OLAP analytics.

2.1 Preliminaries on MapReduce

MapReduce (MR), which was first proposed in [22], has emerged as a powerful parallel computation paradigm. MR has several advantages which make it attractive, such as its high scalability (scalability of thousands of machines), good fault tolerance (automatic failure recovery by the framework), ease-of-programming (simple programming logic) and high integration with cloud(availability to every user and low expense with a pay-as-you-go model). It has been widely used by various applications such as scientific

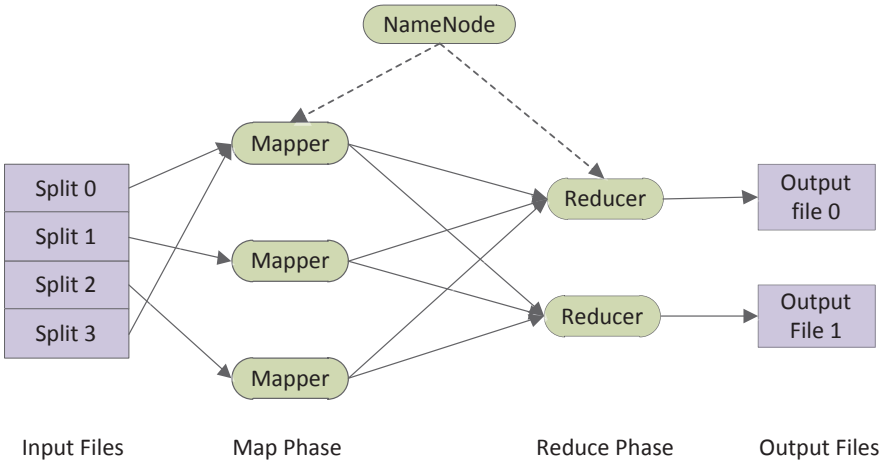


Figure 2.1: The MapReduce computation paradigm

data processing, media data processing, data mining and machine learning etc.

Under the MR framework, the system architecture of a cluster consists of two kinds of nodes, namely, the NameNode and DataNodes. The NameNode works as a master of the file system and is responsible for splitting data into blocks and distributing the blocks to the data nodes (DataNodes) with replication for fault tolerance. A JobTracker running on the NameNode keeps track of the job information, job execution and fault tolerance of jobs executing in the cluster. A job may be split into multiple tasks, each of which is assigned to be processed at a DataNode.

The DataNode is responsible for storing the data blocks assigned by the NameNode. A TaskTracker running on the DataNode is responsible for the task execution and communicating with the JobTracker.

The computation of MR follows a fixed model with a `map` phase and followed by a `reduce` phase [22]. Figure 2.1 provides the MR computation paradigm. The MR library is responsible for splitting the data into chunks and distributing each chunk to the processing units (called `mappers`) on different nodes. The `mappers` process the data read from the file system and produce a set of intermediate results which are shuffled to the other processing units (called `reducers`) for further processing. Users can set their

application logic by writing the `map` and `reduce` functions in their applications.

Map Phase: The `map` function is used to process $(key, value)$ pairs $(k1, v1)$ which are read from data chunks. Through the `map` function, the input set of $(k1, v1)$ pairs are transformed into a new set of intermediate $(k2, v2)$ pairs. The MR library will sort and partition all the intermediate pairs and pass them to the `reducers`.

A partitioning function is responsible to partition the pairs emitted from the `map` phase into M partitions on the local disks, where M is the total number of `reducers`. The partitions are then shuffled to the corresponding `reducers` by the MR library. Users can specify their own partitioning function or use the default one provided by the MR framework.

Reduce phase: At the `reducer`, the intermediate $(k2, v2)$ pairs with the same key that are shuffled from different `mappers` are sorted and merged together to form a values list. The key and the values list are fed to the user-written `reduce` function iteratively. The `reduce` function makes a further computation to the key and values and produces new $(k3, v3)$ pairs. The output $(k3, v3)$ pairs are written back to the file system.

2.2 Combinatorial Statistical Analysis

Combinatorial statistical analysis (CSA) plays an important role in many scientific applications to find significant object associations. In this thesis, we focus on epistasis discovery as one representative application, which has widely adopted CSA. Hence, in this section, we provide the related works for CSA in epistasis discovery.

In epistasis discovery, scientists aim to discover the correlation between a combination of Single Nucleotide Polymorphisms (SNPs) and the diseases such as heart attack and cancer. Traditionally, many researchers focused on the association of individual

SNPs with the phenotypes (such as the diseases). However, these methods can only find weak associations as they ignore the joint genetic effects, which is called Epistasis, across the whole genome [50]. Recently, there has been a shift away from the one-SNP-at-a-time approach towards a more holistic and significant approach that detects the association between a combination of multiple SNPs with the phenotypes [49]. In the meanwhile, the number of discovered SNPs is becoming larger and larger. For example, the Hapmap project provides the dataset containing 3.1 million SNPs [24]. Determining the interactions of SNPs has become a very time-consuming job from a computational perspective.

To discover such a significant association, statistical modeling techniques have been proposed [59][83][82]. However, these statistical modeling methods, which work well for a small number of SNPs, are not able to provide acceptable performance and become impractical when the number of SNPs increases enlarging the search space. To prune the search space, the heuristics techniques are proposed to speedup statistical modeling approach [93][92][91]. In particular, a filtering step is added to select a fixed number candidate SNPs. Then the selected candidate SNPs are exhaustively evaluated. On the other hand, many researchers still focus on the exhaustive enumerating approach to test all the possible pairs of SNPs [74][60]. Exhaustive enumerating guarantees that all the combinations of SNPs are tested, thus none of the significant associations will be missed.

However, all the aforementioned related works are designed on a single server machine, which has become no longer practical to provide acceptable computation performance, as the size of dataset and analysis order increases. Thus, due to such a computational difficulty, researchers have made great effort to exploit parallel processing to the computational challenge in epistasis discovering.

Ma at el. [47] proposed a parallel computation tool designed for two-locus analysis (checking the pair association) specially targeting on a supercomputer platform. Given

N SNPs, there are total $C(N, 2)$ pairs to evaluate. In order to distribute and enumerate the $C(N, 2)$ pairs into different processor cores, the N SNPs are first evenly divided into m subsets. Then each combination of the m subsets is sent to one processor core. Therefore, the total number of processor cores (p) needed is $p = m(m+1)/2$. For illustration, we define $n = N/m$. Among the p cores, there are m cores only receiving one subset to make a self-subset pairing operations to pair the SNPs among one subset. In these processors, each of them computes $n(n+1)/2$ pairs. For the rest $p - m$ cores, each of them receives two different subsets and conducts a cross-subset pairing operations where the SNPs from one subset are paired with the ones in another subset. In these cores, it is easy to see that $n * n$ pairs are evaluated in each core. Through their experimental results, they predict the time for pairwise epistasis testing among 1,000,000 SNPs using 2048 cores would require about 20 hours to complete [47].

However, first, N/m may not be always an integer in practice, while they assume that N/m is an integer in their paper. Second, based on the computation task assigned to each core, we can see that the load is not well balanced between the m cores (the ones conduct self-subset pairing) and the rest $p-m$ cores (the ones conduct cross-subset pairing). Third, they only introduce how to conduct the pairwise analysis. It is unclear and more challenging to make a high order analysis. The last but not the least, the tool is specially designed for a supercomputer system which is not easy for others to obtain and thus, not easy to have the proposed solution works on other computation resources such as a shared-nothing cluster.

Thong at el. [37][38] adopted the graphical processing units (GPUs) to exhaustively test all the SNPs pairs. However, the authors did not provide the implementation details on GPUs. Indeed, the GPU is more powerful than a single PC, since it has more computing units and large memory. However, it requires the researchers to fully understand the GPU architecture to optimize the parallel computation. It is still unclear how to develop

an optimized multi-threads program to process the pairs evaluation in parallel. Thong at el. design the analysis on single GPU. However, we argue that this is still not scalable since a single GPU may only have limited computing resources. A scalable technique may need to be able to perform on multiple machines.

In our works [80][81], we have provided the solution to solve the pairwise epistasis testing in genome-wide association study. However, it is more challenging to conduct a high order analysis for any generic *CSA* analysis. In thesis, we mainly focus our work where a flexible and general framework, *COSAC*, for any order of analysis are proposed [77]. *COSAC* is a more general framework which is computationally practical, efficient, scalable for *CSA* systems, and flexible to support any level of analysis with different optimization techniques. In particular, *COSAC* incorporated numerous extensions: (a) a general and flexible framework to support any level of analysis in *CSA* applications. It is non-trivial to perform the combinatorial statistical analysis when analysis level increases. The load balancing becomes more tricky in such a high order analysis scenario. (b) a new practical scheme to support partial enumeration when a scientist has already identified a set of key objects that (s)he would like to investigate further. (c) a novel sharing optimization to speed up the analysis when the analysis level is bigger than 2. (d) a new approach to reduce the memory utility in *CSA* applications.

2.3 Data Cube Analysis

Data Cubes play an important role in data warehousing and OLAP to precompute the aggregate values for different dimensions. Given n dimensions, there are total 2^n different combinations of dimensions, which is called cuboids. Efficient computation of data cubes has attracted a lot of research interests in the last two decades. For instance, given four dimensions A, B, C and D, all the 16 cuboids can be represented as a cube

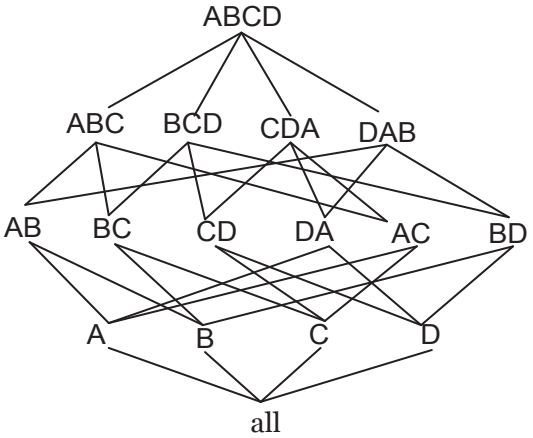


Figure 2.2: A cube lattice with 4 dimensions A, B, C and D

lattice as shown in Figure 2.2. All the research works can be classified into the following categories: (1) efficient computation of full or iceberg cubes: the computation of the full cube needs to compute the aggregate of each group in a complete cube, while the computation of iceberg cubes only needs to process the group which meet a certain condition or threshold[7] [11] [63] [33] [95]. (2) selective view materialization: these batch of researches aims to materialize only partial of the cubes instead of a complete cube [58][32] [35] [70]. (3) computation of special data cubes: these researches include computing condensed, quotient or dwarf cubes or compressed cubes by approximation such as wavelet cubes, quasi-cubes etc [75] [72] [69][42] [41].

The first one, efficient computation of full or iceberg, is of great importance among the aforementioned categories as it is the fundamental problem, and the new techniques for this category may have a strong influence to all the other categories [85]. Therefore, in this thesis, we focus on introducing the different computation approaches for materializing a full or iceberg cube in the literature. We classify the existing approaches into three categories on efficient cube computations, bottom-up, top-down and hybrid cube computations, each of which is introduced in the following sections.

2.3.1 Top-down Cube Computation

Zhao et al. [95] proposed a top-down computation approach (we refer this approach as MultiWay approach) which overlaps the computation of different group-bys based on a Multi-Way Array. The approach includes three-step procedure. First, it scans the table and loads it into an array. Second, it computes the cube on the resulting array. Third, it dumps the resulting cubed array into the tables. The array is used as an internal in-memory data structure to load the base cuboid and compute the cube. For a more memory efficient processing, the array may be partitioned into different chunks, each of which can be fit into memory.

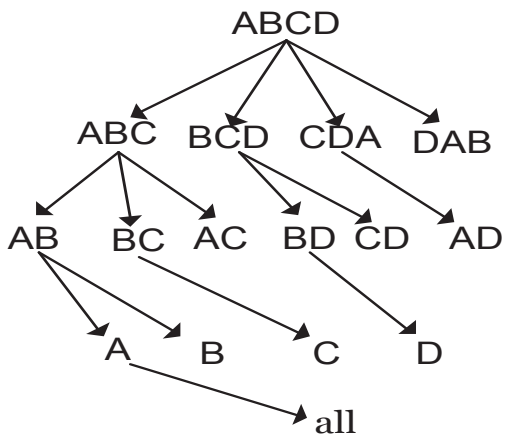


Figure 2.3: Top-Down Computation

To illustrate how this top-down approach computation works, we take the example in Figure 2.3 as a running example. Given four dimensions A, B, C and D, ABCD is considered to be the base cuboid. As shown in Figure 2.3, the results of computing cuboid ABC can be used to process AB and similarly the results of AB can be used to process A. This shared computation makes MultiWay approach efficient and allows different cuboids to be computed simultaneously. Figure 2.3 shows the entire execution plan based on the MultiWay approach. The advantages of this approach is that it uses the array indexing to avoid tuple comparison and the array structure offers compression

as well as indexing.

The MultiWay approach is not effective or feasible when the dimensionality is high and the data is too sparse, since the array and the intermediate results will be too big to fit into memory. Meanwhile, MultiWay cannot take advantage of the Apriori pruning [8] during the iceberg cubing. For instance, if one cell $A_1B_1C_1$ in ABC does not satisfy the condition such as $count(A_1B_1C_1) > t$, there is no guarantee that $count(A_1B_1) < t$, since a cell A_1B_1 in AB is likely to contain more tuples than in the cell $A_1B_1C_1$ in ABC.

2.3.2 Bottom-up Cube Computation

Beyer et al. [11] proposed another bottom-up cube computation approach which is referred as BUC computation. The idea of BUC is to combine the I/O efficiency of processing multiple cuboids, but to take advantage of minimum support pruning like Apriori. To achieve pruning, BUC processes the lattice from the bottom, the apex cuboid and moving upward to the larger, less aggregated group-bys, as shown in Figure 2.4. For instance, if the cell A_1 does not satisfy the condition of $count(\mathrm{A}_1) > t$, we are sure that the cell $\mathrm{A}_1\mathrm{B}_1\mathrm{C}_1$ does not satisfy the condition $count(\mathrm{A}_1\mathrm{B}_1\mathrm{C}_1) > t$ either, since it is likely that $\mathrm{A}_1\mathrm{B}_1\mathrm{C}_1$ contains less value than the cell A_1. Therefore, the computation of the up-level cuboids can be pruned by the low-level cuboid.

The majority of the run time in BUC is spent on partitioning the data. To facilitate efficient partitioning, the linear sorting method, CountingSort[67] is adopted. The CountingSort, is fast in BUC, since it does not perform any key comparisons to find boundaries and the counts computed during the sort can be reused to compute the group-bys. However, partitioning and sorting incur the most costs in BUC's cube computation. This is because the recursive partitioning does not reduce the input size which incurs high overhead for both partition and aggregation. Furthermore, BUC is sensitive in data

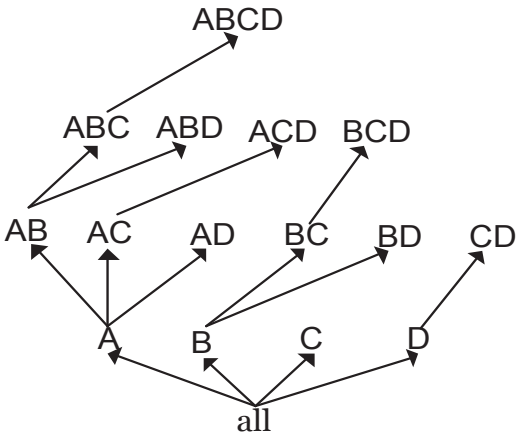


Figure 2.4: Bottom-Up Computation

skew where the performance degrades as skew increases.

2.3.3 Hybrid Cube Computation

Dong at el. [85][84] proposed a Star-Cubing method, which is a hybrid cube computation to integrate the strengths of both bottom-up and top-down cube computations and explores both multidimensional aggregation and a priori pruning. Star-cubing organizes input tuples in a hyper-tree structure, called Star-Tree. The Star-Tree is an extension of a H-Tree [33]. In H-Tree structure, each level is one dimension in the base cuboids. A d-dimension tuple forms one path of d nodes from the root and the leaves with the same value in the same level are linked together by a side-link. A head table is associated with each H-Tree to keep track of each distinct value in all dimensions and the link to the first node with that value in H-Tree.

While Star-Tree is used to represent individual cuboids in Star-Cubing, each level represents a dimension and each node represents an attribute. In steading of maintaining a side link and a head table, each node in the Star-tree has four fields including the attribute value, aggregate value, pointer(s) to possible descendant(s), and pointer to possible sibling. If the single dimensional aggregate on an attribute value p does not

satisfy the iceberg condition, the node p is replaced by $*$ so that the tree can be further compressed, since there is no need to distinguish such nodes for a Iceberg computation.

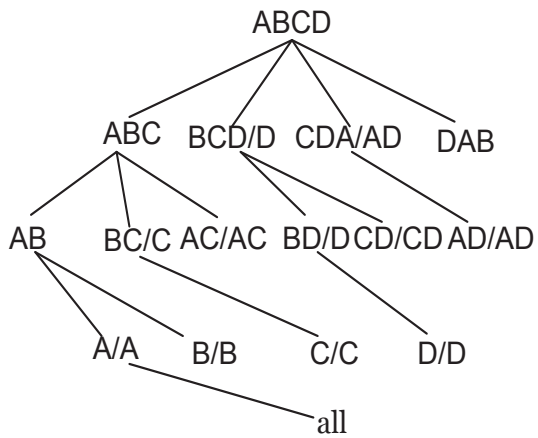


Figure 2.5: Star-Cubing Computation

The Star-Cubing algorithm explores both the top-down and bottom-up models. On the global computation order, it is similar to the top-down order as shown in Figure 2.3. However, it adopts the bottom-up model for each sub-partition tree by the **shared dimensions**. Note that the shared dimensions are defined according to the common dimensions shared in those particular sub-trees. For instance, all the cuboids in the leftmost sub-tree of the root include dimensions ABC, all those in the second sub-tree include dimensions BC and so on. Figure 2.5 shows the extended lattice with the spanning tree marked with the shared dimensions. For instance, BCD/D means cuboid BCD has shared dimension D, CDA/DA means cuboid CDA has shared dimensions DA, and so on. Since the shared dimensions are identified early in the tree expansion, the shared dimension can be computed early to share the computation. Therefore, for instance, AD extending from CDA can be pruned since AD has already been computed in CDA/AD. Given the shared dimensions, it is easy to see that, if the measure of an iceberg cube is anti-monotonic and also the aggregate value of the shared dimensions does not satisfy the condition, all the cells extended from these shared dimensions cannot satisfy the iceberg condition either.

The Star-Cubing has been evaluated to be more efficient than MultiWay and BUC. However, all the aforementioned cubing methods are designed for a centralized system, thus are not feasible for a parallel processing.

As the data size increases, a significant amount of parallel data cube research has been performed. In the following section, we review several important methods in the literature.

2.3.4 Parallel Array-based Data Cube Computation

Goil and Choudhary proposed one approach to parallelize the data cube computation in the MOLAP (Multidimensional OLAP) environment, based on the data organized in array-based structures [26] [27] [28]. In their approach, a *data partitioning* model was chosen to parallelize the data cube workload. Intuitively, they distribute each view to multiple processing units so that every processing unit computes a portion of every group-by, since it is easy to partition the array-based structures across nodes.

Specifically, in [26] [27] [28], the data is globally sorted and partitioned based on a given dimension A such that the data set is split into r partitions, P_1, P_2... P_r each of which is for one processing unit. Meanwhile, the partitioning guarantees that the value of A in any tuple of P_i is locally sorted and smaller or equal to the one in any tuple of partition P_j where $1 \leqslant i \leqslant j \leqslant r$. Note that a single value of A may straddle partitions P_i and P_{i+1}. The partial results are obtained on distributed views and may eventually be merged with the partial results on other nodes. For instance, when data is partitioned on the dimension A, then all the cuboids with A as their first dimension can be processed almost independently. This is because there is almost one set of contiguous tuples with the same value of A which can be found on different processors. For the cuboids not containing A, there is a need to merge the partial results in each node. This can be done for example through resorting and partitioning the data according to another dimension

say B, where all those remaining cuboids with B as the first dimension can be processed independently. The process is repeated d times for each distinct dimension.

This technique can reduce the re-partitioning cost. However, the total amount of data redistribution network traffic can still be quite large. Furthermore, for large data, the main memory may not be large enough to concurrently house all of the necessary arrays. Thus, arrays must be carefully partitioned and controlled. Given the complexity of this approach and the overhead incurred, it is unclear how effective this method is likely to be in terms of parallel speedup.

2.3.5 Parallel Hash-based Data Cube Computation

Lu at el. [46] present a parallel data cube implementation for the high-end multi-computer, Fujitsu AP3000. This work uses hashing for aggregation of common records, rather than the aforementioned sorting model. Here, the dimensions of each record are concatenated to form a *hash key* which is used to identify a unique *aggregation bucket*. In each aggregation bucket, the dimensions with the same value are added together. Meanwhile, if collisions occur such as two or more hash keys pointing to the same bucket, collision resolution must be employed. Hashing for data cube computation was first proposed in conjunction with the PipeHash [66]. This technique is attractive since it is not only relatively simple to implement but also bounded by $O(n)$ which outperforms the sort-based methods that typically rely on $\theta(nlogn)$ sorting algorithms. Counter-intuitively, in [66], the experimental results demonstrate that PipeSort has superiority than the PipeHash. The reason of this is because that hashing costs cannot be shared amongst child group-bys since the dimension combinations for different views are completely unique. Moreover, it is significant to choose the "constants" of the hashing with such a large number of keys. As a consequence, these two factors make the hash-based cubing algorithm slower than expected computation time.

Under this scheme, the algorithm parallelizes the computation by either (1) producing individual and partitioned hash tables by multiple processors or (2) computing groups of hash tables on individual nodes. To optimize the computation, a single common parent is used to produce all hash tables during a given iteration which means a computation round limited by the available main memory, where the group of available cuboids was chosen from a view list sorted in terms of the estimated size. The experimental results demonstrate some performance improvement on one to five processors, but no advantage beyond this point. The potential of this approach is limited by the failure to exploit smaller intermediate group-bys and the overhead of independently hashing each cuboid.

Muto and Kitsuregawa propose another more efficient parallelization technique that used a minimum cost spanning tree for hash-based cube computation [53]. In particular, their technique is to partition the individual views on a given dimension (similar to Goil and Choudhary) and then independently compute child view partitions using hash tables constructed from the smallest available parent cuboid. They also proposed the approach to balance the work load through dynamically migrating partitions from busy processors to idle ones. However, there is no physical implementation done by the authors where only simulated results are given. Furthermore, they assume that all the communication would be free since it could be completely overlapped with computation is unlikely to be borne out in practice due to the interdependencies between cuboids.

2.3.6 Parallel Top-down and Bottom-up Cube Computation

Ng at el. provide four separate algorithms designed for fully distributed PC-based clusters and large, sparse data cubes [57]. Specifically, the first two techniques are proposed based upon the bottom-up design and another two are based on the top-down design. Brief reviews are provided as follows:

- **RP (Replicated Parallel BUC):** The first technique constructs the cube from the

coarse granularity cuboids to fine granularity cuboids. It takes the lattice and carves it into d sub-trees where each sub-tree contains a set of cuboids containing the same attribute. For instance, some views contain attribute A, some contain attributed B and so on. The algorithm distributes the unevenly sized sub-trees across the network in a round robin fashion. Of course, if more than d processes exist, the extras remain idle during the computation. Not surprisingly, this leads very poor performance because of the coarseness of the partitioning.

- **BPP (Breadth First Writing, Partitioned Parallel BUC):** BPP is to partition the data across all processors. In order to avoid the excessive communication cost, the entire fact table is replicated into d distributed copies each of which is for one dimension. Unfortunately, the performance and load balancing results are only marginally better than RP. The main reason is because the costs associated with computing partitions of equivalent size vary widely due to the data skew and clustering patterns in the data set.

- **ASL (Affinity SkipList):** ASL is designed to decompose the lattice into its 2^d individual components and distribute them one by one to the best processors. Here, the best processor is the one associated with the cuboids which has been processed with the common attributes. Note that for this technique and the one below, a dynamic scheduler is adopted where a master scheduler dictates which individual view or a set of views are assigned to given processors at runtime.

- **PT (Partitioned Tree)**: PT recursively divides the lattice into subtrees which is partitioned based on a particular attribute. And then each sub-tree is assigned to an available node where the BUC algorithm is actually computed further. Similarly, to exploit computation sharing, the scheduler tries to reuse the data which has already processed on the node with some common attributes. The experimental

results demonstrate that the finer granularity scheduling of the ASL and PT provides a better load balancing and further gains a better overall performance. When the cluster has less than eight processors, the techniques gain a good speedup. However, the performance declines quickly after this point. This is because the scheduling used by these algorithms cannot capture the global cost information of the complete lattice. As a consequence, the cost reductions reduces when the workload is highly distributed since the computation of the localized view subsets is poorly coordinated.

However, all of these cubing algorithms are only developed for a cluster with a small number of machines, thus are no longer applicable under the MR framework because they are not able to leverage the full parallel power provided by MR [54].

2.3.7 Cube Computation under MapReduce

Several research works have been conducted on performing the cube computation using the MR framework. Sergey et al. [68] and You et al. [88] provided a parallel algorithm for calculating the cell closure in a closed cube computation which requires the measures to be algebraic using the MR framework. Nandi et al. [54] [55] developed a scheme to handle special holistic measures, when one reducer gets the "hot spot" group with a large number of tuples during the cube computation. In addition, Abello et al. [6] studied three different approaches to retrieve data cubes(full source scan, indexed random access and index filtered scan) from BigTable by means of MR and the definition of criteria to choose among them. However, it is still unclear how to efficiently materialize the views with a generic algorithm which can balance the load well and optimize the computation for different measures. Furthermore, none of existing works provided any solution for view maintenance when new data is inserted to update the views under the MR framework. A naive solution for view maintenance mechanism can result in

significant overhead.

Even though there are some research efforts on incremental computation in MR, none of them provided explicit support and techniques for data cube analysis under the OLAP and data warehousing semantics. For instance, existing works [12][36][43] have studied some techniques for incremental computations for single operators in the MR framework. HaLoop [14] is designed to support iterative operations through a similar caching mechanism. Restore [23] is developed to keep the intermediate results (either the output of one MR job or the data operated within one job) to the DFS in a work flow and reuse them in the future. However, none of these techniques can be directly used in data cube analysis for efficient view maintenance.

Therefore, this calls for a new generic cubing approach and efficient cube maintenance techniques towards a large amount of data which can incorporate the unique feature of the MR framework.

2.4 Graph Cube Analysis

2.4.1 Graph Summarization

In graph OLAP, the aggregate graph can be considered as a summarization of the underlying graph in terms of a particular perspective and granularity. From this sense, the graph OLAP shares the similar terminology as providing a generation of summarizes of the graph as much research has devoted to. These techniques include the graph simplification, compression and summarization. For instance, [9] [56] study the problem of simplifying the graph by preserving its skeleton according to topological features. However, in these works, the attributes on vertices and edges become unimportant. In its abstract form, they are mainly working on the graph where the vertex and edge do not contain any attributes. [62] [13] aim to compress the large graphs, especially Web

graphs. However, these works only focus on how the Web information can be stored and thus facilitate an efficient computation such as pageRank, which do not give any insight of into the graph structures.

Tian et al. and Zhang et al. proposed approaches to summarize the large graphs [73][89]. Two operations are provided including the SNAP (Summarization by Grouping Nodes on Attributes and Pairwise Relationships) operation and a less restrictive k-SNAP operation. In many applications, graphs are so large that it is almost impossible to understand the information encoded in them by mere visual inspection. Therefore, graph summarization becomes an effective way to help users extract and understand the underlying information.

The SNAP operation is to summarize the input graph by grouping nodes based on user-selected node attributes and relationships which, as the author mentioned, is similar to OLAP-style aggregations [73][89]. The k-SNAP operation allows users to provide the k value to cluster the graph into k subgraphs [73][89]. It achieves the drill-gown or roll-up operation by increasing and decreasing k. However, the graph OLAP performs the aggregation and OLAP along the dimensions defined upon the graphs.

2.4.2 Graph OLAP

Much research work [17][18][61] has been devoted to put graphs in a multi-dimensional and multi-level OLAP framework. In [17][18], graph OLAP is classified into two sub cases: informational OLAP and topological OLAP , where the informational OLAP has been mainly discussed. Intuitively, the informational OLAP works on a set of snapshot graphs. For instance, in academia, the publications in each year can be considered as a snapshot graph which consists of authors and papers and their relationships etc. The informational OLAP is to aggregate the publication networks in each year into an aggregate network. Note that the aggregate network contains the same number of objects

as in each snapshot. In other words, it is like overlaying multiple pieces of information, remaining the objects whose interactions are being looked at.

In contrast to informational OLAP, the topological OLAP operates on nodes and edges within an individual graph [61]. It groups the objects into a super object based on specific dimensions. For instance, in the publication network, the authors from the same institution may be grouped together to generalized the network. Therefore, the aggregate network contains less number of objects. The authors have introduced the concepts of conducting the OLAP operations like roll-up, drill-down and slice/dice based on different graph OLAP sub cases.

The existing works provided a high level discussion of building a graph OLAP framework and graph OLAP operations on graphs. However, no specific graph cube models and cube materialization algorithms are provided.

2.4.3 Graph Cube on Multidimensional Networks

Subsequently, Zhao et al. proposed a Graph Cube model for multidimensional networks [94]. The Graph Cube model is designed on the networks where each vertex contains a set of multidimensional attributes but the edges are identical without attaching any attributes.

Given n attributes attached with each vertex, graph cube generates 2^n cuboids each of which is an aggregate graph based on specific dimensional attributes. Besides the cuboid query, the authors also provide a new set of queries which are defined as crossboid query which crosses multiple multidimensional spaces of the network.

However, the graph cube model is highly restricted to the multidimensional networks where the edge does not contain any attributes. In the real world, a lot of information networks are attributed graphs where both the vertex and edge contain attributes. Building a data warehousing model based on the attributed graphs are more challenging and

important.

On the other hand, normally, the information network are large where the algorithm designed on the single machine is not able to provide acceptable performance. However, non of the existing works have provided any parallel and distributed solutions on graph OLAP.

Therefore, we are motivated to design a new and more general graph cube model based on the attributed graphs, and develop scalable, effective and efficient parallel and distributed graph OLAP techniques in order to meet the requirements of large-scale graphs in real applications.

CHAPTER 3

COMBINATORIAL STATISTICAL ANALYSIS

3.1 Overview

In this chapter, we address the problem of building parallel solutions for one computation extensive analysis, combinatorial statistical analysis (CSA). CSA has been widely used to find the significant correlations that are typically measured by statistical methods among different objects. Intuitively, CSA evaluates the significance of the associations between a combination of objects by adopting the statistical methods, such as χ^2 test. For illustration, we address the problem by taking the epistasis discovery as an example, where the CSA has been widely adopted. Although we have chosen epistasis discovery for demonstration, our solution is not specific to just this domain, and should apply broadly to all the CSA applications.

In this work, we propose a framework for efficient COmbinatorial Statistical Analy-

sis systems MapReduce(MR)-based Cloud platforms (COSAC). COSAC addresses the CSA problem in two phases: 1) Distribution Phase: We develop and compare different task distribution schemes to enumerate the large number of combinations to the processing units in terms of balancing the load. Given a total number of n objects, in order to find the associations among any m objects, there are total $C(n, m)$ combinations to evaluate. The scheme partitions the enumerated combinations into *n-m+1* sets, each with a different number of combinations. These sets of tasks are then distributed to the processing units to balance the number of combinations across the units. 2) Statistical Analysis Phase: Each node has to evaluate the statistical significance of the combinations allocated to it. We develop an optimization to salvage the common computations between the various combinations and provide a technique called Integer Representation and Bitmap Indexing (IRBI) to speed up the statistical testing. Such two phases have solved the two key challenges in CSA including balancing the load to each processing units in a distributed environment and conducting an efficient statistics testing.

The COSAC framework includes three layers. The first layer is the index builder layer which is used to preprocess the raw data to facilitate efficient data processing. The second one is the analysis layers for parallel combination enumeration and statistical analysis. Two analysis schemes have been proposed, *Exhaustive Testing* and *Semi-Exhaustive* Testing. The *Exhaustive Testing* supports exhaustive evaluation of the statistics significance of all the combinations without losing any significant result. The *Semi-Exhaustive* can be used to analyze part of the combinations to prune the computation spaces. The third layer is the top-k retrieval layer that is designed to help users to further retrieve the top-k most significant results from the large volume of analysis results data.

Based on COSAC, we have designed and compared various flexible object combination enumeration schemes with regard to load balancing and scalability for large scale

of datasets using the MR paradigm. The enumeration of combinatorial objects takes an important role in computer science and engineering. We also propose the techniques to use the integers to represent the long string raw data and adopt Bitmap index to index each object on the samples. Thus, we can conduct the analysis only based on the representation data and index data. Both of these two optimizations are memory-efficient, CPU-efficient and contribute the efficient statistical testing. Furthermore, we study how to salvage the computation for a sharing optimization with significant performance savings, instead of conducting the testing for each combination independently. Extensive experimental evaluations have been conducted and the results indicate that our framework is computationally scalable, efficient and practical.

The rest of this chapter is organized as follows. In the next section, we provide some preliminaries about epistasis discovery. In section 3.3, we provide the main architecture of our proposed framework. Section 3.4 introduces our approach for preprocessing the raw data and how to make efficient statistical testings using our transformed data within one combination. Section 3.5 presents the task distribution models. In Section 3.6, we describe the strategy of combination enumeration with sharing optimization for the given task in each processing unit. Section 3.7 reports the experimental results. Finally, we summarize this work in Section 3.8.

3.2 Preliminaries

In this section, we first review some preliminaries on epistasis discovery.

In genome-wide association study (GWAS), there are two types of data - genotype data that codes the genetic information of each individual like SNPs and phenotype data that measures the quantitative traits for an individual such as diseases. Epistasis discovery aims at discovering significant correlation between the SNPs and phenotypes

(diseases) and is becoming increasingly important and challenging in GWAS. It plays an important role in addressing the complexity of many common human diseases (e.g., heart attack, breast cancer and diabetes). Thus, finding these interactions is a first step towards understanding the cause of these diseases.

Figure 3.1 shows an example of the kind of raw dataset that we are dealing with in epistasis discovering. Each row contains the individual sample information with the sample id, the genotypes of multiple objects (SNPs) and phenotype (disease type). The first and last columns are the sample id and phenotype. The rest of the columns are the genotypes of each object. In this example, we have 6 objects (SNPs) and 8 individual samples. The data are the genome information from two kinds of patients, each of which has either breast cancer or heart attack. Each object is marked as a bi-allelic value (i.e. a locus has allele A and T) and has three genotype candidates, AA, AT and TT. Thus, given n objects and L individual samples, the data can be considered as a table with $n+2$ columns and L rows.

The goal of epistasis discovery research is to identify a set of most significant epistatic interactions (i.e., combinations of multiple objects (SNPs)) that correlate to the phenotypes. This means enumerating all combinations of a particular analysis level, and then identifying those that are significant. Statistics methods have been used as powerful measure tools for evaluating the significance of their correlation. Most of the widely used

Sample Id	1	2	3	4	5	6	Phenotype
1	AT	TT	AT	TT	TT	TT	Breast Cancer
2	AA	AA	TT	AA	AT	AA	Heart Attacks
3	TT	AA	AA	AT	AA	TT	Breast Cancer
4	AT	AA	AT	AA	AT	AA	Heart Attacks
5	AT	AA	AT	TT	AT	TT	Breast Cancer
6	AA	AA	TT	AA	AA	AT	Heart Attacks
7	AT	TT	AA	TT	AA	AA	Heart Attacks
8	AA	TT	AA	AT	AT	TT	Heart Attacks

Figure 3.1: An example of the raw data with 8 samples and 6 objects.

Table 3.1: Contingency Table

	0,0	0,1	0,2	1,0	1,1	1,2	2,0	2,1	2,2
Heart Attack	#	#	#	#	#	#	#	#	#
Breast Cancer	#	#	#	#	#	#	#	#	#

statistical methods measure the significance of the interrelation based on the *contingency table*, e.g. χ^2-test, Likelihood Ratio, Normalized Mutual Information, Uncertainty coefficient, Odds Ratio and Armitage Trend Test [30]. The contingency table is used to record and analyze the relation between two or more categorical variables and display the frequency distribution of the variables.

In this work, we adopt the χ^2-test[10], which is widely used, to illustrate how the significance of the correlation of a combination of SNPs can be computed from the information stored in the contingency table.

Take 2-level epistatic interactions as an example. Let $n_{0(j,k)}$ denote the number of samples in the "Heart Attack" group whose first locus's genotype code is 'j' and second locus's genotype code is 'k', where j and k take on values 0, 1 or 2 (corresponding to AT, AT or TT in the raw data). Likewise, we can denote $n_{1(j,k)}$ for the "Breast Cancer" group. For 2-level epistasis discovery, with these two groups of information, we can derive a 2×9 contingency table - as each object (categorical variable) has 3 possible candidate values, the contingency table has 9 (3*3) columns; furthermore, as there are two groups of patients, the contingency table has 2 rows as shown in table 3.1. We can calculate the χ^2-test value of this epistatic interaction from this contingency table using the following formula:

$$\chi^2 = \sum_{i=0}^{1}\sum_{j=0}^{2}\sum_{k=0}^{2}\frac{(n_{i(j,k)} - n_i n_{(j,k)}/n)^2}{n_i n_{(j,k)}/n}$$

where $n = \sum_{i=0}^{1}\sum_{j=0}^{2}\sum_{k=0}^{2} n_{i(j,k)}$, $n_i = \sum_{j=0}^{2}\sum_{k=0}^{2} n_{i(j,k)}$, and $n_{j,k} = \sum_{i=0}^{1} n_{i(j,k)}$.

The null hypothesis behind the χ^2-test is that there is no association between two-

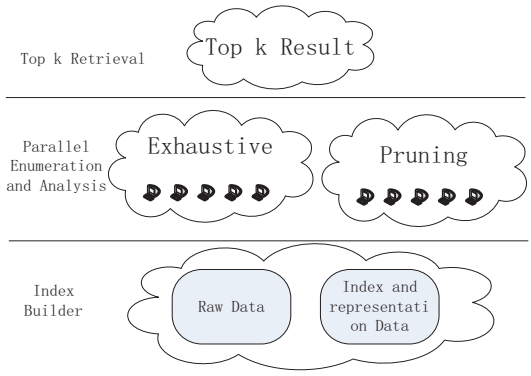


Figure 3.2: *COSAC* framework architecture

locus epistatic interaction and phenotype. As the χ^2-test statistics follows the χ^2 distribution, thus the corresponding significance level can be obtained after Bonferroni correction. The lower the value is, the more confident we are to reject the null hypothesis. The resultant p-value for the 2-level epistatic interaction can be obtained as P(x>C) where C is the χ^2-test value, and P(x) is the probability at value x under the χ^2 distribution. The smaller the p-value is, the more significant the combinations is. The above expressions can be easily generalized for any k-level interaction. We shall omit this discussion here. It suffices to mention that, in general, for a k-level epistatic discovery, we will have a 2×3^k contingency table.

3.3 The *COSAC* Framework

Figure 3.2 shows the architecture of our proposed *COSAC* framework. It has an index builder (the bottom part), a parallel enumeration and analysis component (the middle part) and a top K most significant result retrieval component (the top part).

Index Builder: The Index Builder is used to preprocess the raw data to facilitate efficient data processing. On the one hand, we propose a technique to use simple integers to represent the long data values (e.g. strings) to reduce the data size. On the other hand, we build the Bitmap index [15] for each object to speed up the statistical testing which

is similar to the mechanisms we adopted in [81] where bit strings are used. The resultant data representation can both speed up statistics testings and minimize the data size and memory utilization.

Parallel Enumeration and Analysis: The *COSAC* framework supports parallel combination enumeration and statistics analysis on the MR framework. As we shall see in our experiments, *COSAC* is scalable, flexible and efficient. Moreover, it inherits the fault tolerance feature of MR infrastructure. *COSAC* consists of two key components: a global load distribution scheme that distributes combination enumeration tasks across the processing nodes, and a local processing scheme that optimizes the statistical analysis of the combinations assigned at each node. *COSAC* supports both exhaustive and semi-exhaustive enumeration. The *Exhaustive Testing* can be used to exhaustively analyze the statistics significance of all the combinations of objects without losing any significant result. The *Semi-Exhaustive Testing* supports users to analyze part of the combinations to prune the computation spaces. For example, it is not uncommon for scientists to predetermine a subset of SNPs that they are interested in, and then focus their study on these subsets alone or against a superset of SNPs. In this case, there is no need to analyze the significance of all the combinations.

Top-k Retrieval: In our system, users can choose to output all the analysis results or the ones above some threshold set by users. As the computation is expensive in these systems, users may want to do one time computation and multiple observations on it. We provide an efficient top-k retrieval technique to help users to further retrieve the top-k most significant results from the large volume of analysis result data.

o_ID	PT	GT	list
1	Heart Attacks	AA	8, 6, 2
1	Heart Attacks	AT	7, 4
1	Breast Cancer	AA	5
1	Breast Cancer	AT	1
1	Breast Cancer	TT	3
2	Heart Attacks	AA	6, 3, 2
	...		
6	Breast Cancer	TT	5, 3, 1

(a1)

o_ID	PT	GT	Bitmap Index
1	0	0	10100010
1	0	1	01001000
1	1	0	00010000
1	1	1	00000001
1	1	2	00000100
2	0	0	00100110
	...		
6	1	2	00010101

(b1)

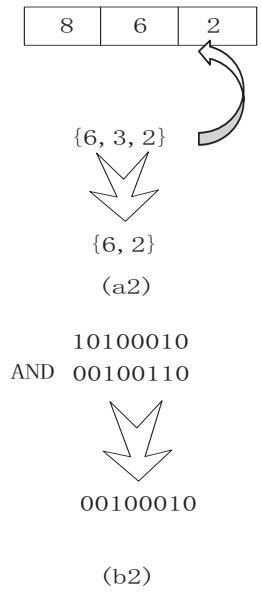


Figure 3.3: Data reformation and contingency table construction

3.4 Efficient Statistical Testing

To enumerate the combination of multiple objects, we should collect the information for single object first. A naive approach, as used in [80], is to preprocess the raw data to each object according to the genotype and phenotype with a sample id list. Figure 3.3 (a1) displays the data format after preprocessing. The data in the first column is the object id. The second and the third columns are the phenotype and genotype in this object. The last list is used to store the sample ids whose related object has the same phenotype and genotype as shown in the second and third columns. Therefore, all the rows with the same object id in Figure 3.3 (a1) belong to a single object.

However, the above preprocessing technique is not only inefficient with regard to statistical testing, it is also not memory efficient. Therefore, we propose a new technique, called IRBI (*Integer Representation and Bitmap Indexing*), which is both CPU-efficient with regard to statistical testing, and storage and memory efficient.

Instead of operating on the original raw data, the IRBI method uses simple integers

to represent the long string data values in the raw data. Furthermore, IRBI builds the Bitmap index for each object to facilitate CPU-efficient statistical testing. Considering the example in Figure 3.1, we use 0 to represent the phenotype value "Heart Attacks" and 1 to "Breast Cancer". For the genotype data, AA, AT and TT are represented as 0, 1 and 2 respectively. It is important to note that the number of objects is large in these systems and a lot of terms use long string values, such as "Pineoblastoma and supratentorial primitive neuroectodermal tumors". Our adopted IRBI method can largely reduce the data size.

To collect single object information, we build a Bitmap index for each object (each column on the table) based on the phenotype and genotype. Each bit in the Bitmap index corresponds to a sample id. For one given phenotype and genotype, the corresponding positions in the Bitmap index are set to 1 if the samples have the same phenotype and genotype as the given ones. Otherwise, they are set to 0. Figure 3.3 (b1) shows the index data format under IRBI. For example, the first five rows are the index data for the first object. As the phenotype and genotype of the sample ids 2, 6 and 8 are 0 and 0, the index data is "10100010" as shown in the first row in the new formatted data. The corresponding positions for 2, 6 and 8 are therefore 1 in the index data "10100010". Thus, the IRBI approach reduces the data sizes and memory utility.

Now, recall that in the statistical analysis, the contingency table has to be collected for each combination.

To construct the contingency table, the first step is to calculate the $n_{i(j,k)}$ for each grid in the table. If we want to calculate the $n_{i(j,k)}$ for the pair of object x and object y, we need the information from $\langle x, i, j, list1(sampleID)\rangle$ in object x and $\langle y, i, k, list2(sampleID)\rangle$ in object y. We can derive $n_{i(j,k)}$ from the intersection between the two sample id lists. Under the naive scheme (without Bitmaps) as shown in Figure 3.3 (a2), this can be easily done using a hashing method - first, we build a hash table for the sample ids in the first

list; second, we use the sample ids in the second list to probe the hash table for matching sample ids. For example, to get $n_{0(1,0)}$ for the pair of object 0 and object 1, we intersect the two sample lists as shown in Figure 3.3 (a2). However, our preliminary study suggests that using such an approach to collect the contingency table is computationally expensive.

In our *COSAC* framework, our solution employs the Bitmap index. As mentioned above, instead of storing the sample ids in the list, we build the Bitmap index for each object. Figure 3.3 (b1) is the Bitmap index data for the raw data in Figure 3.1. In our framework, all the operations are based on the index data. With the index data, if we want to calculate the $n_{i(j,k)}$ for the pair of object x and object y, we need the information from $\langle x, i, j, index \rangle$ in object x and $\langle y, i, k, index \rangle$ in object y. We can conduct an AND operation on the two index data to find the intersection between them more efficiently. We can easily get the number of intersection samples from counting the 1's bits from the AND result. Figure 3.3 (b2) depicts how $n_{0(1,0)}$ for the pair of object 0 and object 1 can be calculated using the Bitmap index. Thus, the IRBI approach is much more CPU-efficient than the naive scheme to collect the data (known as contingency table) during statistical testing. When we collect the contingency table for more than 2 objects, similar operations can be conducted. For example, to combine 3 objects, we can combine 2 objects first and then combine the result with the third object.

Now, we introduce how to efficiently build the Bitmap index under MR framework. In our *COSAC* framework, the index building is conducted in one MR job where the `map` phase parses the objects from each sample and the `reduce` phase builds the index for each object. Take the data in Figure 3.1 as an example, the `map` phase reads the raw data line by line, each of which is an individual sample data. The `map` function parses the different objects in each line and emits (key,value) pairs (each for one object) which includes all other necessary information for each object like sample id and phenotype

data. Our own partitioning function partitions the pairs based on the object and the MR library shuffles all the (key,value) pairs belonging to the same object into a designated `reducer`. Once the `reduce` function gets all the information for one object, it builds the index based on the sample ids related to this object, and then writes the index data into the HDFS.

One optimization we have adopted is to combine the small index data into a big file. From our observation, if we store the index data for each object in a file, there are too many small files. During further processing, the MR library assigns each small file to one `Mapper` which brings too much overhead for launching a large number of mappers. Thus, we provide one optimization technique to combine all the index data emitted from one `reducer` into one big file. This will highly reduce the overhead for further processing on the index data. Note that the integer representation is also conducted in the `map` phase to build a one to one mapping between the raw data to the integer values.

For *CSA* applications, the Bitmap index is an effective structure. First, the operations are typically read-mostly. There are very few update operations. So there is no need to change the Bitmap index frequently. Second, each object has few candidates. In other words, each column has a small domain (i.e., there are few distinct values). Thus, the Bitmap index will not be too sparse. Last, the number of samples (rows) is small which guarantees that the Bitmap index is not large. We note that index building is an efficient, cheap and one-time fixed pre-processing step in our framework.

Discussion: In a broader context, our proposed technique above can be widely used in many different applications besides the *CSA* applications . On one hand, the integer representation approach can be used in many computation intensive analysis applications to reduce the data size and results in a more memory efficient computation. On the other hand, the proposed Bitmap index approach can be used in the systems which

Table 3.2: Frequently Used Notations in COSAC

Symbol	Description
n	the number of total objects
s	the number of seed objects
n-s	the number of non-seed objects
m	the analysis level
r	the number of `reducers`(processing units)

use statistics methods as the evaluation tool. It is expensive to conduct a large number of statistical testings, while the technique we adopted can be a promising solution for efficient statistical testings.

3.5 Parallel Distribution Models

In this section, we introduce how our framework can enumerate the combinations precisely and balance the load well based on the MR framework for large scale of objects. We propose promising parallel distribution models for both *Exhaustive Testing* and *Semi-Exhaustive Testing* with good load balancing. All the frequently used notations are provided in table 3.1.

3.5.1 Exhaustive Testing

Exhaustive testing scheme is used to exhaustively enumerate all the combinations of objects and analyze the statistical significance of each combination. The formal definition of *Exhaustive Testing* is given as follows.

Definition 3.1. ***(Exhaustive Testing)*** *Given* n *objects for m-level analysis, exhaustive testing enumerates and evaluates the significance of all the combinations of size* m*. Therefore, the total number of combinations that should be enumerated and evaluated is* $C(n, m) = \frac{n!}{m!(n-m)!}$.

Now we are ready to introduce how the *COSAC* framework supports the *Exhaustive Testing* scheme well. Numbering the *n* objects from *1* to *n*, the total $C(n, m)$ combinations can be divided into *n-m+1* sets each of which involves all the combinations starting with a fixed object *i*. For instance, the combinations of *C(5,3)* are: $\{$123, 124, 125, 134, 135, 145, 234, 235, 245, 345$\}$. These combinations can be divided into three sets: Set_1 starts with fixed object 1 $\{$123, 124, 125, 134, 135, 145$\}$, Set_2 starts with fixed object 2 $\{$234, 235, 245$\}$ and Set_3 starts with fixed object 3 $\{$345$\}$.

To enumerate these *n-m+1* sets of combinations correctly, our proposed schemes try to distribute the tasks among multiple processing units with a good load balancing. Note that the number of combinations in Set_i is *C(n-i, m-1)*. Therefore, the number of combinations from Set_1 to Set_{n-m+1} decreases.

To distribute the *n-m+1* sets into *r* processing units, a very naive approach is to simply distribute approximately equal number of rows to each `reducer`. The middle of Figure 3.4 depicts all the *n-m+1* sets of tasks where the amount of computation tasks increase as the set number decreases. In particular, the first $\frac{n-m+1}{r}$ sets are assigned to the first `reducer`, the next $\frac{n-m+1}{r}$ sets are assigned to the second `reducer` and so on. However, such a naive solution will result in load-imbalance as some `reducers` are more heavily loaded than others, e.g., the `reducer` assigned the first $\frac{n-m+1}{r}$ rows is likely to be a bottleneck. Hence, in this work, we study three alternative different distribution models including the Greedy, Bestfit and Round Robin. We introduce the distribution strategy of each model as follows:

Greedy Model. Ideally, each `reducer` should process $\frac{C(n,m)}{r}$ computations. Therefore, starting from the first row, we seek to allocate consecutive sets to a `reducer` such that the total number of computation tasks for these rows is closest to $\frac{C(n,m)}{r}$. The RHS of Figure 3.4 show that, under the greedy scheme, each `reducer` may be assigned different number of sets to process. For instance, Set_1 and Set_2 are assigned to the

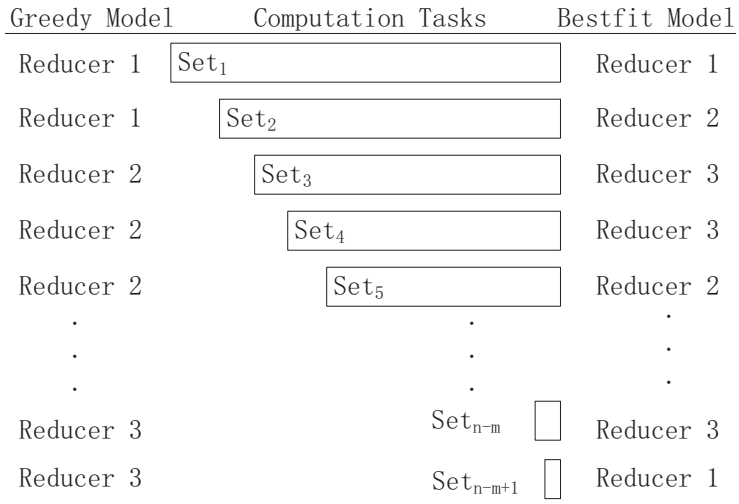


Figure 3.4: COSAC: Parallel distribution models for Exhaustive Testing

first reducer, while Set_3 Set_3 and Set_5 are assigned to the second reducer and so on. However, since the ith set has a large number of computation tasks when i is small, our experimental results indicate that this strategy may get a better balanced load when m is small, say 2. However, when m increases, it is hard to balance the load.

***Bestfit Model*:** In **Bestfit** model, the sets are assigned to each processing unit from the first to the last set. Instead of assigning consecutive sets to a unit, we seek to allocate the next available set to the unit which has the smallest amount of task. For instance, as shown on the RHS of Figure 3.4 the first 3 sets (from Set_1 to Set_3) are allocated to each of the 3 reducers one by one. And Set_4 will be assigned to one of the 3 reducers which has the smallest number of enumerating combinations (the 3rd unit in this example), Set_5 will be assigned to node 2, and so on. Note that the number of combinations in the assigning sets is in a decreasing order which means the task set is becoming smaller and smaller. This guarantees that it is easier to achieve the ideal load balancing. From our experimental results, we will see that the **Bestfit** model has excellent load balancing and outperforms the **Greedy** model in deeper analysis cases.

In this distribution model, it not only provides a precise enumeration mechanism but

also balances the load well across the nodes in a parallel environment. It is also worthy to note that the number of objects we are facing is always large, mostly tens of thousands, even millions. Using a very large number of machines to process a very small number of objects is a very rare case in reality. This further provides the guarantee of our proposed **Bestfit** model to achieve good load balancing in practice.

Round Robin Model: Round Robin model is another seemingly better load balancing model which assigns the combinations in a round robin fashion to each `reducer`. Recall the example in the Figure 3.4, given $C(n, m)$ combinations and r processing reducers, there are two variants of round robin implementations. The first one assigns the next available combination to the reducer with the smallest assigned load. We refer to this implementation as RR_I. The second one assigns the first $\lfloor \frac{C(n,m)}{r} \rfloor$ combinations to the first reducer, the second $\lfloor \frac{C(n,m)}{r} \rfloor$ combinations to the second reducer and so forth. And the unassigned $C(n, m)\%r$ combinations are assigned to any $C(n, m)\%r$ reducers. We refer to this implementation as RR_II.

In this distribution model, the load is better balanced because the difference between each `reducer` is at most one combination. However, we do not think this will improve the whole performance much as it needs to maintain much more meta information which brings a lot of other overheads in the `reducer`. One of the main overheads is the searching overhead at the `reducer` which has to store the meta information to identify which objects should be combined together. More importantly, under the **Bestfit** scheme, there are more opportunities to share results of partial computation, which we will further elaborate in section 3.6. The small benefit of round robin via load balancing is less than what we can benefit from the **Bestfit** model. More discussion and experimental evaluation for **Round Robin** can be found in section 3.7.

Now we are ready to see how to perform this under the MR model where the pseudo code is listed in Algorithm 3.1.

Algorithm 3.1: **Performing *CSA* in MR**

1 **Map Phase:**
2 *Map Function:* Input: $< \emptyset,\ o\_Tuple >$
3 # *o_Tuple:* partial information of one object, like each row in Fig. 3 (b1)
4 *o_ID* $\Leftarrow$ *getObjectID*(*o_Tuple*);
5 $v \Leftarrow$ *getRestInfor*(*o_Tuple*);
6 **for** *each reducer* i_ID *that needs the object under the distribution model* **do**
7 $k \Leftarrow$ *o_ID.r_ID*;
8 emit $< k, v >$;
9 **Shuffle Phase:**
10 *Partitioning Function:* Input: $< k, v, r >$
11 Return *getReducerID(k)*;
12 **Reduce Phase:**
13 *Reduce Function:* Input $< k, (v_1, v_2, ..., v_m) >$
14 $obj \Leftarrow$ Assemble the value into one complete object;
15 Put obj into a list;
16 *Close Function:*
17 Call *task distributor* and start enumeration; # See Algorithm 3.4

Map Phase: Each `mapper` reads a chunk of input data (the index and representation data) by lines. Note that each line is the partial information for one object as shown in Figure 3.3 (b1). For each line, it then determines the `reducers` which this partial object information should be shuffled to according to the distribution model. Note that each object may be shuffled to multiple `reducers` and each `reducer` only needs one copy of the objects that it needs to process. However, MR only supports shuffling one output (key,value) emitted from one `mapper` to one `reducer`.

We resolve this problem by replicating and emitting as many copies of an object as required (Lines 6-8). For instance, if an object needs to be shuffled to *r* reducers, we output *r* (key,value) pairs in the `mapper`. In addition, each such pair is "tagged" with the corresponding `reducer` identifier, *r_ID* to distinguish the `reducer` that the pair should be shuffled to. Specifically, for each `reducer` which an object should be shuffled to, we generate and emit a (key, value) pair where the key is set as *"o_ID.r_ID"* where o_ID is the object ID. The result of the information will be stored into the value

(Line 4,5).

Shuffle Phase: We implement our own partitioning function to partition the intermediate results ((key, value) pairs) according to the r_ID in the key (Lines 10-11).

Reduce Phase: The MR library sorts and merges the intermediate results based on the key. All the (key, value) pairs with the same key will be grouped together. Therefore, all the tuples from the same object will be grouped together and passed to the *reduce* function in one iteration (Line 13). The *reduce* function can get one complete object information in each iteration (Line 14). In addition, because the (key, value) pairs in each reducer have the same *r_ID*, all the pairs are sorted by the object id. The objects are supplied to the `reduce` function one by one in order and stored in a list (Line 15). Note that the list is also in a sorted order with regard to the object id. In the *reduce* function, all the objects are received and stored in a list. The *Close* function is called after the *reduce* function. In the *Close* function, we develop a `task distributor` which is used to to get the computation tasks that it needs to process under different distribution models. For instance, the reducer will get all the set ids which it needs to execute. After getting the task, the `reducers` can do the enumeration and analysis in parallel (line 17). In section 3.6, we will introduce how *COSAC* can efficiently do the enumeration and statistical testing.

3.5.2 Semi-Exhaustive Testing

While facing a large number of objects, users may want to reduce the computation space to speed up the testing. For instance, users may only be interested in several objects, which we shall refer to as *seed* objects in this thesis, and whether they have any significant correlation with the other *non-seed* objects.

Now we formalize the pruning search requirement in data analysis systems.

Definition 3.2. *(**Semi-Exhaustive Testing**) There are two kinds of objects including seed*

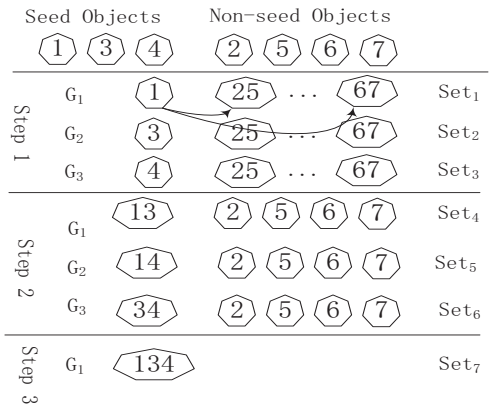


Figure 3.5: Combinations enumeration in Semi-Exhaustive Testing.

objects (the user interested objects) and non-seed objects (the rest objects). From the seed and non-seed objects, we only enumerate and evaluate the significance of the combinations involving at least one seed object.

Under *Semi-Exhaustive Testing* scheme, all the combinations should be enumerated are the ones including *i* seed objects and *m-i* non-seed objects where *i* is from *1* to *m*. Therefore, the total number of combinations we should test is as follows:

$C(s,1)*C(n-s,m-1)+C(s,2)*C(n-s,m-2)+.....+C(s,i)*C(n-s,m-i)+...+C(s,m-1)*C(n-s,1)+C(s,m)*C(n-s,0)$

where *C(s,i)* means the number of combinations of selecting *i* seed objects and *C(n-s,m-i)* is the number of combinations of selecting *m-i* non-seed objects.

The enumeration task can be divided into *m* **steps** of enumerations where the ith step enumerates all the combinations with *i* seed and *m-i* non-seed objects respectively. For instance, assume that are 3 seed objects (1, 3, 4) and 4 non-seed objects (2, 5, 6, 7). Figure 3.5 lists all the computation task of enumerating the combinations in 3-level analysis. In each step, the enumeration task can be conducted in two **phases** as follows.

- *Phase 1:* Get all the combinations with *i* objects from the seed objects. We refer to these seed combinations as $seed\_combs$.

- *Phase 2:* Use the $seed\_combs$ to further combine with the non-seed objects.

In Figure 3.5, the left part under seed objects lists all the $seed\_combs$ in each step as mentioned in the first phase. The right part under the non-seed objects lists all the non-seed combinations that the $seed\_combs$ should further combined with as required in the second phase. For demonstration, the enumeration task in each line is referred to as one set and numbered from top to bottom as shown in the RHS of Figure 3.5.

Thus, there are $\sum_{i=1}^{m} C(s,i)$ sets of tasks (7 in the example). In each step, we assign a **group id** for each line according to an increasing numeric order for the $seed\_combs$. For instance, in step 2, the group id of 13 is marked as 1, 14 and 34 are marked as 2 and 3. This generates a mapping between each group id and $seed\_comb$.

To distribute these tasks to multiple processing units, we propose a distribution model, **BestfitPru**, which is in the same spirit of **Bestfit** model. In **BestfitPru** model, the $\sum_{i=1}^{m} C(s,i)$ sets are assigned to different processing units based on the size of the set in a decreasing order. The next available set is assigned to the processing unit which has the smallest processing task. Note that when the set id increases, the number of combinations in the set decreases. In other words, the **BestfitPru** model distributes the tasks from the sets with a larger number of tasks to the ones with a smaller number of tasks. The methodology after this distribution strategy is to achieve a better load balancing.From the experimental results, we have observed that **BestfitPru** has balanced the load well.

Now we introduce how the **BestfitPru** model can be performed under the MR framework. Since the pseudo code is similar to Algorithm 3.1, we shall omit it here.

Map Phase: For each object information, the `emitter` determines its object class (seed or non-seed). Here, we shuffle each object information to all `reducers`. Therefore, for each object, *r* (key,value) pairs are emitted and each of the pairs will be shuffled to one `reducer`. Each pair is also "tagged" with the corresponding reducer identifier to distinguish the `reducer` that the pair should be shuffled to. The key format is set as

Algorithm 3.2: **Semi-Exhaustive Testing** ***task distributor***

```
Input: Id of current reducer: curR ,
Number of seed objects: s,
Number of non seed objects: n-s,
Number of objects in each combination: m
1 for step id i:1 → m do
2     numOfSets ⇐ C(s,i);
3     for group gID:1 → numOfSets do
4         minR ⇐ getMinRed(arrSummary);
5         arrSummary[minR] ⇐ arrSummary[minR] + getComNum(m-i, n-s);
6         if minR == curR then
7             Get seed combination in terms of i and gID # See Algorithm 3.3
8             Start enumerating and evaluating the significance # See Algorithm 3.4
```

"*r_ID*|*o_Class*|*o_ID*" where *r_ID* is the identifier of which `reducer` this pair should be shuffled to, *o_Class* is the object class identifier and *o_ID* is the object id. In our system, we use 0 as the class identifier for the non-seed object and 1 for the seed object. And the rest of the object information is stored in the value. We note that there are many ways of setting the format of key and value. The reason using this format is because, under this case, all the (key,value) pairs shuffled to the same `reducer` have the same *r_ID*. After sorting, all the non-seed objects will be supplied to the reduce function earlier than the seed objects, as the *o_Class* 0 for non-seed objects is smaller than 1 for seed objects. Therefore, it is easy to parse and separate seed and non-seed objects in the reduce phase.

Shuffle Phase: Our own partitioning function partitions the intermediate (key, value) pairs based on the *r_ID* in the key.

Reduce Phase: In the reduce function, it gets all the each `reducer` can get all the seed and non-seed objects. The objects are stored into two different lists each of which for one object class. It is important to note that each `reducer` maintains these two exactly same lists as well as their storage order in the list. In the *Close* function, each reducer gets its own enumeration and analysis task from the `task distributor`.

Algorithm 3.2 outlines the main steps of how `task distributor` distributing the

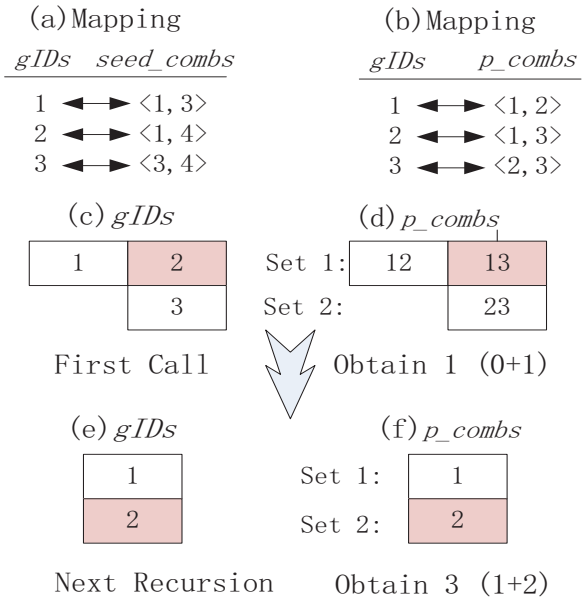


Figure 3.6: Converting the group id to the position combination

task to each `reducer` according to the **BestfitPru** model. The `task distributor` sends out the enumeration task from step *1* to *m*. In the ith step, Algorithm 3.4 first calculates the number of groups in this step, which is *C(s,i)*. During task assignment, the next unassigned group will be assigned to the `reducer` which currently has been assigned the smallest number of computation task. Here, a data structure like an array is chosen to store the tasks assigned to each `reducer`. The task group is added to the `reducer` having the smallest computation task (Line 5). Meanwhile, if the smallest reducer is the current reducer, the `task distributor` invokes the enumeration and analysis task execution by passing the group id gID and step id i.

The `reducer` starts to enumerate and analyze the objects based on the given *gID* and *i*. Recall that the enumeration task can be conducted in two phases where the first one finds the *seed_comb* corresponding to *gID* in step i and the second one uses this *seed_comb* to combine with the non-seed objects. In the latter of this section, we describe the approach to get the *seed_comb* through *gID* and i for the first phase. The second phase will be introduced in section 3.6.

Finding the *seed_comb* is the same as finding the position of each seed object in the *seed_comb* in the list. The seed object position combination is referred as *p_comb*. Since the group id is assigned to *seed_combs* according to the numeric order of seed and the seeds are also sorted in the list, the relationship between group IDs and *p_comb* remains the same feature as the *seed_combs*. When the *p_comb* value increases, the group id becomes larger.

Let us still take the data mentioned in Figure 3.5 as an example. The three seed objects 1, 3 and 4 are stored in a list where their positions are 1, 2 and 3. For instance, in step 2, there are three *seed_combs* each of which is corresponding to one group id. Figure 3.6 shows an example of converting the group id 2 into the position combination where gID , $seed\_comb$ and p_comb indicate the group ID, seed object combination and the seed object position combination in the list respectively. Figure 3.6 (a) and (b) present the mappings between $gIDs$ and $seed\_combs$ and p_combs respectively. Figure 3.6 (c-f) depict the procedure of converting group id 2 into the the position combination <1,3> by two recursions of Algorithm 3.3.

Algorithm 3.3 outlines a recursive algorithm to address how to get the *p_comb* for a given gID and i. In each recursion, Algorithm 3.3 calculates one object position in *p_combs*. There are four input parameters. The first parameter, sOP, records the seed position starting to search. The second parameter i indicates the step level. gID and s are the group id and the number of seed objects.

Given i and s, there are *C(s,i)* seed combinations corresponding to *C(s,i)* p_combs. Intuitively, these p_combs can be divided into *s-i+1* sets each of which includes *C(s-setNo, i-1)* combinations with the same prefix where the *setNo* is the set number. For instance, all the three p_combs in step 2 in the example can be divided into two sets as shown in Figure 3.6 (d). Set 1 includes all the combinations with 1 and set 2 with 2. Under this, it is easy to see that the *setNo* which the *gID* falls into is the seed position.

Algorithm 3.3: **Convert from group id to combinations**

```
   Input: Start position: sOP ; Step ID: i;
   Group id: gID; Number of seeds: s
1  sum ⇐ 0;
2  for setNo: 1 → s-i+1 do
3      setSize ⇐ C(s-setNo, i-1);
4      if gID <= total+setSize then
5          newOP ⇐ sOP+setNo;
6          # Emit the new position newOP
7          if i > 1 then
8              s ⇐ s-setNo;
9              gID ⇐ gID - SUM;
10             Converter(newOP, i − 1, gID, s);
11     else
12         sum ⇐ sum+setSize ;
```

Therefore, Algorithm 3.3 first determines which set the *gID* allocates (Line 4). After getting the *setNo*, the seed object position *newOP* can be simply obtained by *sOP*+*setNo* (Lines 7-8). If the step id is bigger than 1, the algorithm goes to next recursion with new input parameters, where *sOP*, i, *gID* and *s* are changed as *newOP*, *i-1*, *gID-sum* (the position where the old *gID* is allocated in the set *setNo*) and *s-setNo*. Otherwise, the algorithm stops recursion as it derives.

In the aforementioned example, to convert group id 2 in Figure 3.6 (c) into the *p_comb* $< 1, 3 >$ in Figure 3.6 (d), Algorithm 3.3 is first called with four input parameters (0, 2, 2 and 3). Since *gID* 2 falls to the set 1, the first object position 1 is derived by 0+1. And then, Algorithm 3.6 enters the second recursion with new parameters (1, 1, 2 and 2). As shown in the Figure 3.6 (e) and (f), the *gID* 2 falls to set 2, thus another new position, 3 is derived from 1+2. In the second recursion, step id i equals to 1 and Algorithm 3.3 stops recursion. Thus, in two recursion, Algorithm 3.3 is able to convert group id 2 into the *p_comb* $< 1, 3 >$ and further obtain its *seed_comb* $< 1, 4 >$.

3.6 Processing of Allocated Combinations

In section 3.5, we have introduced how tasks are distributed to each reducer. In this section, we shall look at how each reducer processes the tasks that have been allocated to it. Clearly, a naive strategy is to process the allocated tasks one at a time. For epistasis discovery applications, each task is the statistical test of a combination of objects computed based on the method described in section 3.4. However, we can optimize the performance by salvaging partial work done by one task for another task when they share some common computation. We refer to this as the *sharing optimization*.

For the purpose of our discussion, we shall illustrate the sharing optimization under the *Exhaustive Testing* scenario. As mentioned in section 3.5.1, the `task distributor` assigns the tasks to each `reducer` based on sets. Each `reducer` receives a set of set ids which it should conduct.

Sharing Optimization: Our *COSAC* framework minimizes the cost of processing a set of tasks and the contingency tables collection by salvaging common operations during the processing of combinations.

Let us assume that there are 5 objects and we need to perform 3-level analysis. Take the first set {123, 124, 125, 134, 135, 145} as an example, we illustrate how the sharing optimization is performed while enumerating and analyzing this set of combinations.

Given the first set, it is easy to see that {12} exists in all the three combinations {123, 124, 125} and {13} is shared between {134} and {135}. The methodology of our sharing optimization is that we only combine the parts that are common among multiple combinations once. And then the common part is used to combine with the other objects further. For instance, we calculate the results of the combinations {12} once, and use {12} to combine with 3, 4 and 5 further. In this case, we omit computing the sharing parts multiple times (as would have been done under the naive strategy) to reduce the computation overhead. Meanwhile, the the data (contingency tables) also do not need to

Algorithm 3.4: **Enumeration**

```
Input: Fixed combination Object: comObj,
Start object position: sOP,
Combination forward depth: cFD,
analysis level: m
1 end=List.size()- m + cFD-1 ;
2 for i: sOP → end do
3     newComObj=Combine(comObj,List.get(i));
4     if cFD == m then
5         MakeStatisticTest(newComObj);
6     else
7         Enumeration(newComObj,i+1,cFD+1,m);
```

be collected multiple times.

Algorithm 3.4 outlines the main steps of performing this sharing optimization which is able to support any deep level analysis flexibly. Algorithm 3.4 has four input parameters. First, it is the *comObj* which is used to store either the start object or the common combined objects. Second, it is the next object position (*sOP*) which *comObj* needs to combine with. Third, the *cFD* indicates the forward analysis depth. In other words, it records the number of objects in the combination after adding another objects to the *comObj*. Lastly, *m* is the analysis level which is set by users.

We take enumerating the first set aforementioned as a running example to illustrate how the algorithm works. Initially, the *comObj* is the object 1 and the next position of the object in the list to combine is 2. As combining another object with object 2, there will be 2 objects in the combination. Thus, the *cFD* is 2. And the analysis level *m* is a constant value once it is set by the users (3 in this example). The algorithm first calculates all the possible positions for the objects which need to combine with *comObj* (Lines 1, 2). In the above example, 2, 3 and 4 are the object positions which 1 needs to combine with. For all these possible positions, the algorithm then gets the corresponding objects from the list to combine with *comObj* one by one while the result is stored into a new object (*newComObj*) (Lines 2, 3). Meanwhile, the algorithm checks whether the depth is the

same as the analysis level *m*. If it is, it has enumerated one complete combination as required. And then it conducts a statistical testing on this combination and continues to combine with other possible objects (Lines 4, 5). Otherwise, the algorithm enters the next level to combine more objects with the new inputs (*newComObj*, *i+1*, *cFD+1*, *m*) (Lines 6, 7). For instance, in the example, since 2 is not equal to the analysis level 3, when 1 and 2 are combined, the algorithm enters into the next level to add another new object.

In the next level, the algorithm runs the same steps until it has combined with all the objects it needs. In the next level, $\{12\}$ is further combined with 3, 4 and 5. And then the algorithm returns to the upper level to combine 1 and 3 and so forth, until all the enumeration has been done.

In this way, the proposed algorithm does not only achieves the sharing optimization for efficient enumeration but also is flexible to support any analysis level. In the section 3.7, our experimental results have shown that the sharing optimization has significantly minimized the cost.

Semi-Exhaustive Testing: Under the *Semi-Exhaustive Testing* scheme, the task each reducer facing is to combine *seed_comb* with other non-seed objects as described in section 3.5.2. Algorithm 3.4 can also be used here. For a given analysis level *m*, if the number of objects in the *seed_comb* is *i*, the enumeration can be performed through calling the algorithm with input parameters: *seed_comb*, 0 (combine with the first object in the non-seed list), *i+1* (combination forward deep) and *m* (analysis level).

3.7 Experiment

We develop our *COSAC* framework based on the Apache Hadoop [2] which is an open source equivalent implementation of the MR framework, running on HDFS (Hadoop

Distributed File System). We conduct a series of experiments on our local cluster with over 40 nodes. For our local cluster, each node consists of a aX3430 4(4) @ 2.4GHZ CPU running Centos 5.4 with 8GB memory and 2x 500G SATA disks. Moreover, since our tasks at hand are computationally intensive, we set the number of `reducers` per node to be equal to the number of cores at the node, which is 4 in our local cluster . This guarantees that each `reducer` can get one core. Therefore, there are a total of 4*N `reducers` which can be run simultaneously on a N-node local cluster. In each analysis job, we set the number of `reducers` to 4*N to make full use of the resources in the cluster.

To evaluate *COSAC* on a large dataset, we generate the synthetic example dataset, which is in the same format as shown in Figure 3.1. Each dataset has 2000 samples which is considered large in GWAS. Each sample is generated with n+2 columns including one sample id, n objects and one disease status. As used in GWAS, each object has three possible candidate values (AA, AT or TT) and the disease status consists of one of two possible candidates (Breast Cancer or Heart Attacks). We note that the computation overhead is related to the data size, and is independent of the types of data. This is because we adopt the IRBI techniques to transfer the data and index the data. Thus, the results on the synthetic datasets we used would correspond to that for real datasets of the same sizes.

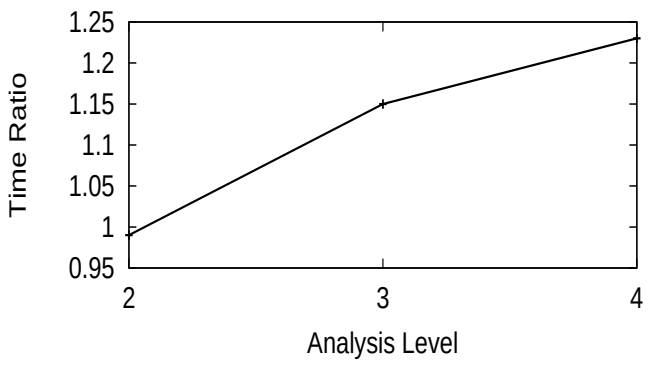


Figure 3.7: Execution time ratio for Round Robin/Bestfit

3.7.1 Performance Comparison among different Models

Bestfit and Round Robin: In this experiment, we study the performance comparison between **Bestfit** and **Round Robin** models. As aforementioned in section 3.5.1, **Round Robin** can be implemented in two different manners, RR_I and RR_II. Comparing these two implementations, from our knowledge RR_II is better than RR_I, as it provides more opportunities of sharing optimization than the first one. It is easy to see that the first one divides the tasks in each set (like each line in the Figure 3.4) into all the reducers, which reduces the number of tasks where sharing optimization can be exploited (as shown in section 3.6). Therefore, in our work, we only evaluate the more efficient round robin implementation, the second one, for performance comparison with the **Bestfit** model. Based on our intuition, **Round Robin** is inferior, because of different overheads (like maintaining much more meta information) and skewed sharing computation percentage in each reducer. To verify our address, we implement two programs, each of which simulate the slowest reducer under these two models, to conduct the performance comparison. Recall that in a MR job, the slowest reducer is the bottleneck of the whole job. In other words, comparing the slowest reducers is equal to comparing the whole job processing time.

For simplicity, each of our developed programs executes the same tasks for the slowest reducer in these two models on one single machine. In the **Bestfit** model, the slowest reducer is the one which gets the most computation tasks. The sharing percentage in each `reducer` is almost the same. In the **Round Robin** model, the slowest `reducer` is the one with the lowest sharing percentage, like the `reducer` which gets more small sets (like the last $\frac{C(n,m)}{r}$ tasks).

Figure 3.7 shows the execution time ratio of the slowest reducers between **Round Robin** and **Bestfit** for processing 500 objects in 10 reducers with different analysis levels. From the experimental results, when there is no sharing computation in 2-level

analysis, we can see that the **Bestfit** model is almost the same as the **Round Robin** model. This further confirms that **Bestfit** is able to balance the load well, since these are no sharing opportunity of these two models in 2-level analysis. The same execution time represents that these two models obtain almost the same number of combinations to processing. When the analysis level increases, **Bestfit** is always better than the **Round Robin** model. The reason is, with the sharing optimization for high level analysis, **Bestfit** model can balance the gain of sharing optimization among the reducers. However, in **Round Robin** model, this gain is skewed among different reducers. We note that this experimental evaluation is already biased against the **Bestfit** model in real systems. Because the **Round Robin** needs to maintain much more meta information to tell each `reducer` the assigned tasks which brings a lot of other overhead. But in our evaluation, we omit these overheads for **Round Robin** model. Therefore, the small benefit of load balancing for **Round Robin** is less than what we can benefit from the **Bestfit** model.

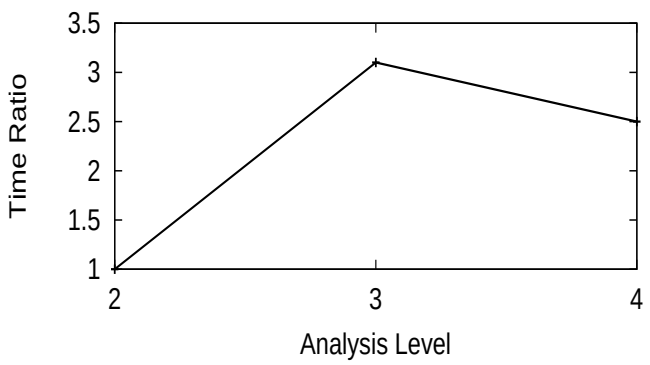


Figure 3.8: Execution time ratio for Greedy/Bestfit

Bestfit and Greedy: In this experiment, we study the performance comparison for the *Exhaustive Testing* between **Bestfit** and **Greedy** which is proposed in [81] when we vary the analysis level. This experiment is conducted with 500 objects on a 41-node cluster including 1 master node and 40 slave nodes. Figure 3.8 shows the execution time for both of the two models when we exhaustively analyze the correlation effects

for different analysis levels. We observe that the **Bestfit** model always outperforms the Greedy model especially when the analysis level is large. There are two reasons for this. First, the **Bestfit** model is able to provide better load balancing across the system nodes. Second, **Bestfit** benefits more from the sharing optimization. Recall that opportunities for sharing is bigger for larger sets. While **Bestfit** spreads the load of large sets across different nodes, **Greedy** allocates large sets to only a few nodes. As such, nodes with large sets complete much faster than those with smaller sets (even if the load is almost the same across all reducers). As discussed above, we observe that the last `reducer` is always the bottleneck of the whole processing job for large analysis level under the **Greedy** model.

Based on all these results, for the subsequent experiments, we only evaluate the **Bestfit** model in our *COSAC* framework.

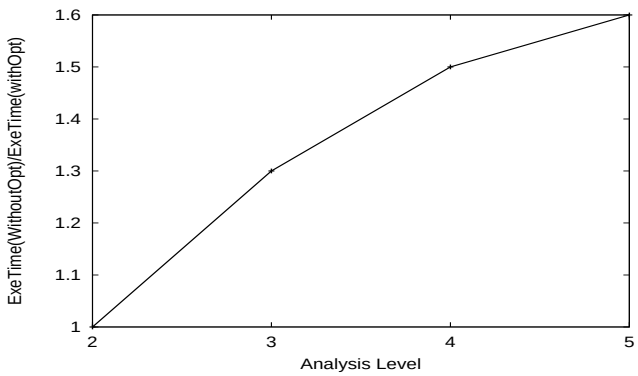
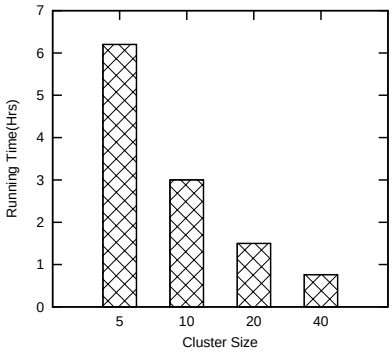


Figure 3.9: Execution time ratio for CSA without/with sharing optimization

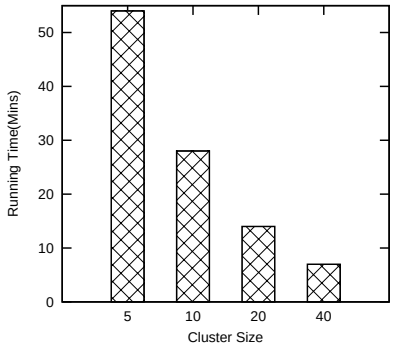
3.7.2 Sharing Optimization

In this experiment, we study how much the sharing optimization can benefit compared with the one without sharing optimization during combination enumeration and contingency table collection. For providing a fair and pure comparison, we implement two programs and evaluate the performance using a single CPU-core on a single ma-

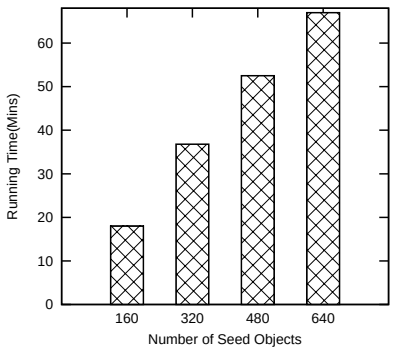
chine. One of the programs enumerates all the combinations and collect the contingency table without sharing any common parts between different combinations. Another program is implemented according to our proposed sharing optimization strategy as described in section 3.6. Figure 3.9 shows the results for the execution time ratio between the scheme without sharing optimization and with optimization under different analysis level. The data set used in this experiment contains 500 objects. From the result, we can see that as the analysis level increases, the sharing optimization benefits more. This is reasonable as higher analysis level means more sharing opportunity during the combination numeration and contingency table collection.



(a) Scalability for Exhaustive Testing



(b) Scalability for Semi-Exhaustive Testing



(c) Scalability for Semi-Exhaustive Testing with different seeds

Figure 3.10: *COSAC* Scalability Evaluation

3.7.3 Scalability

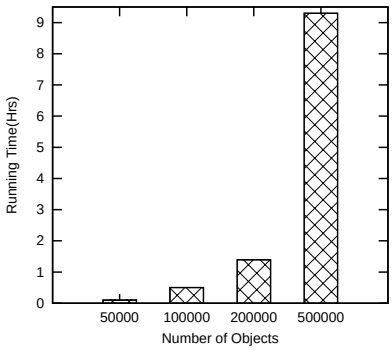
In this set of experiments, we study the scalability of the *COSAC* framework for both *Exhaustive Testing* and *Semi-Exhaustive Testing* schemes as the system resources increase. These experiments are conducted with 3000 objects and the analysis level is 3. Figure 3.10 (a) shows the execution time for *Exhaustive Testing* on the clusters with 5, 10, 20 and 40 nodes. The result shows a linear speedup in performance that the execution time reduces by half as we double the resources.

Figure 3.10 (b) reports the execution time for *Semi-Exhaustive Testing* with 160 seed objects on the clusters with 5, 10, 20 and 40 nodes. Similarly, we observe not only almost a linear speedup but also a big reduction in the execution time compared to *Exhaustive Testing*. Both the results confirm that our framework has a good scalability with increasing system resources and can achieve good load balancing.

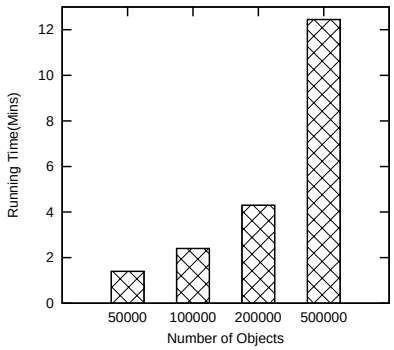
In this experiment, we report the scalability of *Semi-Exhaustive Testing* while we vary the number of seed objects. Figure 3.10 (c) reports the experiment conducted with 5000 objects with 160, 320, 480 and 640 seed objects for 3-level analysis on the same 41-node cluster. From the result, we can see that the execution time increases while increasing the number of seed objects. In fact, we observe that the execution time almost doubles when we double the number of seed objects. This further confirms that our framework can balance the load well.

3.7.4 Performance

This set of experiments study the performance of the *COSAC* framework when we vary the number of objects for 2-level analysis on a 41-node cluster. Figure 3.11 (a) shows the execution time for exhaustively computing all the significant interactions for pairwise analysis with 50 000, 100 000, 200 000 and 500 000 objects without outputting any result. The result shows that our *COSAC* framework offers a feasible and practical



(a) Performance for Exhaustive Testing



(b) Performance for Semi-Exhaustive Testing with 160 seed objects

Figure 3.11: *COSAC* Performance Evaluation

solution to perform pairwise epistasis for a large number of objects. The PLINK [60], which is considered as the most computationally feasible method in the review [20], process the 2-level analysis for 89,294 objects in 14 days. According to [47], it would require 1.2 years to do the 2-level analysis of 500 000 objects using the serial program on a 2.66GHz single processor without parallel processing. Our previous work [80] estimates that processing 500 000 objects using Naive approach instead of IRBI technique needs roughly 25 days for 2-level analysis on a 43-node cluster. But our *COSAC* framework can finish this task around 9 hours on a 41-node cluster via our optimization techniques. We expect our framework to be able to process 1 million objects around 8 hours on a 200-node cluster. This further confirms the efficiency of our framework. Furthermore, MR/Hadoop framework has been shown to scale to thousands of machines well with good fault tolerancess. Therefore, we believe that *COSAC* will be able to provide powerful computation capacity to discover more interesting results from large-scale datasets.

Figure 3.11 (b) depicts the processing time for *Semi-Exhaustive Testing* for 2-level analysis with 160 seed objects with 50 000, 100 000, 200 000 and 500 000 objects without outputting any result also. The results show that the *Semi-Exhaustive Testing* can reduce the computation time a lot. It can efficiently process the 500 000 objects in

about 12 minutes. This experiment highlights the efficiency gains by pruning technique supported by our framework and confirms the practice and utility of our framework for large scale data analysis.

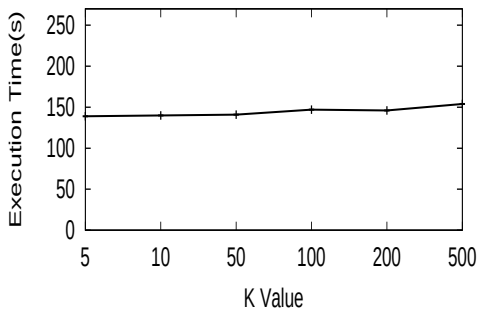


Figure 3.12: Execution time for different k values

3.7.5 Top-k Retrieval

Another important component that we can further provide is to allow users to retrieve only the top-k most significant results with the lowest p-values for observation. In our framework, we store the analysis result in HDFS to allow users to do further analysis. We have also developed such a capability under the MR framework. The basic idea is to split the output of the analysis result into chunks. Each chunk is then assigned to one `mapper`. Next, each `mapper` will select the top-k most significant results and shuffled these results to one `reducer`. Finally, the `reducer` can determine the global top-k answers based on all local top-k ones it receives. Our top-k scheme is very efficient gaining from avoiding shuffling a large volume of data from map phase to reduce phase.

This set of experiments is conducted on 15 machines each of which consists of 2 Intel Nehalem quad-core processors (8 cores) and 48GB memory in TACC (Texas Advanced Computing Center). Figure 3.12 shows the execution time while retrieving top-k most significant tuples from 80GB result dataset for 3-level analysis when we vary k from 5 to 500. As expected, the execution time does not change much on different k values. This

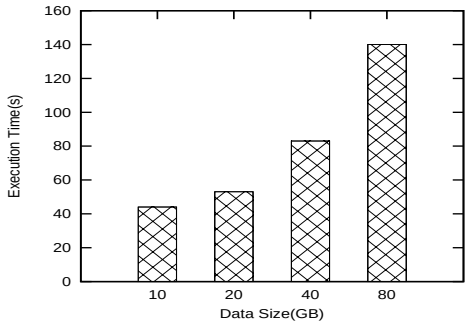


Figure 3.13: Execution time for top-50 retrieval from different datasets

is because the main overhead of top-k retrieval is the scanning overhead to find the local top-k tuples in the mappers. For different k values, this scanning overhead is almost the same once the base data is the same.

Figure 3.13 depicts the execution time for retrieving the top 50 tuples on different size of 3-level analysis result datasets from 10GB to 80GB. It is not surprising that the execution time increases when the dataset becomes larger. Another observation is that when the dataset is small, even though the dataset is reduced to half, the execution time may not reduce too much. This is reasonable, since the setup and runtime overheads of the MR framework reduces the benefits of reducing dataset. As shown in the result, when the dataset size increases, the affect of these overheads becomes smaller.

3.8 Summary

This work aims to provide practical and efficient techniques for combinatorial statistical analysis systems using existing cloud computing techniques. We proposed a practical, efficient, scalable and flexible framework based on the MapReduce paradigm which is well supported in cloud by cloud providers. The elasticity and pay-as-you-use features have made large-scale data analysis in scientific data processing available for all end users. Our work has demonstrated how MapReduce can be used in computation-

intensive systems. In this work, we have provided a flexible parallel object enumeration scheme with good load balancing for large-scale data in the MapReduce framework. Furthermore, we proposed the technique called Integer Representation and Bitmap Indexing for efficient statistics analysis and the sharing optimization by salvaging computations in each processing unit. We have also discussed how these techniques can be generally used in many other different applications besides combinatorial statistics analysis systems.

CHAPTER 4

DATA CUBE ANALYSIS

4.1 Overview

In Chapter 3, we have studied how to build a scalable framework for a computation intensive analysis, the combinatorial statistical analysis, based on the MapReduce paradigm. There are different challenges and difficulties to develop scalable parallel data processing techniques for data intensive applications such as the data cubes analysis.

For instance, an efficient solution of data intensive applications has to guarantee that the data shuffling and data read/write overheads among the cluster are minimized. Otherwise, the high overheads incurred during these analyses may significantly affect the performance. Furthermore, it is more challenging (compared to computation intensive analysis) to develop an effective load-balancing strategy - besides having to consider the computation overhead in each reducer. In addition, there is also a need to factor in the data I/O and shuffling overhead. Data cube analysis is such a very typical and popular representative data intensive analysis. In this Chapter, we tackle the problem of

developing an effective, scalable and practical parallel system for data cube analysis.

In many industries, such as sales, manufacturing, transportation and finance, there is a need to make decisions based on aggregation of data over multiple dimensions. `Data cubes` [31] are one such critical technology that has been used in data warehousing and On-Line Analytical Processing (OLAP) for data analysis in support of decision making.

Much research has been devoted to the data cubes analysis in the literature [7][95][96][44]. However, existing techniques can no longer meet the demands of today's workloads. On the one hand, the amount of data is increasing at a rate that existing techniques (developed for a single machine or a small number of machines) are unable to offer acceptable performance. On the other hand, more complex aggregate functions (like complex statistical operations) are required to support complex data mining and statistical analysis tasks. Thus, new mechanisms are needed to efficiently support data cube analysis on more complex aggregate functions over a large amount of data.

Therefore, in this thesis, we are motivated to develop a scalable parallel data cube analysis platform for large-scale data. While we provide a new cubing algorithm, our key contributions lie in the design of techniques to extend the MR framework for efficient data cube analysis to broaden the application of data cubes primarily for append-only environments [79].

Our main contributions are as follows. First, we present a distributed system, `HaCube`, an extension of MR, for data cube analysis on large-scale data. `HaCube` modifies the Hadoop MR framework while retaining good features like ease of programming, scalability and fault tolerance. It also builds a layer with user-friendly interfaces for data cube analysis. We note that `HaCube` retains the conventional Hadoop APIs and thus is compatible with MR jobs. Second, we show how batching cuboids for processing can minimize the read/shuffle overhead to salvage partial work done for efficient data cube materialization. Third, we propose a general and effective load balancing scheme

$\mathcal{LBCCC}$ (short for Load Balancing via Computation Complexity Comparison) to ensure that resources are well allocated to each batch. $\mathcal{LBCCC}$ can be used under both HaCube and MR frameworks. Fourth, we adopt a new computation paradigm, MMRR (`MAP-MERGE-REDUCE-REFRESH`), with a local store under `HaCube`. `HaCube` supports efficient view updates for different measures, both distributive such as SUM, COUNT and non-distributive such as MEDIAN, CORRELATION, and thus is able to support more applications with data cube analysis in a data center environment. To the best of our knowledge, this is the first work to address data cube view maintenance in MR-like systems. Finally, We evaluate `HaCube` based on the TPC-D benchmark with more than 3 billions tuples. The experimental results show that `HaCube` has significant performance improvement over Hadoop.

The rest of this chapter is organized as follows. Section 4.2 reviews some background material. In Section 4.3, we provide an overview of the `HaCube` architecture and computation paradigm. Sections 4.4 and 4.5 present our proposed data cube materialization and view maintenance approaches. In Section 4.6, we discuss some issues including the fault tolerance strategy and storage cost. We report our experimental results in Section 4.7 and summarize this work in Section 4.8.

4.2 Preliminaries

In this section, we introduce the notations and background of data cube materialization and view maintenance.

4.2.1 Data Cube Materialization

In OLAP, the attributes are classified into **dimension attributes** (the grouping attributes) and **measure attributes** (the attributes which are aggregated) [31]. Each GROUP

BY in a CUBE computation is defined as a cuboid which captures the aggregate data. To speed up query processing, these cuboids are typically stored into a database as views. The problem of **data cube materialization** is to efficiently compute all the views. If the cube is being built for the first time, we refer to its materialization as *initial* cube materialization. Figure 4.1 shows all the cuboids represented as a cube lattice with 4 dimensions A, B, C and D.

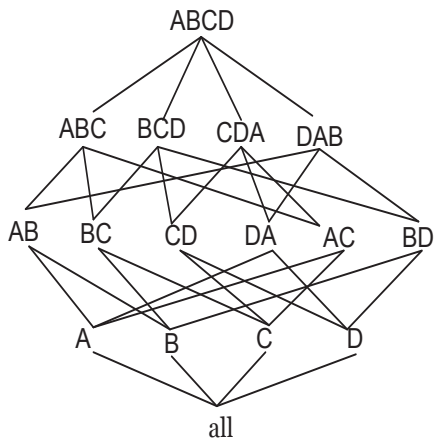


Figure 4.1: A cube lattice with 4 dimensions A, B, C and D

4.2.2 Data Cube View Maintenance

The goal of **cube view maintenance** is to get the latest view when new data are produced and added. We refer to this newly produced data as delta data ΔD, the data used for the previous view materialization as base data D and the previously materialized cube as base view V. In terms of view update requirements, the measure functions can be classified into two categories: non-distributive and distributive measures [31].

Non-distributive measures are those whose updated views can only be reconstructed by **recomputation** based on the entire base data D and ΔD. In append-only applications, these functions include STDDEV, MEDIAN, CORRELATION, and REGRESSION functions.

Distributive measures are the ones whose views can be either updated by recomputation (as in non-distributive functions) or incrementally computed which is referred to as **incremental computation**. Incremental computation updates a view based on V and ΔD in two steps [52]: (a) In the `propagate` step, a delta view ΔV is calculated based on the ΔD. (b) In the `refresh` step, the updated view is obtained by merging V and ΔV. In append-only applications, functions that can be computed incrementally include SUM, COUNT, MIN, MAX and AVG. Note that we have classified algebraic functions, like AVG, as distributive measures. To avoid recomputation, views of algebraic functions can be updated by keeping some extra information. For instance, for computing AVG, we can record both the sum and count in the views.

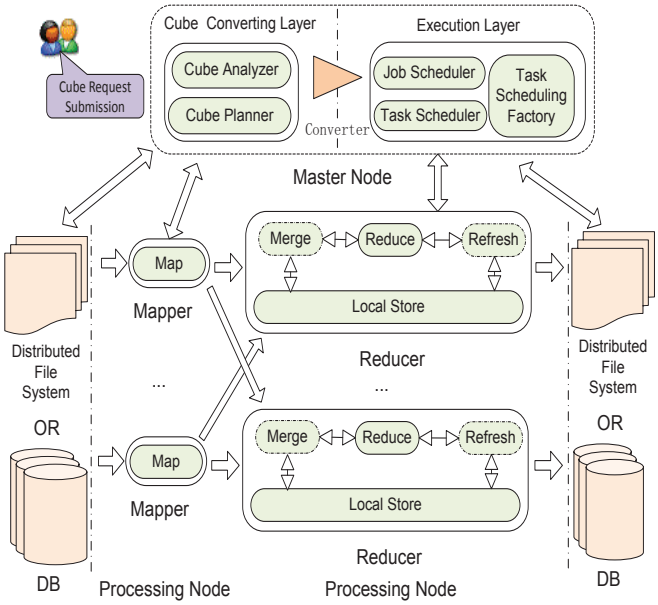


Figure 4.2: HaCube Architecture

4.3 HaCube: The Big Picture

To support efficient data cube analysis, we develop a new system, `HaCube`, which is an extension of MR based on Hadoop [2]. `HaCube` integrates another layer for fast data

cube analysis development and adopts a new computation paradigm to support data cube operations. We now present the `HaCube` architecture and its computation paradigm.

4.3.1 Architecture

Figure 4.2 gives an overview of the basic architecture of `HaCube`. Similar to MR, all the nodes in the cluster are divided into two different types of function nodes, including the master and processing nodes. The master node is the controller of the whole system and the processing nodes are used for storage as well as computation.

Master Node: The master node consists of two functional layers:

1. The `cube converting layer` is used to receive the user cube analysis request (either initial cube materialization request or view update request) and to convert the cube analysis request into an execution job which can be either an initial cube materialization job or a view update job. The cube converting layer contains two main components: `Cube Analyzer` and `Cube Planner`.

The cube analyzer is designed to analyze the cube, such as figuring out the cube id (the identifier of the cube analysis application), analysis model (a new initial cube materialization request or a view update request), measure operators (data cube aggregation function), and input and output paths.

The cube planner is developed to transfer the cube analysis request into either a cube materialization or view update job. The execution job is divided into multiple tasks each of which handles part of the cuboid calculation. The cube planner consists of several functional components such as the execution plan generator (combine the cuboids into batches to reduce the overhead), and load balancer (assign the right number of computation resources for each batch).

2. The `execution layer` is responsible for managing the execution of jobs passed from the cube converting layer. It has three main components: `job scheduler`,

`task scheduler` and `task scheduling factory`.

We use the same job scheduler as in Hadoop which is used to schedule different jobs from different users. In addition, we add a task scheduling factory which is used to record the task scheduling information of a job which can be reused in other jobs. Furthermore, we develop a new task scheduler to schedule the tasks in terms of the scheduling history stored in the task scheduling factory.

Processing Node: A processing node is responsible for the task execution assigned from the master node. Similar to the MR framework, each processing node contains one or more processing units each of which can either be a `mapper` or a `reducer`. Each processing node has a `TaskTracker` which is in charge of communicating with the master node through heartbeats, reporting its status, receiving the task, reporting the task execution progress and so on. Unlike the MR framework, there is a `Local Store` built at each processing node running `reducers`. The local store is developed to cache the useful data in the local file system on the reducer node from a job. It is a persistent storage in the local file system and will not be deleted after a job execution. In this way, tasks (possibly from other jobs) assigned to the same reducer node are able to access the local storage directly from the local file system.

4.3.2 Computation Paradigm

`HaCube` inherits some features from the MR framework, such as data read/process/write format of (key, value) pairs, sorting all the intermediate data and so on. However, it further enhances MR to support a new computational paradigm. `HaCube` adds two optional phases - a `Merge` phase and a `Refresh` phase before and after the reduce phase - to support the MAP-MERGE-REDUCE-REFRESH (MMRR) paradigm as shown in Figure 4.2.

The `Merge` phase has two functionalities. First, it is used to cache the data from the

reduce input to the local store. Second, it is developed to sort and merge the partitions from mappers with the cached data in the local store. The `Refresh` phase is developed to perform further computations based on the reduce output data. Its functionalities include caching the reduce output data to the local store and refreshing the reduce output data with the cached data in the local store. These two additional phases are intended to fit different application requirements for efficient execution support.

As mentioned, these two phases are optional for the jobs. Users can choose to use the original MR computation or MMRR computation. More details can be found in Section 4.5 about how MMRR benefits the data cube view maintenance.

4.4 Initial Cube Materialization

In this section, we describe our proposed parallel algorithm, `CubeGen`, for efficient cube materialization which is suitable for both the `HaCube` and the MR frameworks. There are two alternative implementation of cube materialization: full materialization (compute all the cuboids) and partial materialization(compute selective cuboids). Since full materialization is the fundamental problem where the techniques on it may have a strong influence to all other techniques [85], we focus on fully materializing the graph cube in this thesis.

We first present some principles of sharing computation through cuboid batching, followed by a batch generator. Then we provide our proposed load balancer which guarantees that the resources to each batch are well allocated. Finally, we introduce the implementation details of `CubeGen`. Recall that for n dimension attributes, there are 2^n cuboids that need to be computed. To simplify our presentation, we omit the cuboid "all". This special cuboid can be easily handled through an independent processing unit.

4.4.1 Cuboid Computation Sharing

Given 2^n cuboids, the naive solution of processing each cuboid in one MR job incurs significant overheads, such as data read overhead (completing one cube needs multiple base data traversal), sort overhead (sorting each cuboid data independently is costly) and shuffle overhead (shuffling each cuboid data from the `Mapper` to the `Reducer` independently is expensive). Thus, finding the right way to batch and combine the computation becomes important for efficient materialization.

We provide the following lemma as a formal basis for combining and batching the cuboids computation under MR-like frameworks.

Lemma 4.1. *Let A and B be a set of dimension attributes such that $A \bigcap B = \emptyset$. In MR-like systems, given cuboid A and AB, A can be combined and processed together with AB, once the MR job sets AB as the key, sorts the key based on AB and partitions the key based on A. A is referred to as a* `Sub-Cuboid` *of AB (denoted as $A \subseteq AB$). Note that the notion of sub-cuboid requires the dimensions of the sub-cuboid to be a prefix of the dimensions of the cuboid.*

Proof. Without loss of generality, we assume $A = d_1$, ..., d_x and $B = d_y$, ..., d_{y+z} where d_i is a dimension attribute. For processing the cuboid AB, when the mapper emits the (key, value) pairs where the key is set as d_1, ..., d_x, d_y, ..., d_{y+z} and is sorted, $d_1, ..., d_x$ is in sorted order also. Partitioning AB based on A guarantees that all the output (key, value) pairs with the same $d_1, ..., d_x$ are shuffled to the same reducer. As all the same values of A are shuffled to the same reducer and A is sorted, we can calculate A at the same time as we process AB. Thus, the data read/sort/shuffle overheads of processing the cuboid A can be removed though sharing the computation. □

The above results can be generalized using transitivity: Since we can combine the processing of the pair of cuboids $\{A, AB\}$ and the pair $\{AB, ABC\}$, we can also com-

bine the processing of the three cuboids $\{A, AB, ABC\}$. Thus, given one cuboid, all its sub-cuboids can be calculated together as a batch. For instance, for the cube lattice in Figure 4.1, as $A \subseteq ABCD$, $AB \subseteq ABCD$ and $ABC \subseteq ABCD$, the cuboids A, AB, ABC can be processed with $ABCD$. We note that BC cannot be processed with $ABCD$ because $BC \nsubseteq ABCD$. Given a batch, the principle to calculate this batch is to set the sort attributes as the key and partition the (key, value) pairs based on the partition attributes in the key in the MR job. We formally define these two attribute classes below.

Definition 4.1. ***Partition Attributes:*** *The dimension attributes in cuboid A are called the partition attributes if A is the sub-cuboid of all other cuboids in one batch.*

Definition 4.2. ***Sort Attributes:*** *The dimension attri- butes in cuboid A are called the sort attributes if all the other cuboids are sub-cuboids of A.*

For instance, given the batch $\{A, AB, ABC, ABCD\}$, $ABCD$ and A can be set as the sort attributes and partition attributes respectively.

Based on the aforementioned principles, `CubeGen` combines and batches all the cuboids that share the same prefix (e.g. AB and ABC) to share the data read/ sort/shuffle computation. In this way, we can remove the unnecessary overheads as much as possible. One advantage of choosing such a prefix-based and sort-based computation sharing is that the `HaCube` and the MR frameworks sort all the intermediate data. Thus, a prefix-based and sort-based algorithm can exploit the sorting for free.

To achieve good performance, we need to address two issues. First, how can we find the minimum number of batches from the 2^n cuboids? As more cuboids are combined together, the shuffling overhead incurred for data shuffling will be reduced. Second, how can we balance the load to assign the right number of computation resources to each batch? As different batches may have different computation complexity and data size, it is not optimal to evenly assign the computation resources to each batch. Before providing

the detailed algorithm for `CubeGen`, we first introduce how it solves the aforementioned challenges by developing a `plan generator` and a `load balancer`.

4.4.2 Plan Generator

The goal of the plan generator is to generate the minimum number of batches among the 2^n-1 cuboids, excluding "all". The plan generator first divides the 2^n-1 cuboids into n groups each of which consists of the cuboids with i dimension attributes. For instance, given the cube lattice with 4 dimension attributes in Figure 4.1, it can be divided into 4 groups (from the bottom of the lattice to the top) as follows: $G_1 = \{A, B, C, D\}$, $G_2 = \{AB, BC, CD, DA, AC, BD\}$, $G_3 = \{ABC, BCD, CDA, DAB\}$, $G_4 = \{ABCD\}$.

Recall that one cuboid can be batched with all its sub-cuboids. Thus, we adopt a *greedy approach* to combine one cuboid with as many of its sub-cuboids as possible. Initially, all the cuboids in each group are marked as available. Each construction of a batch starts with one available cuboid, α, from the non-empty group with the maximum number of dimensions. It then searches all the available sub-cuboids of α from other groups that can be batched together. For instance, the first batch construction starts with $ABCD$ in the example above (Since $ABCD$ has 4 dimensions, it is the one with maximum number of dimensions). Note that since cuboid α has different permutations (e.g. $ABCD$ can also be permuted as $ABDC$, $ACBD$, $BCDA$, $CDAB$, $DABC$ etc.), the algorithm enumerates all permutations and the one with the maximum number of available sub-cuboids will be chosen. Once one batch is constructed, all the cuboids in this batch are deleted from the search space and become unavailable. Similarly, the next batch construction is conducted among the remaining available cuboids. The construction finishes when there are no available cuboids left.

The approach we adopt to generate the batches is similar to the one proposed in [44]. Lee et al. provide an extensive proof that the algorithm is able to generate $C_n^{\lceil \frac{n}{2} \rceil}$ batches

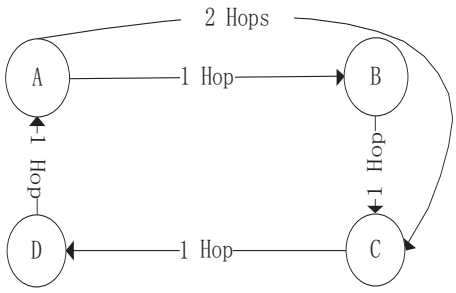


Figure 4.3: A directed graph of expressing 4 dimensions A, B, C and D

which is the minimum number. Recall that there are $C_n^{\lceil \frac{n}{2} \rceil}$ cuboids in group $G_{\lceil \frac{n}{2} \rceil}$ and that none of them can be combined with each other. So there are at least $C_n^{\lceil \frac{n}{2} \rceil}$ batches. Interested readers are referred to [44] for more details.

To improve the efficiency of batch construction, two optimizations are adopted to reduce the search space.

- First, recall that for a set of dimensions, we need to compute a batch for each permutation. This, however, may not be necessary. In fact, when all the sub-cuboids of a particular permutation are available, we know that we have found a permutation with the maximum number of sub-cuboids. Therefore, as soon as we encounter such a permutation, we do not need to continue the search for this set of dimensions.

- Second, we organize all the dimensions as one directed graph such that one dimension points to another and we refer to the distance between two adjacent dimensions as one hop. For instance, given 4 dimensions A, B, C and D, they can be expressed as a directed graph such that A, B, C and D point to B, C, D and A respectively as shown in Figure 4.3. During permutation enumeration, changing from A to B or C is referred as moving one hop or two hops from A.

  To find the permutation of a cuboid α with the maximum number of sub_cuboids, the enumeration starts from the permutation that is obtained by moving the equiv-

alent number of hops for each dimension of the unavailable cuboid in the same group. This is to guarantee that, most likely, the first search permutation is the one we need to reduce the search space. For instance, assume that the first batch is generated as $ABCD$, ABC, AB and A. Then the next initial permutation for a new batch is BCD which is computed through moving one hop for each dimension in the unavailable cuboid ABC. It is clear that the new batch can be generated with BCD (since all its sub_cuboids are available) and there is no need to search other permutation of BCD.

In the same way, the batches of CDA and DAB will be generated. These two optimizations speed up the batch construction.

Figure 4.4 shows an example of the generated batches using the dotted lines. Different to the three classic cube computation approach, MultiWay, BUC or Star-Cubing mentioned in Section 4.2.2, `CubeGen` generates the batches that are highly restricted to the prefix rule towards a parallel computation. This new batching method is able to facilitate the sorting feature of the MR-like systems such that there is no need to perform any extra sorting during the cube computation towards an efficient cube materialization. The existing cubing approaches that are designed for a centralized machine or a small cluster are neither efficient nor applicable for the MR-like systems. One of the main reasons is that they may generate batch with the non sub-cuboids that is against Lemma 1. For instance, in existing methods, both AB and BC may be processed together with ABC.

4.4.3 Load Balancer

Given a set of batches from the plan generator, the load balancer is used to assign the right number of computation resources to each batch to balance the load.

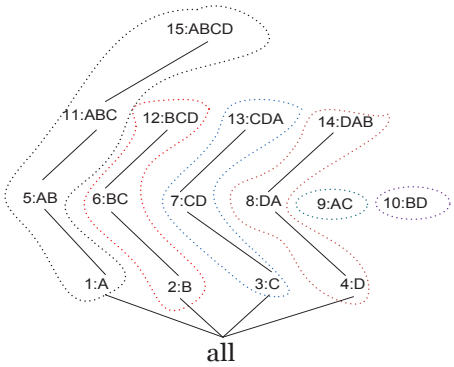


Figure 4.4: The numbered cube lattice with execution batches

We argue that existing works (like [57]) that balance the batches by evenly assigning the computation resources may not always be a good choice.

- First, it requires users to provide very specific information about the application data to be able to estimate a model to find the balanced batches.
- Second, the cuboids may not be combined into balanced batches.
- Third, in a MR-like system, it is hard to make a precise cost estimate of each batch. For instance, the total cost of each batch includes the following main parts: the data shuffling cost (shuffling the intermediate data from mappers to reducers), the sorting cost (all the intermediate data are sorted), data processing cost (applying the measure function to each cuboid in a batch) and data writing cost (writing the views to the file system). It is hard to estimate each of these component costs and even harder to evaluate the total cost of each batch (as this requires setting the appropriate weights when combining these components).

Therefore, the load balancing among different batches becomes a very tricky and challenging problem in MR-like systems.

In the thesis, we propose a novel load balancing scheme $\mathcal{LBCCC}$ (short for Load Balancing via Computation Complexity Comparison) to assign the right number of com-

putation resources to each batch. Intuitively, $\mathcal{LBCCC}$ adopts a profiling based approach where a learning job (we refer as CCC - Computation Complexity Comparison job) is first conducted on a small test dataset to evaluate the computation overhead relationship between each batch and then generate the number of reducers for each batch that are proportional to computation overhead for the actual `CubeGen` jobs. The computation cost relationship is estimated through the execution comparisons when each batch is provided the same number of computation resources. The execution time relationship over the same computation resource indicates the entire batch processing overhead relationship, thus helps to make an accurate load balancing decision.

In particular, $\mathcal{LBCCC}$ first conducts the CCC job, a cube materialization learning job, on a small test dataset where each batch is assigned to one reducer. It then records and utilizes the execution time of each batch to estimate the computation overhead relationship among different batches.

The test data can be obtained either by sampling or produced within a time window provided by users. Note that the sampling can be accomplished during the $\mathcal{CCC}$ job, since the MR framework provides APIs for sampling data directly. Therefore, by default, we use the sampling approach provided by the Hadoop API where one tuple is sampled from every s records. Users can also plug in their own sampling algorithm easily. The sampling algorithms have been widely studied in the literature. Since it is not our focus on studying how to choose or design a good sampling algorithm, we shall not discuss it here and more sampling algorithms can be found in [25].

In the $\mathcal{CCC}$ job, given a set of base data, the `mapper` conducts sampling on it. For each sampling tuple, the `mapper` emits multiple (key,value) pairs each of which is for one batch. Then the $\mathcal{CCC}$ job shuffles the pairs that belong to the same batch to one particular reducer. Given b batches, the $\mathcal{CCC}$ job uses b reducers each of which is in charge of processing one batch. The implementation of the $\mathcal{CCC}$ algorithm is similar to

the `CubeGen` algorithm provided in Algorithm 4.1 except for the number of reducers assigned to each batch. The algorithm detail can be found in section 4.4.4.

The $\mathcal{CCC}$ learning job records the execution time T_i for processing batch B_i. Based on the execution time recorded, the load balancer generates the resources assignment plan for each batch. Given r reducers, the right number of reducers, R_i for batch B_i can be calculated as follows:

$$R_i = \frac{T_i * r}{\sum_{j=0}^{b-1} T_j}$$

The load balancer integrates this plan into the `CubeGen` algorithm to balance the load. The experimental results show that $\mathcal{LBCCC}$ is able to balance the load very well.

We note that this evaluation only needs to be done once before the initial cube materialization and is used for subsequent jobs in the same application. Furthermore, performing the $\mathcal{CCC}$ job is cheap as only b reducers are needed. Note that the load balancer can support different kinds of batching approaches. Therefore, the $\mathcal{LBCCC}$ load balancing scheme is general and effective for different cubing algorithms.

4.4.4 Implementation of CubeGen

Assume that the batch plan B with b batches (B_0, B_1, ..., B_{b-1}), and load balancing plan R with b resource assignments (R_0, R_1, ..., R_{b-1}) have been generated by the plan generator and load balancer. The proposed `CubeGen` algorithm can conduct the cube materialization in one job and its pseudo-code is provided in Algorithm 4.1.

Map phase: The base data is split into different chunks each of which is processed by one mapper. `CubeGen` parses each tuple and emits multiple (key, value) pairs each of which is for one batch thus removing the multiple data reading overheads (Lines 5-10). The sort attributes in the batch are set as the key and the measure attribute is set as the value.

Algorithm 4.1: **CubeGen Algorithm**

```
1  Function: Map(t)
2  # t is the tuple value from the raw data
3  Let 𝔹 be the batch set includes B_0, B_1, ..., B_{b-1}
4  Let I_i be the identifier of batch B_i
5  for each B_i in 𝔹 do
6      k ⇐ get sort attributes in B_i from t
7      v ⇐ get measure attribute m from t
8      # If there are multiple measure attributes e.g. m_1, m_2, they can be put to v
       together such as v ⇐ (m_1, m_2)
9      v.append(I_i)
10     emit(k,v)
11 Function: Partitioning(k, v)
12 Let R_i be the number of reducers assigned for B_i
13 Let S_i be Σ_{j=0}^{i-1} R_j
14 Let attr be the partition attributes in B_i
15 return S_i + hash(attr, R_i)
16 Function: Reduce/Combine (k, {v_1, v_2, ..., v_m})
17 # 𝕄 is the measure function
18 Let ℂ be the cuboid set in the batch identifier
19 for C_i in ℂ do
20     if C_i is ready then
21         k ⇐ get dimension attributes in C_i
22         v ⇐ 𝕄(v_1, ..., v_m, v'_1, ..., v'_k, ...)
23         # If users need multiple measure functions e.g. (𝕄_1, 𝕄_2), they can all be
           applied in the same job, such as v_1 ⇐ 𝕄_1(v_1, ..., v_m, v'_1, ..., v'_k, ...) and v_2
           ⇐ 𝕄_2(v_1, ..., v_m, v'_1, ..., v'_k, ...)
24         emit(k,v)
25     else
26         Cache {v_1, v_2, ..., v_m}
```

To distinguish which (key, value) pair is for which batch with which cuboids, we add a batch identifier appended after the value.

The identifier is developed as one Bitmap with 2^n bits where n is the number of dimension attributes and each bit corresponds to one cuboid. First, we number all the 2^n cuboids from 0 to $2^n - 1$. Second, if the cuboid is included in one batch, its corresponding bit is set as 1, otherwise 0. For instance, Figure 4.4 depicts an example of a numbered cube lattice. Assume that B_0 consists of cuboids $\{A, AB, ABC$ and $ABCD\}$. The identifier for B_0 is set as '$10001000\ 00100010'$.

The partitioning function partitions the pairs to the appropriate partition based on the identifier and the load balancing plan R. `CubeGen` first schedules the data into the right range of reducers. Recall that the batch B_i is assigned R_i reducers. Therefore, the assigned reducers for batch B_i are from $\sum_{j=0}^{j-1} R_j$ to $\sum_{j=0}^{j-1} R_j + R_i - 1$. Then the (key, value) pairs are hash partitioned among these R_i reducers according to the partition attributes in the key (Lines 12-15).

Reduce Phase: In the `Reduce` phase, each reducer obtains its computation tasks (the cuboids in the batch) by parsing the batch identifier in the value. The library sorts all the (key, value) pairs based on the key and passes them to the reduce function. The reduce function decomposes the tuple to process multiple cuboids in the batch. Each cuboid maintains a container to store the data received within one group. Once it gets all the data in one group, it materializes this group using the measure function. Or else, it caches the data in the container until it obtains all the data for one group (Lines 18-26). We develop multiple file emitters to write different views to different destinations.

Analysis: The total overhead of data reading and shuffling is as follows:

$$\sum_{i=0}^{b-1} B_i = Read(\mathbb{D}) + \sum_{i=0}^{b-1} Shuffle(B_i)$$

We only list the cost for data reading and shuffling as these costs can potentially be reduced. There are other overheads like writing overhead (write the view to DFS or

databases) which are not reducible. Thus, these overheads are omitted above.

Compared to the naive solution, `CubeGen` only incurs one data reading. The shuffle cost of batch B_i equals to the shuffle cost of the cuboid with the maximum number of dimensions in the batch plus the cost of adding one identifier in the intermediate result. Since we use a Bitmap for the identifier, this additional data size is relatively small. Therefore, the shuffling cost remains as only a part of the cost in the naive solution.

Handling Multiple Measure Attributes: We note that if there are more than one measure attributes (e.g. m_1, m_2, ..., m_n) that users want to aggregate, the multiple measure attributes can be processed in the same job. In particular, in the `CubeGen` algorithm, the multiple measure attributes can be put into the value together in the map phase as shown in Line 8 in Algorithm 4.1, instead of emitting n different (key, value) pairs. For instance, given n measure attributes, the value would be (m_1, m_2, ..., m_n). This guarantees that the multiple measure attributes can share the dimension attributes in the intermediate data. Thus, it minimizes the intermediate data size and removes significant data sorting and shuffling overheads compared to emitting multiple independent (key, value) pairs each of which is for one measure attribute.

The aggregation on each measure attribute can be conducted in the reducer by parsing and aggregating the different measures attributes from the value.

Handling Multiple Aggregation Functions: In many situations, users may want to materialize the data by applying multiple measure aggregation functions such as $\mathbb{M}_1$, $\mathbb{M}_2$, .., $\mathbb{M}_m$. We note that this materialization can also be conducted in the same job, instead of m jobs. Specifically, the multiple aggregation functions can be applied to the data in the reduce phase simply, as shown in Line 23 in Algorithm 4.1. This guarantees that all the operations before the reduce function can be shared among different measure functions.

Algorithm 4.2: **A Refresh Job in MR**

```
1 Function: Map(t)
2 # t is the tuple value from either 𝕍 or Δ𝕍
3 k ⇐ get dimension attributes from t
4 v ⇐ get aggregate value from t
5 emit(k,v)
6 Function: Reduce(k, {v_1, v_2})
7 emit(k, m(v_1, v_2))
```

4.5 View Maintenance

4.5.1 Supporting View Maintenance in MR

For non-distributive measures like MEDIAN, views can only be updated by reconstructing the cube from the entire dataset, i.e., $\mathbb{D} \cup \Delta\mathbb{D}$. Under the MR framework, the updated view can be obtained by issuing one MR job using our `CubeGen` algorithm to recalculate the cube over $\mathbb{D} \cup \Delta\mathbb{D}$.

The key problem with such a MR-based recomputation view updates is that reconstruction from scratch in MR is expensive because the base data (which is large and increases in size at each update) has to be reloaded to the mappers from the DFS and shuffled to the reducers for each view update.

For distributive measures like SUM, view updates can also be done by recomputation as is done in the non-distributive measures, and so we will not discuss this further. An alternative mechanism is to incrementally update the view through incremental computation with a propagate step (get $\Delta\mathbb{V}$ from $\Delta\mathbb{D}$) and a refresh step (merge $\mathbb{V}$ and $\Delta\mathbb{V}$).

Specifically, incremental view updates can be conducted using two MR jobs. The first propagate job generates $\Delta\mathbb{V}$ from $\Delta\mathbb{D}$ using our proposed `CubeGen` algorithm. The second refresh job merges $\mathbb{V}$ and $\Delta\mathbb{V}$.

The algorithmic description of the refresh job is given in Algorithm 4.2. Intuitively, the entire views can be partitioned to the mappers. Each mapper parses the dimension

attributes and the aggregate value from each tuple in the view and emits one (key, value) pair where the key is the dimension attributes and the value is the aggregate value (Lines 1-5). Then the (key, value) pairs with the same key are shuffled to the same reducer to conduct the merge operation (Lines 6-7). Note that the merge function can use the same reduce function as adopted in the cube construction.

The key problem with such a MR-based incremental computation view updates is the significant overheads incurred because the data has to be read/written/shuffled around the cluster multiple times. For instance, the materialized $\Delta\mathbb{V}$ from the propagate job has to be written back to the DFS, reloaded from the DFS and shuffled from mappers to reducers in the refresh job. Likewise, $\mathbb{V}$ has to be reloaded and shuffled around in the refresh job. These overheads make incremental computation expensive in the basic MR framework.

4.5.2 HaCube Design Principles

`HaCube` avoids the aforementioned overheads through storing and reusing the data between different jobs. We extend the MR framework to add a local store in the reducer node which is intended to store useful data in the local file system in a job. Thus, the task shuffled to the same reducer is able to reuse the data already stored there. In this way, the data is read directly from the local store (and thus significantly reducing the overhead that would have been incurred to read the data from the DFS and shuffle them from mappers).

We further extend the MR framework to develop a new task scheduler to guarantee that the same task is assigned to the same reducer node and thus the cached data can be reused among different jobs. Specifically, the task scheduler records the scheduling information by storing a mapping between the data partition number (corresponds to the task) and the TaskTracker (corresponds to the reducer node) and puts it to the task

scheduling factory from one job. When a new job is triggered to use the scheduling history from previous jobs, the task scheduler fetches and adopts the scheduling information from the factory to distribute the tasks. The scheduler automatically checks the situation of the over-loaded nodes and moves the task to other nodes.

In addition, two computation phases (`Merge` and `Refresh`) are added to support efficient view updates by conducting more computation with the cached data locally. The `Merge` phase is added to either cache the intermediate reduce input data in one job or preprocess the data between the newly arriving data and cached data before the `Reduce` phase. The `Refresh` phase is added to either cache the reduce output data in one job or postprocess the reduce result with the cached data after the `Reduce` phase.

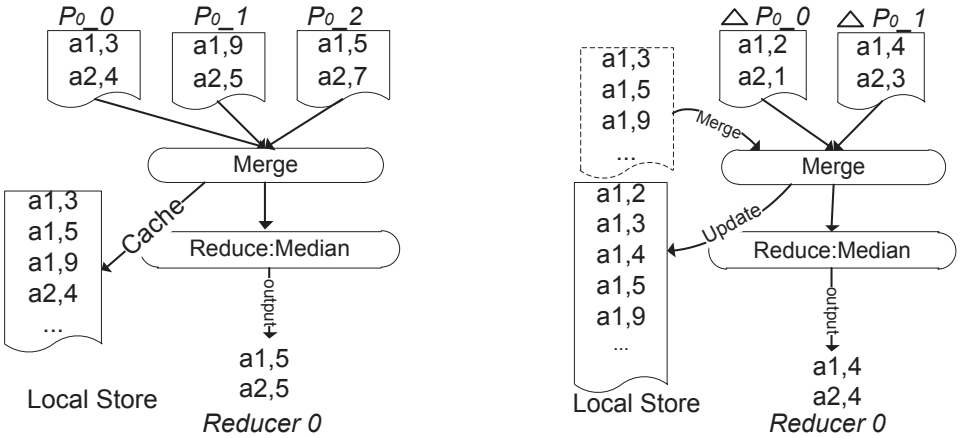
4.5.3 Supporting View Maintenance in HaCube

We are now ready to present how `HaCube` supports view maintenance.

Non-distributive Measures

`HaCube` performs a Map-Merge-Reduce (MMR) computation for the recomputation of view updates for non-distributive measures. In the initial cube materialization job, `HaCube` is triggered to cache the intermediate reduce input data to the local store. Specifically, the `CubeGen` algorithm partitions the cuboids computation task to each reducer.

For instance, Figure 4.5(a) shows an example of calculating the cuboid a for the MEDIAN measure. We assume that reducer 0 is assigned to process cuboid A. Thus, each mapper emits one sorted partition for reducer 0, such as P_0_0, P_0_1 and P_0_2. Here, each partition is a sequence of (dimension-value, measure-value) pairs, e.g., (a1, 3), (a2, 4) in Figure 4.5(a). When the partitions are shuffled to reducer 0, it performs a merge-sort (the same as MR does) to sort all the partitions based on the key in the `Merge` phase.



(a) Cube materialization in a Map-Merge-Reduce computation paradigm

(b) View maintenance in a Map-Merge-Reduce computation paradigm

Figure 4.5: Recomputation for MEDIAN in HaCube

Note that the global sorted data is stored in the local disk. Then, the data are supplied to the reduce function to get the view V ($< a1, 5 >$ and $< a2, 5 >$) for MEDIAN which is emitted and written to the DFS.

Since recomputation needs to use all the base data for view updates, the intermediate sorted reduce input data in the `Merge` phase, which is deleted in MR, will be cached in the local store for subsequent reuse when the `Reduce` phase finishes in `HaCube`. This guarantees the atomicity of the operation - if the reduce task fails, the data will not be written to the local store. Meanwhile, the scheduling information is recorded.

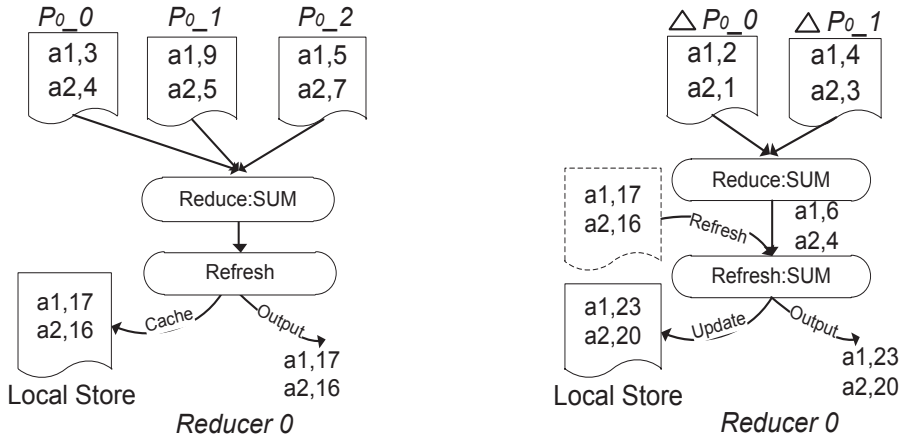
A view maintenance job is launched when $\Delta\mathbb{D}$ is added for view updates. Intuitively, this view update job conducts a cube materialization job using the `CubeGen` algorithm based on $\Delta\mathbb{D}$. It differs from the initial materialization job (on $\mathbb{D}$) in the scheduling and the `Merge` phase. For task scheduling, instead of randomly distributing the tasks to reducer nodes, it distributes the tasks according to the scheduling information stored from the initial cube materialization job to guarantee that the same tasks are processed at the same reducer.

For instance, the emitted partitions for cuboid A (ΔP_0_0 and ΔP_0_1) from mappers are scheduled to the same node running reducer 0 as shown in Figure 4.5(b). In the `Merge` phase, since the base data is already cached in the local store, `HaCube` merges the delta partitions with the base data by directly reading from the local store in the `Merge` phase. Recall that the cached data is the sorted reduce input data from the previous job, and so it has the same format as the delta partition. Thus, it can be treated as a local partition and a global merge-sort can be conducted. Then the merged base and delta data will be supplied to the reduce function for a view update in the `Reduce` phase. When the `Reduce` phase finishes, the local store is updated with both the base and delta data (becoming an updated base dataset) for further view update use.

In `HaCube`, view updates via recomputation do not need the base data to be reloaded from the DFS and shuffled from mappers to reducers. Thus, the overhead is reduced and view updates are performed more efficiently. Recall that in the `CubeGen` algorithm, cuboids are batched together. Each reducer executes part of the tasks for computing multiple cuboids. `HaCube` caches intermediate (key, value) pairs for the batch instead of caching the data for each cuboid.

To cache the intermediate sorted reduce input data, one naive way is to push them to the local store. However, this incurs much overhead, since moving a large amount of data is expensive. In our case, since the intermediate sorted data are maintained in a temporary file in the local disk, instead of moving the data, we simply register the file to the local store. The local store only needs to maintain the cached data location for further usage. When a job finishes, the `HaCube` framework will not delete the temporary files which have been registered for caching in local store. We note that these temporary files are deleted in the traditional MR framework. With this optimization, `HaCube` does not incur any extra data movement or writing, and thus incurs negligible overhead. As we shall see, the experimental study shows that there is almost no overhead added for

caching the data compared to the traditional MR framework that does not cache the data.



(a) Cube materialization in a Map-Reduce-Refresh computation paradigm

(b) View maintenance in a Map-Reduce-Refresh computation paradigm

Figure 4.6: Incremental computation for SUM in HaCube

Distributive Measures

`HaCube` performs a `Map-Reduce-Refresh` (MRR) computation for the incremental computation view updates for distributive measures. The initial cube materialization job is triggered to cache the reduce output $\mathbb{V}$ to the local store after the `Reduce` phase. Intuitively, `HaCube` conducts the materialization job using our `CubeGen` algorithm. However, after the `Reduce` phase, a `Refresh` phase is triggered and used to cache the reducer output data, the view $\mathbb{V}$ to the local store.

For instance, Figure 4.6 (a) shows an example of initial cube materialization indicating a partial computation for the cuboid A for a SUM measure. In this job, $\mathbb{V}$ ($< a1, 17 >$ and $< a2, 16 >$) is cached to the local store in the node running reducer 0 in the `Refresh` phase. The scheduling information is also recorded so that it can be used during view updates.

When $\Delta\mathbb{D}$ is added, `HaCube` can conduct both the propagate and refresh steps in

one view update job. Since V is already cached in the reducer node, this job executes in a MRR computation paradigm where MR (Map-Reduce) phases obtain ΔV based on $\Delta\mathbb{D}$ (propagate step) and the `Refresh` phase merges ΔV with V locally (refresh step). Specifically, it runs the `CubeGen` algorithm on $\Delta\mathbb{D}$ using the same scheduling plan as the initial job.

For instance, the data partitions ΔP_0_0 and ΔP_0_1 are shuffled to the same processing node with reducer 0 as shown in Figure 4.6 (b). The ΔV is obtained via the `Reduce` phase.

Following the `Reduce` phase, a `Refresh` phase is invoked to refresh the view by merging $\mathbb{V}$ and $\Delta\mathbb{V}$ locally. Since the intermediate data is sorted, the cached $\mathbb{V}$ is also in sorted order. Likewise, the $\Delta\mathbb{V}$ tuples are also output in sorted order, thus can be efficiently merged with the cached $\mathbb{V}$ to update the view. $\mathbb{V}$ is read sequentially chunk-by-chunk into the memory for the merging process. Thus, once the `HaCube` collector receives one $\Delta\mathbb{V}$ tuple d, d searches for the right position to insert (if d does not exist in $\mathbb{V}$) into or merge (if d already exists in $\mathbb{V}$) with $\mathbb{V}$. We note that normally the merging operation can be done using the same measure operator. Once a chunk of $\mathbb{V}$ is searched to the end, the next chunk of $\mathbb{V}$ will be read into memory until the whole $\mathbb{V}$ has been merged. Meanwhile, the cached view in the local store will be updated with the updated one.

For instance, in Figure 4.6 (b), the `Reduce` phase calculates the $\Delta\mathbb{V}$ ($< a1, 2 >$ and $< a2, 4 >$) according to the $\Delta\mathbb{D}$. In the `Refresh` phase, the updated view ($< a1, 23 >$ and $< a2, 20 >$) is obtained by merging the old view $\mathbb{V}$ ($< a1, 17 >$ and $< a2, 16 >$) cached in the local store and the $\Delta\mathbb{V}$, and is then updated to local store as well.

In this mechanism, the entire refresh step only needs to locally read the old view $\mathbb{V}$ once instead of being reloaded from the DFS and shuffled to reducers. Thus, `HaCube` significantly reduces the view maintenance overhead for distributive measures.

4.6 Other Issues

4.6.1 Fault Tolerance

Since `HaCube` is built on the MR framework, it preserves the fault-tolerance mechanisms of the MR framework. For instance, the data is replicated in DFS and thus is safe when nodes fail. When a map task fails, the framework schedules the task on another free mapper in the system instead of restarting the whole job.

In addition, we provide an additional fault tolerance strategy to guarantee data availability in the reducer nodes in `HaCube`. The caching mechanism plays an important role in improving the efficiency of data cube analysis. It is important to make sure that the cached data in the reducer node is accessible when a subsequent job arrives. We handle two kinds of failures in the reducer nodes, including the recoverable and the unrecoverable reducer failures.

Recoverable Failures: Recoverable reducer failures include the task failure and reducer node failure. These failures can be recovered once the corresponding failed task or node is restarted. When the task fails in the reducer node, the scheduler kills the task and reschedules it. If the job does not need to use the data in the local store, it will be scheduled to any reducer node. Otherwise, it is scheduled to the same node for data locality. The local store is in a persistent local file system on the reducer node. Thus, after restarting this task, the data is still readable. Similar to the reduce task failure, if the reducer node fails, the data is still accessible after the node is restarted.

Unrecoverable Failures: Unrecoverable reducer node failures happen when the reducer node is totally corrupted and not usable at all. In this case, the data in the local store will be lost. To handle this failure, alternative recovery strategies can be adopted for incremental computation and recomputation.

For incremental computation, recovery is straightforward. This is because the views

from previous jobs cached at the local store are also stored in the DFS. Thus, when the reducer node is corrupted, the views can be easily recovered from the DFS.

Under recomputation, the local store caches the sorted intermediate reduce input data from the merge phase. To handle node failures, `HaCube` adopts a **lazy checkpointing** strategy - a snapshot of the local store is stored to the DFS periodically. For cube analysis, if we make a snapshot of the cached data after each view update, it provides the fastest recovery. This is to ensure that data can be directly recovered from the previous view update stage. However, it is costly to perform checkpointing for each update.

On the other hand, if no snapshots are taken, once a node fails, we have to recompute it from scratch which is also computationally expensive. Instead, we advocate an intermediate solution that takes a snapshot after every s view updates where s can be set by the users according to the view update and computer failure frequencies in their cluster. With such a lazy checkpointing scheme, if a failure happens, the system can recover by using the most recent snapshot and the new delta data added after the last checkpointing. Thus, `HaCube` only needs to store the latest snapshot and the data after the snapshot instead of storing all the base data from the beginning.

4.6.2 Storage Cost Discussion

We argue that `HaCube`'s storage costs are acceptable.

- First, we can also reduce the number of replicas stored in the DFS accordingly. The data cached in the local store can essentially be viewed as one replicated dataset.

- Second, `HaCube` only needs to cache one copy of the dataset for different measures in each computation model. Recall that all the measures can be processed together. Thus, for all measures issuing recomputation, the cached sorted raw data is able to serve all of them. For all measures issuing incremental computation, the

cached view data can be stored together to reduce storage overhead. For instance, assume that both SUM and MAX need to be calculated, we can store these two views together in the format of <dimension attributes, SUM, MAX> instead of maintaining them independently.

4.7 Performance Evaluation

Algorithm 4.3: **A Naive Algorithm**

```
1 Function: Map(t)
2 # t is the tuple value from the raw data
3 # C_i is the cuboid in the cube lattice
4 k ⇐ get dimension attributes in C_i from t
5 v ⇐ get measure attribute from t
6 emit(k,v)
7 Function: Reduce(k, {v_1, v_2, ..., v_m})
8 # m is the measure function
9 emit(k, m(v_1, v_2, ..., v_m))
```

We implement `HaCube` by modifying Hadoop. We evaluate `HaCube` on the Longhorn Hadoop cluster with 40 nodes in TACC (Texas Advanced Computing Center) [4]. Each node consists of 2 Intel Nehalem quad-core processors (8 cores) and 48GB memory.

We perform our studies on the classical dataset generated by TPC-D benchmark generators [5]. The TPC-D benchmark offers a rich environment representative of many decision support systems. We study the cube views on the fact table, $lineitem$, in the TPC-D benchmark. We use the attributes $l\_partkey$, $l\_orderkey$, $l\_s\ uppkey$ and $l\_shipdate$ as the dimension attributes and the $l\_quantity$ as the measure attribute. We choose MEDIAN and SUM as the representative functions for non-distributive and distributive measures respectively. We report the result based on the average execution time of three runs in each experiment.

4.7.1 Cube Materialization Evaluation

Baseline Algorithms

As argued in [54], existing parallel algorithms (like BPP and PT [57]) for cube computation are designed for small PC clusters and are unable to take advantage of the MR infrastructure. Therefore, we will not consider these algorithms here. Instead, we choose two alternative algorithms which are widely used in MR as the baseline algorithms to evaluate data reading and shuffling gains in our `CubeGen` algorithm.

A Naive Algorithm: As mentioned in Section 4.4, the most straightforward way to materialize all cuboids, denoted as `MulR_MulS` (short for multiple read and multiple shuffle) is to compute one cuboid in a MR job at a time.

Algorithm 4.3 shows the pseudo code of `MulR_MulS`. Given base data $\mathbb{D}$ and cuboid C_i, assume that the intermediate data emitted from the `Map` phase is $\mathbb{D}_i$ including the projected dimension attributes and measure attribute in C_i. The read and shuffle overhead in C_i is $Read(\mathbb{D})+Shuffle(\mathbb{D}_i)$. Thus, the total read and shuffle overhead of processing the entire cube is as follows:

$$\sum_{i=1}^{2^n} Cost(C_i) = 2^n * Read(\mathbb{D}) + \sum_{i=1}^{2^n} Shuffle(\mathbb{D}_i).$$

This baseline algorithm is chosen to study the benefit of removing multiple data read overheads by performing the materialization in one MR job.

A Non-batching Algorithm: We also develop another possible materialization solution, `SingR_MulS` (short for single read and multiple shuffle). `SingR_MulS` differs `CubeGen` by not batching the cuboids. In other words, the mapper emits each (key,value) pair for one cuboid and each cuboid is shuffled and processed independently. Note that we adapt our proposed load balancing approach to this algorithm as well to improve its performance.

Algorithm 4.4 provides the pseudo code of `SingR_MulS`. The total read and shuffle

Algorithm 4.4: **A Non-batching Algorithm**

```
1  Function: Map(t)
2  # t is the tuple value from the raw data
3  for Each cuboid C_i in the cube lattice do
4      k_i ⇐ get dimension attributes in C_i from t
5      v_i ⇐ get measure attribute from t
6      emit(k_i, v_i)
7  Function: Partitioning(k_i, v_i)
8  Let R_i be the number of reducers assigned for C_i
9  Let S_i be Σ_{j=0}^{i-1} R_j
10 Let attr be the partition attributes in C_i
11 return S_i + hash(attr, R_i)
12 Function: Reduce(k, {v_1, v_2, ..., v_m})
13 # m is the measure function
14 emit(k, m(v_1, v_2, ..., v_m))
```

overhead of completing the entire cube is as follows:

$$Read(\mathbb{D}) + \sum_{i=1}^{2^n} Shuffle(\mathbb{D}_i).$$

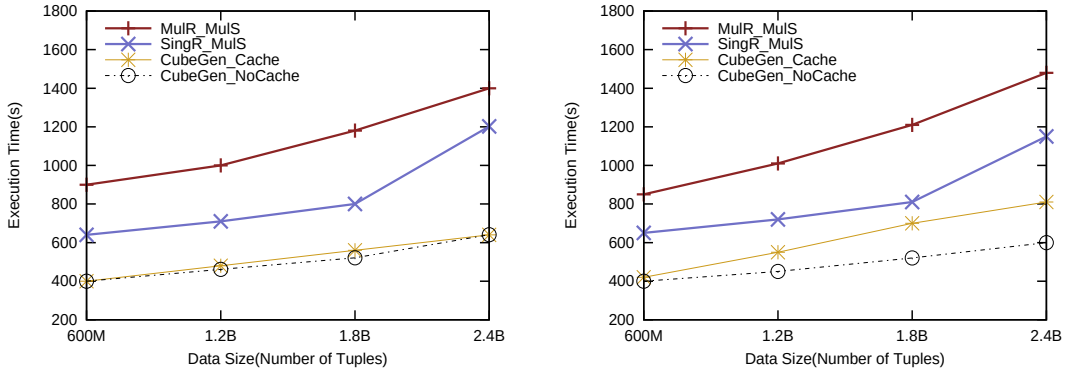
The purpose of this baseline algorithm is to study the benefit of sharing the shuffle and computation through batching cuboids.

4.7.2 Cube Materialization Evaluation

The following set of experiments are conducted on a 35-node cluster. We vary the data size from 600M (Million) to 2.4B (Billion) tuples.

Efficiency Evaluation

We first evaluate the performance improvement of the `CubeGen` algorithm over the two baseline algorithms for initial cube materialization. We study two versions of the `CubeGen` algorithm where `CubeGen_Cache` caches the data and `CubeGen_NoCache` does not. This provides insights into the overhead of caching the data at the local store.



(a) MEDIAN (b) SUM

Figure 4.7: *CubeGen* Performance Evaluation for Cube Materialization

Figures 4.7 (a) and (b) show the execution time of all four algorithms for MEDIAN and SUM operations respectively. As expected, for both MEDIAN and SUM, our `CubeGen`-based algorithms outperform `SingR_MulS` and `MulR_MulS` by a big margin. `MulR_MulS` performs worse than `SingR_MulS` since it needs to read the base data multiple times and process each cuboid independently in multiple jobs.

The results show that `CubeGen_NoCache` outperforms `SingR_MulS` by almost 50%. We observe that `SingR_MulS` incurs high shuffling cost and processing time in the map phase. This is because it generates a much larger number of intermediate data than `CubeGen` which incurs significant overhead of sorting as well as shuffling. The results show that our proposed `CubeGen`-based schemes are much more efficient as they read the data once and optimize the processing of cuboids by batching them.

Impact of Caching Data

Figure 4.7 (a) and (b) also depict the impact of caching data. For MEDIAN, `CubeGen_Cache` caches the reduce input data at the local store. From the result, we can see that the execution time of `CubeGen_Cache` is almost the same as `CubeGen_NoCache` as shown in Figure 4.7 (a). This confirms that our optimization to cache the data through file registration instead of actual data movement does not cause much overhead.

For SUM, we observe that CubeGen_Cache performs worse than CubeGen_No-Cache. This is not surprising as the former needs to write an extra view to the local file system. However, even though CubeGen_Cache incurs around 16% overhead to cache the view, as we will see in Section 4.7.3, it is superior to CubeGen_NoCache when it comes to view updates.

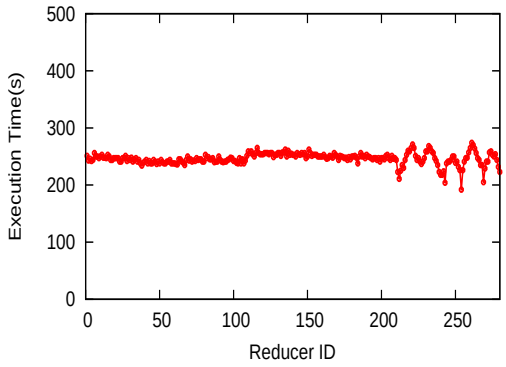


Figure 4.8: The load balancing on 280 reducers

Load Balancing

We next show how the $\mathcal{LBCCC}$ load balancing scheme works. The $\mathcal{CCC}$ learning job is conducted using 2 machines and 1GB testing data generated by the benchmark generator. Then each reducer execution time is recorded to generate a load balancing plan for the Cube-Gen algorithm.

We observe that the $\mathcal{LBCCC}$ scheme is able to balance the load very well in CubeGen. Figure 4.8 shows the load situation at each reducer when CubeGen_NoCache processes 1.2B tuples. We record the Reduce phase execution time of each reducer among all the 280 reducers. We find that 95% of the reducers complete their processing within a 10-second difference in execution time.

For the remaining 5% of the reducers, as shown in the tail part of the execution time line in Figure 4.8, they take 35 seconds more or less than the others. This may be

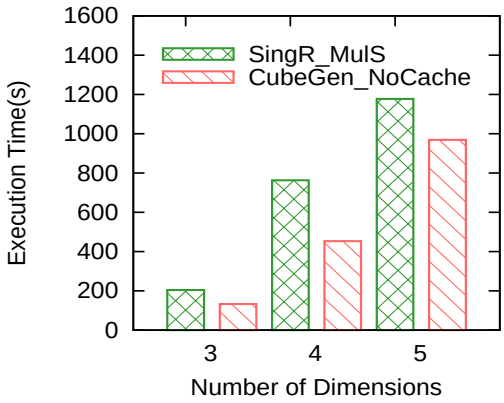


Figure 4.9: Impact of Number of Dimensions

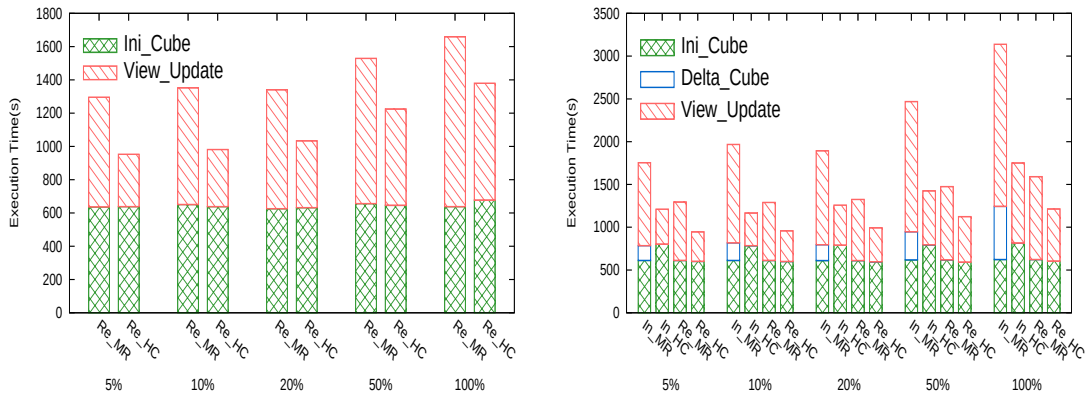
caused by the dimension data hash code skew. We find out that reducers 211 to 280 are assigned to process the same batch. Recall that within these 69 reducers, the data is hash partitioned to each reducer. Thus, if the hash codes of partition attributes are skewed, some reducers will get more data than others. However, we can see that the average execution time of these 69 reducers is almost the same as the others which confirms that our $\mathcal{LBCCC}$ does provide each batch the right number of computation resources. One possible solution to handle this skew challenge is to adopt the partitioning mechanisms such as range partitioning to better allocate the data evenly.

Impact of Number of Dimensions

We further analyze the impact for cube materialization while varying the number of dimensions from 3 to 5. Our current dataset has 4 dimensions: $l\_partkey$, $l\_orderkey$, $l\_s\ uppkey$ and $l\_shipdate$. To generate a 3-dimension dataset, we generate the data by removing the $l\_shipdate$ from current dataset. While for the 5-dimension dataset, we generate the data by adding another dimension, $l\_receiptdate$ to current dataset.

Figure 4.9 shows the execution time of `SingR_MulS` and `CubeGen_NoCache` for SUM on 1.2B tuples. Not surprisingly, increasing the number of dimensions increases the cube building time. The results show that `CubeGen_NoCache` outperforms

SingR_MulS in all these three cases.



(a) View Maintenance for MEDIAN (b) View Maintenance for SUM

Figure 4.10: HaCube View Maintenance Efficiency Evaluation

4.7.3 View Maintenance Evaluation

Efficiency Evaluation

We next study the efficiency of performing the view maintenance in HaCube compared with the Hadoop MR framework. We fix $\mathbb{D}$ with 2.4B tuples in the initial cube materialization and vary the size of $\Delta\mathbb{D}$ from 5% to 100% of $\mathbb{D}$ for view updates on a 35-node cluster.

Figure 4.10 (a) shows the execution time for both the initial cube materialization (Ini_Cube) and the view updates (View_Update) time for MEDIAN. In this set of experiments, we adopt recomputation for view updates of MEDIAN under MR (Re_MR) and HaCube (Re_HC). From the results, it is clear that HaCube is superior over MR in terms of view updates. When $\Delta\mathbb{D}$ is 5%, Re_HC takes 45% execution time of Re_MR. Even when $\Delta\mathbb{D}$ is 100%, Re_HC is 37% faster than Re_MR. The gains come from avoiding reloading and reshuffling $\mathbb{D}$ among the cluster. Thus, the larger $\mathbb{D}$ is, the bigger the benefit will be.

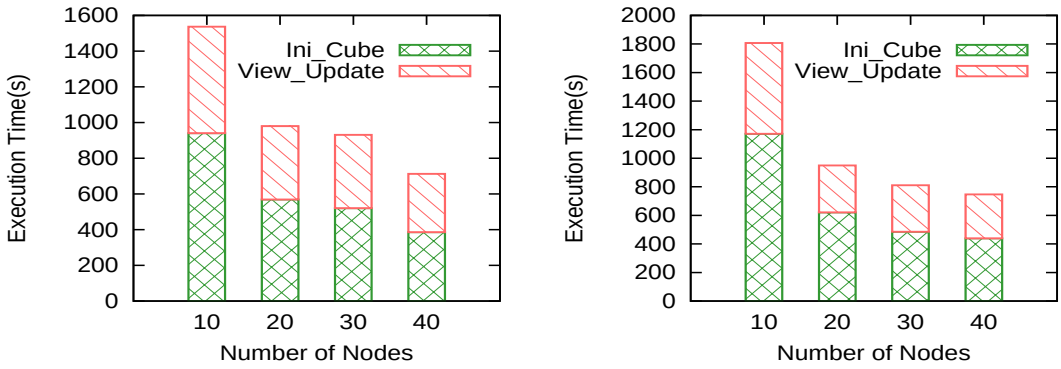
Figure 4.10 (b) depicts the results for SUM. Since view maintenance for distributive measures can either be done by incremental computation or recomputation, we adopt both approaches to update the view. The following notations are used to denote the respective schemes: `In_MR` and `Re_MR` are MR-based methods using incremental computation and recomputation respectively, and `In_HC` and `Re_HC` are the corresponding schemes under `HaCube`.

`In_MR` and `Re_MR` are implemented in the way described in Section 4.5.1. In `In_MR`, `Delta_Cube` (in the figure) corresponds to the propagate job to generate the delta view and `View_Update` (in the figure) is the refresh job. We observe that for both incremental computation and recomputation, `HaCube` outperforms MR. The result shows that, for view maintenance cost, `In_HC` achieves 65% and 55% execution time savings for $\Delta\mathbb{D}$ size in 5% and 100% compared to `In_MR`. `Re_HC` outperforms `Re_MR` in a similar level as the recomputation for MEDIAN. We observe that incremental computation performs worse than recomputation in both MR and `HaCube`. While this seems counter-intuitive, our investigation reveals that DFS does not provide indexing support; as such, in incremental computation, the entire view which is much larger than the base data (in our experiments) has to be accessed.

As future work, we will integrate indexing techniques into `HaCube` so that view update only needs to visit the tuples which are changed to improve the performance further for incremental computation. The results also demonstrate that the smaller the $\Delta\mathbb{D}$ is, the more effective is the `HaCube` paradigm.

Impact of Parallelism

We further analyze the impact of parallelism on `HaCube` for both cube materialization and view update while varying the number of computation resources from 10 nodes to 40 nodes. The experiments are conducted on a $\mathbb{D}$ with 1.8M tuples and a $\Delta\mathbb{D}$ which



(a) Impact of Parallelism: MEDIAN (b) Impact of Parallelism: SUM

Figure 4.11: Impact of Parallelism for View Maintenance

is 20% of $\mathbb{D}$.

Figures 4.11 (a) and (b) report the execution time for MEDIAN and SUM. In this experiment, incremental computation is used for view updates of SUM. We observe that for both recomputation and incremental computation, `HaCube` scales linearly on the testing data set from 10 to 20 nodes, where the execution time almost reduces to half when the resources are doubled. Beyond 30 nodes, the benefit of parallelism decreases a little bit. This is reasonable, since the entire overheads include two parts, the overheads of the setup and runtime the framework and the overheads of the cube computation, the first one may reduce the benefits of increasing the computation resources while the latter one is not big enough.

4.8 Summary

In this chapter, we have investigated the problem of designing a scalable, efficient and practical data cube analysis system in a distributed environment. We made one step towards building such a system by extending the MapReduce framework. We have designed and implemented `HaCube`, an extension of MapReduce, to support data cube analysis on large-scale data. We showed how to conduct the cube computation by fa-

cilitating the sorting feature under MapReduce-like frameworks and how to batch and minimize the overhead to salvage partial work done for efficient cube materialization. We also proposed a general and effective load balancing scheme which is able to balance the load well. We further developed a new computation paradigm in `HaCube` through caching/reusing intermediate data for an efficient view maintenance. Experimental results have demonstrated that `HaCube` has significant performance improvement over Hadoop.

CHAPTER 5

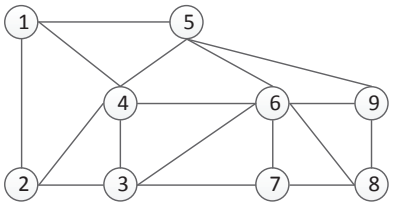
GRAPH CUBE ANALYSIS

5.1 Overview

In the previous chapter, we mainly focus on developing techniques for data cube analysis based on the relational data in the traditional OLAP and data warehousing. In this chapter, we extend the OLAP cube techniques to a more complicated structured data, attributed graphs where both vertex and edge are associated with attributes.

The expressive power of *attributed graphs* makes them attractive in modeling a variety of information networks, such as the Web, blogs and social networks (e.g. Facebook, LinkedIn, Twitter) [65]. Attributed graphs model these information networks as follows: each individual object with its associated information is represented as a vertex with vertex attributes, and the relationships between two objects are captured as edges between two vertices with associated edge attributes. By analyzing the attributed graph of an information network, we may acquire accurate and more explicit insight of the real world and make better decisions.

Example 5.1. *Consider a social network which typically contains a wealth of individual user information (like profile information) and relationship information (like the connection information between different users). Fig. 5.1 provides a simple attributed graph that is used to model the information extracted from a social network. Fig. 5.1 (a) presents the underlying graph structure involving 9 vertices each of which represents one individual user with a user ID, and 17 edges each of which indicates one relationship between two users. Fig. 5.1 (b) shows a vertex attribute table describing each individual's profile information including Gender, Nation and Profession. Fig. 5.1 (c) shows an edge attribute table describing the relationship information between different individuals including the date they connected, their relationship types and strength, where sV and tV are the two vertex IDs of each edge.*



(a) Graph Structure

ID	Gender	Nation	Profession
1	male	USA	professor
2	female	USA	doctor
3	male	China	engineer
4	male	SG	engineer
5	female	SG	professor
6	male	SG	doctor
7	female	China	engineer
8	female	China	doctor
9	male	USA	doctor

(b) Vertex Attribute Table

sV	tV	Date	Type	Strength
1	2	2008	Family	9
1	4	2010	Friend	7
1	5	2011	Colleague	9
2	3	2011	Friend	7
2	4	2011	Friend	4
3	4	2008	Friend	8
3	7	2011	Colleague	8
3	6	2012	Family	3
4	5	2009	Family	9
4	6	2010	Friend	6
5	6	2010	Family	8
5	9	2012	Friend	8
6	7	2008	Friend	7
6	8	2012	Colleague	9
6	9	2012	Friend	5
7	8	2011	Friend	5
8	9	2012	Colleague	8

(c) Edge Attribute Table

Figure 5.1: A running example of an attributed graph

The aforementioned information-enhanced attributed graph is a valuable information resource for information discovery and decision making. We identify several categories of queries that users may be interested in:

Category 1: Discovering knowledge over the vertex or edge attributes. This category involves queries that can be answered from either the vertex attributes or the edge

attributes. Referring to our running example in Fig. 5.1, sample queries are "What is the percentage of users among different professions in this network?" and "How many relationships appeared in 2012?"

Category 2: Integrating knowledge over both the vertex and edge attributes. Queries in this category require integrating information from both the vertex and edge attributes. An example query is "What is the trend of the number of relationships appearing between USA and SG (Singapore) in the last 3 years?"

Category 3: Investigating an aggregated or summarized graph from different granularities. This category requires to see a coarse-grained graphs based on different dimensional spaces. For instance, users may want to see how people connect with each other from different communities, such as gender group. Such queries are useful when the base (original source) graphs are too massive for the underlying relationships to be observed. Instead, the aggregated graphs offer greater ease in discovering the underlying information.

Now, for large attributed graphs, it is computationally expensive to evaluate these queries from the base graphs. As such, it is critical to develop better query and decision making support over attributed graphs. In this work, we adopt a two-pronged approach to address this challenge. First, we observe that traditional data cubes have been successfully deployed to speed up OLAP query processing in RDBMS [31]. However, traditional data cube model is not applicable to graphs as it does not capture the graph structures. Thus, we need to design a conceptual graph cube model that supports the queries in all the three categories. Second, for large attributed graphs, parallelism is an effective and promising approach to ensure acceptable response time. As such, we seek to develop parallel algorithms for evaluating queries represented under our graph cube model.

Recently, numerous distributed graph processing systems have been proposed, such

as Pregel [48], GraphLab [1], PowerGraph [29]. These systems are vertex-centric and follow a bulk synchronous parallel model (where vertices send messages to each other through their connections) tailored for graph processing operators that require iterative graph traversal, such as page rank, shortest path, bipartite matching, semi-clustering and so on [48]. As such, they are not suited for our graph query types that aggregate data over the vertices and edges attributes rather than graph traversal. In fact, adopting these systems for aggregation operations will incur high overhead for message (carrying the attributes values) passing across the graph to find the vertices/edges with the same attribute values. As OLAP cube constructions are not iterative operations, the MapReduce (MR) computation paradigm, which has been successfully demonstrated to be effective for large graph mining [19][39], turns out to be a better fit [22].

We are thus motivated to develop a new graph OLAP and warehousing model over attributed graphs as well as to develop an efficient MR-based parallel computation algorithm for the graph cube computation.

Our major contributions are summarized as follows. First, we propose a new conceptual graph cube model, Hyper Graph Cube, to extend decision making services on attributed graphs. Hyper Graph Cube is able to capture queries of all the aforementioned three categories into one model. Moreover, the model supports a new set of OLAP Roll-Up/Drill-Down operations on attributed graphs. Second, we propose several optimization techniques to tackle the problem of performing an efficient graph cube computation under the MR framework. a), our self-contained join strategy can reduce I/O cost. It is a general join strategy applicable to various applications which need to pass a large amount of intermediate joined data between multiple MR jobs. b), we combine cuboids to be processed as a batch so that the intermediate data and computation can be shared. c), a cost-based optimization scheme is used to further group batches into bags (each bag is a subset of batches) so that each bag can be processed efficiently using a single

MR job. d), a MR-based scheme is designed to process a bag. Third, we introduce a cube materialization approach, MRGraph-Cubing, that employs these techniques to process large scale attributed graphs. To the best of our knowledge, this is the first parallel graph cubing solution over large-scale attributed graphs under the MR-like framework. Finally, we conduct extensive experimental evaluations based on both real and synthetic data. The experimental results demonstrate that our parallel Hyper Graph Cube solution is effective, efficient and scalable.

The rest of this chapter is organized as follows: Section 5.2 presents our Hyper Graph Cube Model, followed by discussions on the query support and OLAP operations. In Section 5.3, we introduce a naive MR-based graph cube computation scheme. Section 5.4 provides our proposed scheme. In Section 5.5, we report the experimental results. Finally, Section 5.6 summarizes this chapter.

5.2 Hyper Graph Cube Model

An attributed graph is able to model various information networks by adding attributes to each vertex and edge. We first provide a formal definition of an attributed graph.

Definition 5.1. ***Attributed Graph:*** *An attributed graph, G, is a graph denoted as G=(V, E, A_v, A_e), where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and A_v = (A_{v1}, A_{v2}, ..., A_{vn}) is a set of n vertex-specific attributes, i.e. $\forall u \in V$, there is a multidimensional tuple $A_v(u)$ denoted as $A_v(u)$ = ($A_{v1}(u)$, A_{v2}(u), ..., A_{vn}(u)), and A_e=(A_{e1}, A_{e2}, ..., A_{em}) is a set of m edge-specific attributes, i.e. $\forall e \in E$, there is a multidimensional tuple $A_e(e)$ denoted as $A_e(e)$= $(A_{e1}(e), A_{e2}(e), ..., A_{em}(e))$.*

To develop graph OLAP and warehousing, we formally define two types of dimensions in attributed graphs as follows:

Definition 5.2. ***Vertex Dimensions:*** *With regard to the attributed graph defined in Definition 5.1, the set of n vertex-specific attributes (A_{v1}, A_{v2}, ..., A_{vn}) are called the vertex dimensions, or V-Dims for short.*

Definition 5.3. ***Edge Dimensions:*** *With regard to the attributed graph defined in Definition 5.1, the set of m edge-specific attributes (A_{e1}, A_{e2}, ..., A_{em}) are called the edge dimensions, or E-Dims for short.*

Take Fig. 5.1 as an example. Fig. 5.1 indicates an attributed graph modeling the user information in the social network. In this example, each vertex is associated with three V-Dims (Gender, Nation, Profession) and each edge is associated with three E-Dims (Date, Type, Strength).

In a graph warehousing context, the graph structure-related characteristics can also be extracted as the dimensions for analysis. For instance, the vertex degree is an important graph structure characteristic that indicates the number of friends one individual has in a social network. Therefore, the vertex degree can be considered as one V-Dim for materialization which may extend the utility of the graph warehousing to support more queries.

To support queries in category 1, the data can be aggregated along the V-Dims or the E-Dims alone while omitting the graph structure. Similar to the philosophy of the traditional data cubes, a measure can be calculated by aggregating all the data tuples from the vertex or edge attribute table whose dimensions are of the same values. In other words, a measure can be calculated as $\Gamma_v(G_v(v'))$ or $\Gamma_e(G_e(e'))$, where $G_v(v')$ and $G_e(e')$ are the group of vertex or edge tuples with the same dimension values v' on V-Dims or e' on E-Dims, and $\Gamma_v(\cdot)$ and $\Gamma_e(\cdot)$ are the measures, such as vertex count, edge count, centrality, degree, diameter etc.

Basically, this is to construct two small cubes along either the V-Dims or the E-Dims. We refer to the aggregation based on V-Dims or E-Dims as V-Agg or E-Agg respectively.

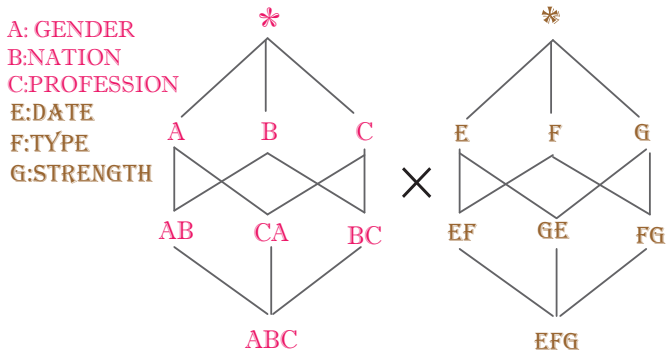


Figure 5.2: The V-Agg lattice cartesian product the E-Agg lattice

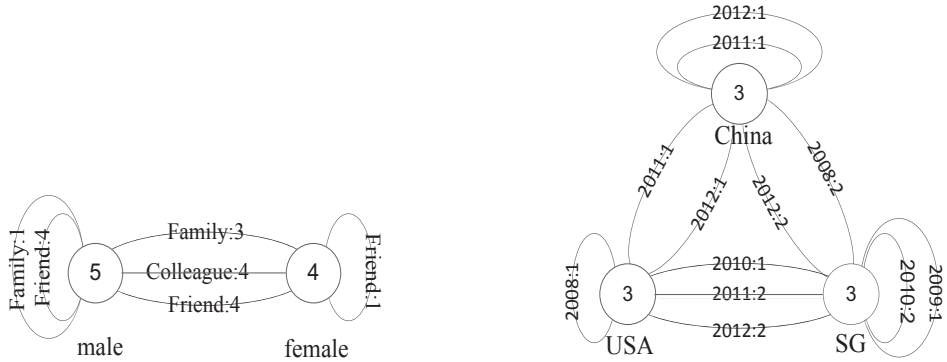
For instance, all the V-Aggs and E-Aggs of the given vertex and edge attribute tables in Fig. 5.1 can be represented as two lattices as shown in the LHS and RHS of Fig. 5.2.

However, to support those queries in categories 2 and 3, the graph can be aggregated from two aspects: aggregating the vertices along the V-Dims and aggregating the edges along the E-Dims while maintaining the graph structure. In so doing, the aggregation of graphs will be an aggregate graph which can be formally defined as follows.

Definition 5.4. ***Aggregate Graph:*** *Given an attributed graph G=(V, E, A_v, A_e) and a possible vertex aggregation $A_v^{'}$=($A_{v1}^{'}$, $A_{v2}^{'}$, ..., $A_{vn}^{'}$) and a possible edge aggregation $A_e^{'}$=($A_{e1}^{'}$, $A_{e2}^{'}$, ..., $A_{em}^{'}$) where $A_{vi}^{'}$ equals to A_{vi} or * and $A_{ei}^{'}$ equals to A_{ei} or *, the aggregate graph w.r.t. $A_v^{'}$ and $A_e^{'}$ is a weighted graph $G^{'}$=($V^{'}$, $E^{'}$, $W_{v'}$, $W_{e'}$) where*

- *$V^{'}$ is a set of condensed vertices each of which is associated with a group of vertices $G_v(v^{'})$ such that $\forall v \in V$, there exists one and only one $v^{'} \in V^{'}$ such that $A_v^{'}(v)$=$A_v^{'}(v^{'})$ and $v \in G_v(v^{'})$. The weight of $v^{'}$, $w(v^{'})$ = $\Gamma_v(G_v(v^{'}))$, where $\Gamma_v(\cdot)$ is an aggregate function based on a group of vertices.*

- *$E^{'}$ is a set of condensed edges each of which is associated with a group of edges $G_e(e^{'})$ such that $\forall e = (u, v) \in E$, there exists one and only one $e^{'} = (u^{'}, v^{'}) \in E^{'}$ such that $u \in G_v(u^{'})$, $v \in G_v(v^{'})$, $A_e^{'}(e) = A_e^{'}(e^{'})$ and $e \in G_e(e^{'})$. The weight*



(a) Aggregate graph on <Gender, Type> (b) Aggregate graph on <Nation, Date>

Figure 5.3: Aggregate Graphs

of $e^{'}$, $w(e^{'})$=$\Gamma_e(G_e(e^{'}))$, *where* $\Gamma_e(\cdot)$ *is an edge aggregate function on a group of edges.*

Note that in this definition, in $A^{'}_{v1}$, $A^{'}_{v2}$, ..., $A^{'}_{vn}$, *at least one of the dimensions is not *. Similarly, at least one of the dimensions in* $A^{'}_{e}$ *is not *.*

We refer to such an aggregation based on both vertex and edge dimensions as VE-Agg. According to Definition 5.4, we also refer to $A^{'}_{v}$ and $A^{'}_{e}$ as the vertex group-by dimensions(denoted as VD) and the edge group-by dimensions (denoted as ED) respectively. With regard to our running example in Fig. 5.1, Fig. 5.3 (a) and (b) illustrate two examples of the aggregate graph based on < {Gender,*,*}, {*,Type,*}> (for simplicity, we omit * in the rest of this chapter) and <Nation, Date> respectively. In these examples, COUNT(.) is the measure for both vertex and edge dimensions. In a graph OLAP, these measures can also be Average Degree, Diameter, Min/Max Degree, Max/Min Centrality, the Most Central Vertex, Containment and so on, besides the traditional measures like SUM, AVG etc. Note that the measures for vertex and edge can be different.

Figure 5.3 (a) provides a *high level aggregate graph* which corresponds to the answer to the example query in category 3. There are two condensed vertices including male and female in the aggregate graph. The vertices for male and female are weighted as 5 and 4,

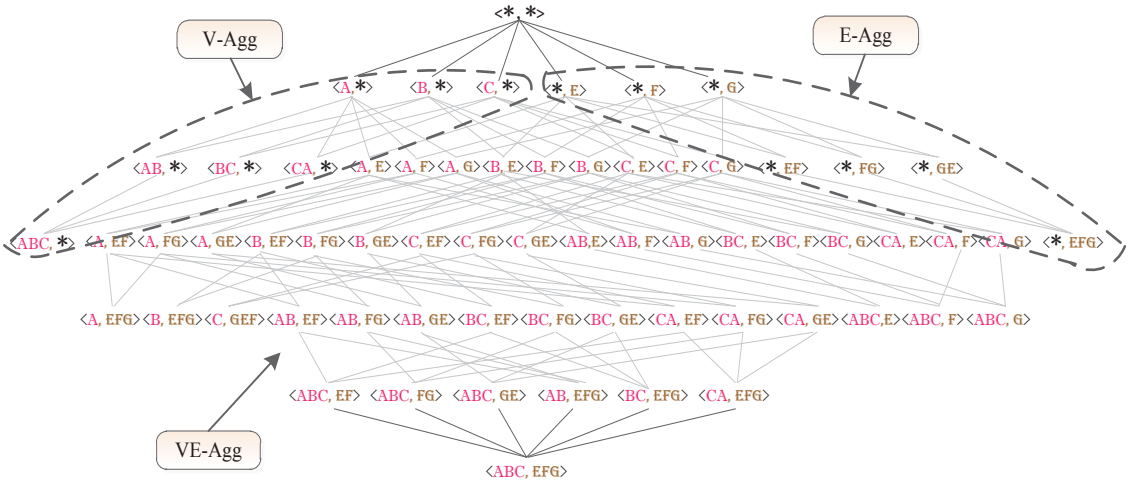


Figure 5.4: The Hyper Graph Cube lattice

since there are 5 males and 4 females in the original graph. The weighted edges indicate the number of relationships in the original graph. For instance, "Family:3" between male and female indicates that there are 3 family edges where one vertex is male and another vertex is female. Fig. 5.3 (b) provides another aggregate graph which can be used to answer the query mentioned in category 2. The weighted edges between USA and SG show the trend of the number of relationships between USA and SG from 2010 to 2012.

Now we provide the formal definition of the Hyper Graph Cube.

Definition 5.5. ***Hyper Graph Cube:*** *Given an attributed graph G=(V, E, A_v, A_e) with n V-Dims and m E-Dims, Hyper Graph Cube constructs 2^{n+m} cuboids to aggregate the graph based on all possible V-Dims and E-Dims. It consists of three different types of aggregations: V-Agg represented as < VD, *>, E-Agg represented as <*,ED > and VE-Agg represented as <VD, ED>. There are 2^n-1 V-Agg cuboids and 2^m-1 E-Agg cuboids which are obtained by performing aggregation along either V-Dims or E-Dims. There are 2^{n+m} -2^n-2^m+1 VE-Agg cuboids which are obtained by aggregating the graph based on both V-Dims and E-Dims, each of which is an aggregate graph defined in Definition 5.4. Note that we reserve the cuboid <*,*> as a special cuboid.*

Intuitively, all the cuboids in Hyper Graph Cube can be represented as the Cartesian product of the V-Agg lattice and the E-Agg lattice. For instance, the hyper cube lattice of the example graph in Fig. 5.1 includes 64 cuboids represented as the Cartesian product between V-Agg and E-Agg as shown in Fig. 5.2. Fig. 5.4 shows the expanded lattice from Fig. 5.2. Considering two cuboids C_1=$<$ VD$_1$, ED$_1$ $>$ and C_2=$<$ VD$_2$, ED$_2$ $>$, C_1 is an ***ancestor*** of C_2 (denoted as $C_1 \prec C_2$), if VD$_1 \subseteq$ VD$_2 \bigwedge$ ED$_1 \subseteq$ ED$_2$. Meanwhile, C_2 is the ***descendant*** of C_1 (denoted as $C_2 \succ C_1$). For instance, <Gender, Type> $\prec$ <{Gender, Nation}, Type>.

Query Support: Based on the Hyper Graph Cube, queries can be easily supported by using their corresponding cuboids. For category 1, the queries can be directly answered by the cuboids in V-Agg or E-Agg. For instance, the query "How many relationships appeared in 2012?" can be answered by the cuboid <*, Date>. For category 2, the queries can be answered by the cuboids in VE-Agg. Furthermore, the cuboid <Nation, Date> in Fig. 5.3 (b) can be used to answer the query mentioned in category 2. In addition, each complete aggregate graph in VE-Agg captures concisely the answer of the request in category 3. For instance, Fig. 5.3 (a) provides the high level aggregate network on <Gender, Type> which is much easier to observe and understand than the massive original graph, where this relationship is hidden.

Roll-Up/Drill-Down OLAP Operations: Roll-Up/Drill-Down are two of the most important OLAP operations to generate views in different levels and granularities. In graph OLAP, each V-Dim or E-Dim may be a dimension associated with a conceptual hierarchy where the Roll-Up/Drill-Down operation can be performed as well. For instance, the dimensions Birth Place and Time may be associated with a geographic or time hierarchies respectively as follows:

- Birth Place: City $\rightarrow$ State $\rightarrow$ Country$\rightarrow$ all
- Time: Month $\rightarrow$ Year $\rightarrow$ Decade $\rightarrow$ all

The Roll-Up/Drill-Down operations along V-Agg or E-Agg are quite similar to the traditional OLAP in the literature [16]. We will not discuss them here. We mainly focus on OLAP operations on the VE-Agg cuboids. Due to the unique feature of VE-Agg, we introduce four different types of Roll-Up/Drill-Down operations in graph OLAP as follows:

Vertex-Up: For Vertex-Up, we fix the edge dimension while we roll up along the vertex dimension to aggregate the graph into a more summarized level, such as navigating the aggregate graph from <City, Year> to <State, Year>.

Edge-Up: For Edge-Up, we fix the vertex dimension while we roll up along the edge dimension to aggregate the graph into a more summarized level, such as navigating the aggregate graph from <City, Year> to <City, Decade>.

Vertex-Up-Edge-Up: For Vertex-Up-Edge-Up, we roll up along both the vertex and edge dimensions to aggregate the graph into a more summarized level, such as navigating the aggregate graph from <City, Year> to <State, Decade>.

Vertex-Up-Edge-Down: For Vertex-Up-Edge-Down, we roll up along the vertex dimension and drill down along the edge dimension to aggregate the graph, such as navigating the aggregate graph from <City, Year> to <State, Month>.

Similarly, we have four corresponding operators - Vertex-Down, Edge-Down, Vertex-Down-Edge-Down and Vertex-Down-Edge-Up - that operate in the opposite direction. We note that we can speed up OLAP operations for distributive and algebraic measures in two ways. First, the Roll-Up/Drill-Down can be conducted based on the intermediate aggregate graph instead of the base graph. Second, the Roll-Up/Drill-Down operations based on the closest aggregate graph is more efficient than others. For instance, to get an aggregate graph based on <Country, Decade>, Roll-Up operation is more efficiently conducted using the aggregate graph based on <State, Year> than <City, Month>.

5.3 A Naive MR-based Scheme

We first briefly introduce the computation paradigm of the MR framework. The computation of MR follows a fixed model with a map phase, followed by a reduce phase. The map function running on a mapper is used to process (key, value) pairs (k1,v1) of one input data chunk read from the distributed file system (DFS). After applying the map function, it then emits a new set of intermediate (k2,v2) pairs. The MR library sorts and partitions all the intermediate (k,v) pairs based on k. In the reduce phase, the library merge-sorts all the (k,v) pairs and supplies the globally sorted data to the reduce function running on a reducer. After the reduce process, the reducer emits new (k3,v3) pairs to DFS.

Algorithm 5.1: The Naive Cubing MR Job

```
1  Function Map()
2    # t is a tuple in the data
3    If t is a vertex (or an edge or a joined edge) then
4      foreach cuboid C_i ∈ V-Agg (or E-Agg or VE-Agg)
5        Project C_i's group-by attributes from t ⇒ k
6        Other information ⇒ v;
7        emit(k,v);
8  Function Reduce( k, v_0, v_1, ..., v_k)
9    let M be the measure function
10   emit(k, M(v_0, v_1, ..., v_k));
```

Under the MR computation model, a naive algorithm may conduct the graph cubing in two steps. Recall that the vertex and edge attributes are typically stored in two separate tables in the original data format as shown in Fig. 5.1. Therefore, in the first step, we join the two tables to obtain a flat table that contains all the dimensions. This join operation can be performed using one MR job (referred as a *Join-Job*), and the joined output is written back to DFS.

The second step conducts the cube computation. One approach is to compute all the cuboids in one MR job, referred as a *Cubing* job, where each cuboid is processed

independently. Obviously, calculating the cuboids in V-Agg and E-Agg based on the original vertex and edge attribute tables is more suitable than based on the joined data. Thus, this cubing job can take both original tables and joined data as input. Recall that each cuboid in the VE-Agg is an aggregate graph as defined in Definition 5.4. To construct such an aggregate graph, each condensed vertex actually can be calculated while processing the cuboids in V-Agg since they would group the vertices in the same way. Thus, we only need to calculate the weighted edges between two condensed vertices in the aggregate graph based on the joined data. The algorithm is provided in Alg. 5.1. This naive algorithm extracts, shuffles and processes each cuboid's group-by attributes independently.

However, for large attributed graph, such a naive algorithm is expected to perform poorly for two reasons:

Size of joined data: As one vertex may be associated with multiple edges, the join output size in the Join-Job can be very large. Assume that the average degree of each vertex, the size of vertex attribute table and the size of the edge attribute table are d, $|V|$ and $|E|$ respectively. The size of the output join data from the Join-Job is almost $d * |V| + |E|$.

The large amount of intermediate data: Processing each cuboid independently may also generate a large amount of intermediate data, since each cuboid needs to extract its own (k,v) pairs which will incur high overhead.

5.4 MR-based Hyper Graph Cube Computation

In this section, we introduce a scalable MR-based Hyper Graph Cube computation approach, MRGraph-Cubing to handle the large attributed graphs. We first introduce the overall process of MRGraph-Cubing: 1) It joins the vertex and attribute tables. For this,

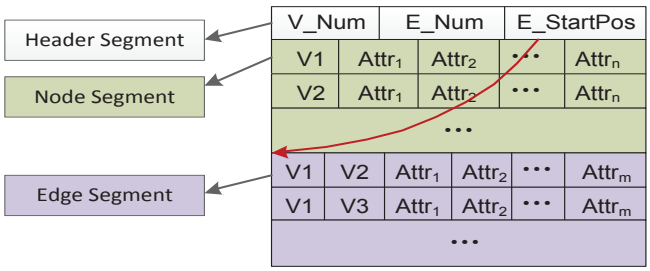


Figure 5.5: The self-contained file format

we propose a **self-contained join** to avoid writing and reloading the joined data to/from DFS (section 5.4.1); 2) It groups cuboids into batches so that the intermediate data and computation can be shared. Our **cuboids batching** scheme identifies the cuboids that can be batched (section 5.4.2); 3) It further bundles batches into bags so that we can process each bag in a single MR job (section 5.4.3); 4) To ensure optimal bundling of batches into bags in (3), we further develop a **cost-based execution plan optimizer** that can generate an execution plan to minimize the cube computation time (section 5.4.4).

5.4.1 Self-Contained Join

To reduce the high overhead incurred by the large size of joined data in the first Join-Job, we propose a *self-contained join* technique to postpone the join operation to the map side of the second cubing MR job. In so doing, after the join, the data will be directly used for graph cube computation and do not need to be written/reloaded via DFS. However, a self-contained join requires the data in each mapper to contain the edges and its corresponding vertex information. Thus, instead of running a Join-Job, we first issue a *Blk-Gen* MR job to reorganize the original data into a series of self-contained data files.

In particular, the Blk-Gen job reads both the vertex and edge attribute tables in the map phase. Then it partitions the edges to different reducers according to its two vertex IDs. Meanwhile, it also partitions the vertex information to the corresponding reducers

whose edges contain the same vertex IDs. Note that each vertex is shuffled to multiple reducers as needed. In the reduce phase, the Blk-Gen job generates a series of self-contained data blocks and outputs each block as one file whose format is described in Fig. 5.5. Under this scheme, in each file, the vertex can be shared by multiple edges instead of being replicated multiple times.

Since MR does not further partition the input file if the file is not bigger than one block, each self-contained file will be supplied to one mapper directly to perform a join in the second cubing join.

5.4.2 Cuboids Batching

To build a graph cube, computing each cuboid independently is clearly inefficient. A more efficient solution, which we advocate, is to combine cuboids into batches so that intermediate data and computation can be shared and salvaged.

For the scheme to be effective, we first identify the cuboids that can be combined and batched together. In this work, we assume that we are materializing the complete cube. Our scheme can be easily generalized to materialize a partial cube (compute only selected cuboids).

Recall that there are three different types of cuboids: V-Agg, E-Agg and VE-Agg. For processing the cuboids in either V-Agg or E-Agg, we can adopt the algorithm which is proposed in chapter 4. As such, we shall give an overview here as our focus in this work is on the VE-Agg. The collection V-Agg/E-Agg cuboids can be batched together under the MR framework if they satisfy the following *combine criterion*:

Criterion 5.1. Among the multiple V-Agg or E-Agg cuboids, any two of them have the ancestor/descendant relationship and share the same prefix.

For instance, given three V-Agg C_1, C_2 and C_3, if the vertex group-by dimensions of them are VD_1=A, VD_2=AB and VD_3=ABC respectively, then $C_1 \prec C_2 \prec C_3$ and they

share the same prefix with each other. In this given example, these three cuboids can be combined and processed together to facilitate the MR sorting feature (the framework sorts all the intermediate (k,v) pairs according to the key) using the approach we proposed as follows: When the mapper reads each tuple, it emits one (k,v) pair to serve C_1,C_2 and C_3 where VD_3 is set as the key and VD_1 is used to partition the intermediate pairs, instead of the emitting three pairs. This would guarantee that the tuples with the same group-by values of C_1, C_2 and C_3 are shuffled to the same reducer, and thus can be processed together. More correctness proof and details can be found in chapter 4.

The benefits of this approach are: 1) In the reduce phase, the group-by dimensions are all in sorted order for every cuboid in the batch, since the framework would sort the data before supplying to the reduce function. This is an efficient way of cube computation since it obtains sorting for "free" and no other extra sorting is needed before aggregation. 2) All the ancestor cuboids do not need to shuffle their own intermediate data but use their descendant's. This would significantly reduce the intermediate data size, and thus remove a lot of data sorting/partitioning/shuffling overheads.

Now, combining cuboids in VE-Agg is much more challenging. This is because each cuboid in VE-Agg is an aggregate graph. The aggregation is performed from both the V-Dims and E-Dims. We claim that the cuboids in VE-Agg can be batched together under the MR framework if they satisfy the following *combine criterion*:

Criterion 5.2. Among the multiple VE-Agg cuboids, any two of them have the ancestor/descendant relationship. In addition, the V-Dims between any two of the cuboids share the same prefix, as well as their E-Dims.

Intuitively, to obtain the aggregate graph of each VE-Agg cuboid, the procedure can be divided into two parts: 1) In part 1, the vertices with the same V-Dims are grouped into condensed vertices and each condensed vertex's weight is computed using the vertex aggregate measure. 2) In part 2, the edges with the same V-Dims of the two vertices and

the same E-Dims are grouped into condensed edges weighted using the edge aggregate measure. As we have mentioned before, the first part can be conducted with the V-Agg cuboids since they share the same vertex grouping condition. Note that if the measure of V-Agg is the same as the vertex aggregate measure in the VE-Agg, the result of V-Agg can be directly used for constructing the VE-Agg. Thus, we focus on introducing the approach of calculating the weighted edges here.

As an example, suppose we are given three VE-Agg cuboids C_1, C_2 and C_3, where their vertex group-by dimensions are VD_1=A, VD_2=AB and VD_3=ABC and the edge group-by dimensions are ED_1=E, ED_2=EF and ED_3=EF respectively, then $C_1 \prec C_2 \prec C_3$ and the V-Dims (as well as the E-Dims) share the same prefix order. In this example, C_1 and C_2 can be processed and combined together with C_3 into one batch. Note that calculating the weighted edges is based on the joined edges. Each joined edge e is a triple-tuple: V-Dims of sV (VD(sV)),the V-Dims of tV (VD(tV)) and E-Dims of e (ED(e)).

Under the MR framework, C_1, C_2 and C_3 can be processed in one batch as follows: When one mapper parses one joined edge e, it emits one (k,v) pair to serve C_1, C_2 and C_3 where the concatenation of VD_3(sV), VD_3(tV) and ED_3(e) (which is ABCABCEF) as the key. And the concatenation of VD_1(sV), VD_1(tV) and ED_1(e) (which is AAE) is used to partition the (k,v) pairs. This guarantees that all the edges with the same group-by values of C_1, C_2 and C_3 are shuffled to the same reducer, thus they can be processed together.

Now, we formally define two special cuboids within one batch.

Definition 5.6. ***Descendant_Cuboid****: Given one batch, if all other cuboids are the ancestor cuboids of cuboid A, A is defined as the Descendant_Cuboid, denoted as Des_Cubd in one batch. During the cube computation, the aggregation dimensions of A can be projected as the key to serve all other cuboids within the batch under the MR framework.*

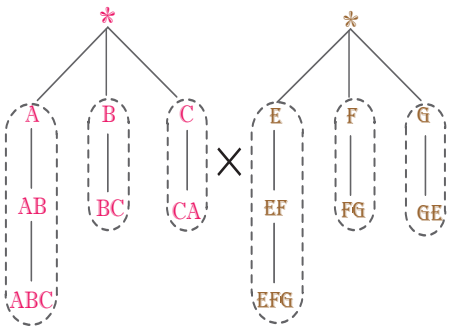


Figure 5.6: The generated batches

Definition 5.7. ***Partition_Cuboid****: Given one batch, if cuboid A is the ancestor cuboid of all others, A is defined as the Partition_Cuboid, denoted as Par_Cubd in one batch. During the cube computation, the aggregation dimensions of the A are the ones used to partition intermediate (k,v) pairs under the MR framework.*

For instance, given one batch with three V-Agg cuboids {<A,*>, <AB,*>, <ABC,*> }, <ABC,*> and <A,*> are called the Des_Cubd and Par_Cubd respectively. Likewise, given one batch with three VE-Agg cuboids {<A,E>, <AB,EF>, <ABC,EF>}, <ABC,EF> and <A,E> are called the Des_Cubd and Par_Cubd respectively.

Basically, the more cuboids we combine into one batch, the less intermediate data will be generated and the more computation sharing we can get. Based on the aforementioned principles, one cuboid can be batched with all its descendant cuboids satisfying criterion 5.1 or criterion 5.2. Therefore, to generate batches, we first search the cuboids in V-Agg and E-Agg and generate the batches within V-Agg and E-Agg according to criterion 5.1. For instance, given the lattice in Fig. 5.2, it generates three batches within V-Agg and E-Agg as shown in Fig. 5.6 using the dotted lines.

For VE-Agg, the batch plan can be generated by combining the V-Agg batches with the E-Agg batches. Intuitively, a cartesian product is conducted between the V-Agg batches and E-Agg batches. Let us still take Fig. 5.2 as an example. The final execution plan will generate 9 batches whose Des_Cubds are <ABC,EFG>, <ABC,FG>, <ABC,

GE>, <BC,EFG>, <BC,FG>, <BC, GE>, <CA,EFG>, <CA,FG> and <CA, GE>. Each batch consists of multiple cuboids. For instance, the batch <ABC, EFG> includes the 9 cuboids <A,E>, <A,EF>, <A,EFG>, <AB,E>, <AB,EF>, <AB,EFG>, <ABC,E>, <ABC, EF> and <ABC,EFG>.

5.4.3 Batch Processing

Since processing V-Agg, E-Agg and VE-Agg are all based on different input data, we propose to process the batches in these three different types separately. The MR-based algorithm of processing V-Agg and E-Agg is similar to the cube computation in relational database like what has been proposed in chapter 4. Thus, we omit it here. As mentioned in section 5.4.2, calculating the condensed vertices can be integrated with V-Agg. Therefore, in this section, we mainly focus on how the batches in VE-Agg can be processed to get all the condensed and weighted edges.

Given a computing cluster and a set of batches, there are multiple plans to process all the batches.

Definition 5.8. ***Execution Plan:*** *Given a set of batches B=*$\{B_1, B_2, ..., B_x\}$ *, the execution plan is a set of MR jobs p=*$\{j_0, j_1, ..., j_k\}$ *where* j_i *is in charge of processing one bag. Each bag consists of one or more batches. The batches processed by* j_i *is denoted as* $bag(j_i)$*. And* p *satisfies the following condition:* $bag(j_0) \bigcap bag(j_1) \bigcap .... \bigcap bag(j_k) = \phi$ *and* $bag(j_0) \bigcup bag(j_1) \bigcup .... \bigcup bag(j_k) = B$.

For instance, two straightforward execution plans are: 1) put each batch into one independent bag and process each bag using one MR job; 2) put all the batches into one bag and process this bag using one MR job. The advantage of the second plan is that the original data only need to be read once. However, each mapper has to replicate and emit all the intermediate data for all batches which incur high overhead to collect, partition

and sort them. In contrast, under the first plan, the MR job only needs to emit the data for one batch resulting in more efficient data collection, partitioning and sorting in the map phase. However, each job needs to read the original data once, which can be very costly when the number of jobs is large.

Clearly, there are many other possible plans. We defer the discussion on finding an optimal execution plan to the next subsection. Here, we shall focus on how a bag (containing a set of batches) can be processed using a single MR job.

Consider one bag B consisting of x VE-Agg batches $\{B_1, B_2, ..., B_x\}$. Suppose the number of reducers needed for each batch is R=$\{R_1, ..., R_x\}$, where R_i is the number of reducers assigned for batch B_i. Alg. 5.2 lists how our proposed MR-based scheme works.

Map Phase: The input data are the self-contained files output from the first Blk-Gen job and each file is supplied for one mapper. In the map phase, the mapper first conducts a join. Basically, it caches the vertex information in memory, as the vertices are supplied to mappers earlier than the edges (lines 3-4). Note that caching the vertex information consumes very little memory, since cached information is always smaller than one block (input file) size. When an edge arrives, it performs a join with the vertex data. Whenever a joined tuple is produced, it constructs and emits x (k, v) pairs for the x batches in bag B (lines 5-8). For each batch B_i, the key is set according to its Des_Cubd as described in Section 5.4.2. Assume the Des_Cubd in B_i is <VD, ED>, for each joined tuple e, VD(sV), VD(tV) and ED(e) are extracted and concatenated as the key. Meanwhile, other measure information can be put to the value. For instance, if the aggregate function is COUNT(), then 1 can be put into the value.

In order to distinguish which (k,v) pair is for which batch, we append one bitmap to the value. The size of the bitmap corresponds to the number of batches where the kth bit is set to 1 if this pair belongs to batch B_{k+1}.

Algorithm 5.2: The MRGraph-Cubing Algorithm

```
1  Function Map()
2    Let t be an input tuple
3    If t is a vertex then
4      Cache t in memory;
5    ElseIf t is an edge then
6      t joins vertex ⇒ e
7      foreach B_i ∈ B
8        extracts k and v from e for B_i and emit(k,v) ;
9  Function Partition(k,v)
10     b ⇐ get batch number from v;
11     Par_Cubd ⇐ get partition value from k;
12     return Σ_{i=1}^{b-1} R(i) + Par_Cubd%R_b;
13 Function Reduce(k, v_0, v_1, ..., v_k)
14   # M ⇐ The measure function
15   B_i ⇐ Get batch from the batch identifier
16   C_i ⇐ The cuboid in B_i
17   For C_i in B_i
18     If the group-by cell in C_i receives all tuples it needs
19       v ⇐ M(v_1, ..., v_m, v'_1, ..., v'_k, ...)
20       emit(k,v);
21     Else
22       buffer the measure for aggregation
23 Function Combine(k, v_0, v_1, ..., v_k)
24   # M ⇐ The measure function
25   id ⇐ Get batch from identifier
26   v ⇐ M(v_1, ..., v_m, v'_1, ..., v'_k, ...)
27   v ⇐ v.append(id);
28   emit(k,v);
```

The partitioning function partitions the intermediate (k,v) pairs to their corresponding reducers according to the Par_Cubd and reducer allocation plan R (lines 9-12).

Reduce Phase: In the reduce phase, all the (k,v) pairs are sorted and grouped based on the key together. Each reducer obtains its computation tasks (the cuboids in the batch) by parsing the identifier in the value. For each input tuple, the reduce function extracts the measure and projects the group-by dimensions for each cuboid in the batch. For the Des_Cubd, the aggregation can be conducted based on each input tuple, since each input tuple is one complete group-by cell. This case is captured in line 18. For other cuboids,

the measures of the group-by cell are buffered until the cell receives all the measures it needs for aggregation (lines 13-22). The aggregation results for different cuboids are written into different destinations in DFS.

Note that if the (k,v) pairs can be pre-aggregated in the map phase, users can write a *combiner* which will reduce the shuffle data size. The combine function is listed in lines 23-28.

Using the Alg. 5.2, multiple batches in one bag can be processed in one MR job. In next section, we introduce our proposed batch execution optimization technique to generate the optimal execution bags.

5.4.4 Cost-based Execution Plan Optimization

Definition 5.9. ***Batch Execution Plan Optimization****: Given a computing cluster and x batches, the batch execution plan optimization problem is to find the best way to bag the batches such that the generated execution plan (as defined in Definition 5.8) has the smallest cube materialization time.*

Intuitively, the optimizer consists of two key components: a) *Plan refinement* searches different execution plans; b) *Plan execution time estimation* estimates each plan's execution time so that the plan with the smallest execution time can be chosen.

Plan Refinement: Given x batches, $B : \{B_1, B_2, ..., B_x\}$, enumerating all the possible plans is equivalent to computing all partitions from B which has been well studied in [21]. The total number of partitions is $\Theta((\frac{x}{ln(x)})^x)$ [21]. Therefore, when x is large, an exhaustive enumeration is no longer applicable. As such, some heuristic algorithms can be used to find a suboptimal execution plan.

In this work, we adopt a greedy algorithm that is iterative in nature and follows the classical local search pattern. Let P_i denote the input execution plan for iteration i. Initially, P_0 corresponds to the case where there are x jobs (i.e., each bag has only one

Table 5.1: Variables Used in Cost Model

Notations	Description
SF	Sort factor configured in MR cluster
S_{spill}	Spill size configured in MR cluster
IO_{lr}	I/O cost for reading from local disk per MB
IO_{lw}	I/O cost for writing from local disk per MB
NW	cost for network transfer per MB
IO_{hr}	I/O cost for reading from DFS per MB
IO_{hw}	I/O cost for writing to DFS per MB

batch). Intuitively, the algorithm works as follows: In (i+1)th iteration, it evaluates all the plans obtained by bundling any two jobs in P_i together and finds the best plan as P_{i+1}. If P_{i+1} is better than P_i (based on a cost model to be discussed shortly), P_{i+1} is passed on to the next search iteration. If P_{i+1} is worse than P_i or P_i contains one job with all batches, the algorithm terminates. And P_i is chosen as the final execution plan. In the worst case, this algorithm needs to enumerate $\binom{x}{2}$+$\binom{x-1}{2}$...+$\binom{2}{2}$ possible plans, which is bounded in $O(x^3)$ time, where $\binom{i}{2}$ is the number of plans to evaluate in the $(x-i+1)^{th}$ iteration. In practice, this plan enumeration time is acceptable during the cube computation.

Plan Execution Time Estimation: Before introducing how to evaluate each plan's execution time, we first provide a cost model which is used to estimate the execution time of each mapper and reducer in one job. The cost model is aware of the cluster hardware, MR configurations, input file and the batches it needs to process. For simplicity, we assume no compression is adopted during the processing. In addition, considering that the I/O cost dominates performance, for simplicity, we omit the CPU cost in the model. Table 5.1 lists the variables related to the MR cluster.

Assume one MR job needs to process one bag with y batches $\{B_1, B_2, ..., B_y\}$ and m input files. The set of reducers to process B_i is R_i. In addition, We refer to S_i as the input file size for mapper M_i, C_{ij} as the jth cuboid in batch B_i, CR_i as the combine ratio of batch B_i in the map phase, CR_{ij} as the combine ratio for C_{ij} in the reduce phase, P_i as the project ratio for B_i and P_{ij} as the project ratio for C_{ij}. Here, the combine (resp.

project) ratio indicates the percentage of data that remains after combine/aggregate (resp. project) operations. If no combiner is used, then the batch combine ratio CR_i equals to 1.

For the map phase, we refer to $M_c = \sum_{i=1}^{i=y}(S_i \cdot P_i)$ as the size of (k,v) pairs for all batches after projection, $N_s = \lceil \frac{M_c}{S_{spill}} \rceil$ as the number of spills created in spill phase, $N_m = \lceil log_{SF} N_s \rceil$ as the number of merge passes, $M_o = \sum_{i=1}^{i=y}(S_i \cdot P_i \cdot CR_i)$ as the intermediate map output data after combining.

The total cost of mapper M_i is calculated as follows:

$$S_i \cdot IO_{hr} + M_o \cdot N_m(IO_{lr} + IO_{lw}) \tag{5.1}$$

where $S_i \cdot IO_{hr}$ is the cost for reading the input file from DFS; $M_o \cdot N_m(IO_{lr} + IO_{lw})$ is the local I/O cost for sorting and partitioning the intermediate data.

For the reduce phase, we refer to R_{ij} as the jth reducer processing B_i, $R_{np} = \lceil log_{SF} m \rceil$ as the number of merge passes on R_{ij}, R_{in}=$\frac{\sum_{k=1}^{k=m}(S_k \cdot P_i \cdot CR_i)}{|R_i|}$ as the input size for R_{ij} and S_f= $\sum_{C_{ij} \in B_i}(R_{in} \cdot CR_{ij} \cdot P_{ij})$ as the final output view size in R_{ij}.

The total cost of the reducer R_{ij} is calculated as follows:

$$R_{in} \cdot NW + R_{in} \cdot R_{np} \cdot (IO_{lr} + IO_{lw}) + S_f \cdot IO_{hw} \tag{5.2}$$

where $R_{in} \cdot NW$ is the cost for shuffling data from mappers to R_{ij}; $R_{in} \cdot R_{np} \cdot (IO_{lr} + IO_{lw})$ is the cost for the merge sort, and $S_f \cdot IO_{hw}$ is the cost for writing the aggregate results to DFS.

Given a cluster, all the variables in Table I can be predetermined. Meanwhile, each input file size for each mapper can be easily obtained from DFS. The only parameters that we need to dynamically collect w.r.t. different datasets for the model are the combine ratio and project ratio for each batch and cuboid. To collect this information, a “mini”

cubing is performed based on a small set of sampling data in memory in each reducer during the Blk-Gen job. The project ratio is related to the dimension size, thus it is easy to obtain. The batch combine ratio CR_i (resp. cuboid combine ratio CR_{ij}) can be obtained by recording the percentage of data in B_i (resp. C_{ij}) that remains after applying the combine (resp. aggregate) function. Since the data is already in memory, this graph information collection is, as we shall see in our experimental results in section 5.5, efficient. The average value among these reducers can be input to the model for planning. We note that this cost model may not provide an accurate estimation. Instead, this is an approximate approach to evaluate the relative cost. The intuition is to avoid bad plans. We emphasize that our scheme is not restricted to this specific model; instead, other cost models can also be used.

Given one execution plan p=$\{j_0, j_1, ..., j_k\}$, assume that the execution time of each mapper and reducer has been obtained by the cost model for each job j_i. Estimating the execution time of multiple jobs is still non-trivial under the MR framework. A very straightforward approach is to estimate p's execution time T(p) simply as the sum of each job j_i's execution time T(j_i) as follows: $T(p) = \sum_{i=1}^{k} T(j_i)$.

However, we claim that this naive approach is not accurate, since it omits the context of cluster resource and MR scheduling strategy. For instance, assume the number of mappers and reducers one cluster can run simultaneously is larger than the total number of mappers and reducers that j_1 and j_2 need, the execution time of running j_1 and j_2 will be MAX$(T(j_1), T(j_2))$ instead of $T(j_1) + T(j_2)$.

In this work, we propose a worker fitting model to precisely estimate multiple jobs execution time which tightly simulates the scheduling mechanism in the MR framework. We choose the FIFO scheduler as our illustration example in this section. According to the MR framework, when multiple MR jobs are submitted, it maintains a map task and a reduce task queues separately. The map and reduce tasks of the first received job are

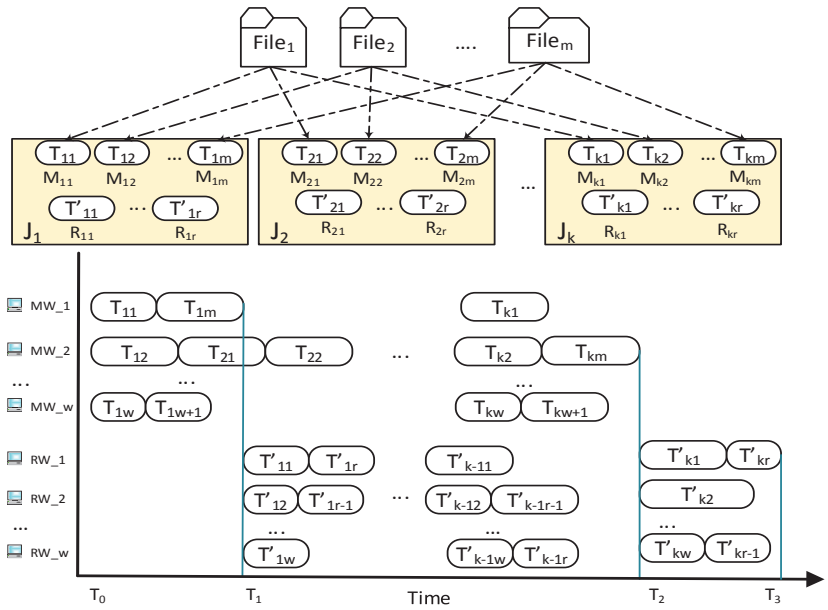


Figure 5.7: The worker fitting model for multiple jobs execution

maintained in the head of the queues. When the cluster has any free slot (mappers or reducers), it schedules the next unprocessed task in the corresponding queue for processing.

Based on this fact, the work fitting model simulates the MR scheduling process as follows: We illustrate it through an example as shown in Fig. 5.7. Intuitively, the mappers and reducers that the cluster supports are considered as map workers and reduce workers which are used to consume the map and reduce tasks. As in Fig. 5.7, there are w map workers and w reduce workers. For a given plan p, k MR jobs are submitted to the cluster.

In Fig. 5.7, the example contains m mappers (as there are m input files) and r reducers. Using the cost model, the optimizer estimates the execution time of each mapper and reducer. For instance, in job j_1, T_{11} is the time needed to process the first input file and T'_{11} is the time needed to process a reduce task. The model captures the slowest reducer finishing time as the plan's execution time.

With a FIFO scheduler, the map tasks in j_1 are assigned to the map workers first,

then the ones in j_2, j_3 and so on. The strategy of task fitting is to assign the map task to the worker with the "cheapest" map tasks. Here, by "cheapest", it means a smaller SUM value of each map's execution time one worker gets. For instance, as in Fig. 5.7, T_{1w+1} is assigned to MW_w since MW_w gets the smallest task from the first round.

On the other hand, the reduce tasks can only be scheduled to shuffle data when some of the mappers in the same MR job have finished. For simplicity, we assume the reduce task in one job starts when all map tasks finish. The optimizer starts to fit the reduce task to the reduce workers using the same task fitting strategy at the time point when the slowest map task finishes and there are "free" reduce workers. Here, free reduce workers means they have finished processing the last assigned reduce task. Otherwise, the reduce task has to wait to be scheduled. For instance, in Fig. 5.7, the slowest map task in j_1 finishes at time point T_1. Then the reduce tasks in j_1 start to be fitted to the reduce workers. At time point T_2, all the map tasks in j_k are finished, and then the reduce tasks in j_k can be scheduled. At time point T_3, all the jobs finish.

We emphasize that this technique is applicable to different schedulers, such as the fair scheduler or the capacity scheduler. With different schedulers, the task queue may be in a different order but still under the same methodology. With this cost-based multiple jobs execution estimation, the optimizer is able to identify the best plan to conduct the graph cube computation.

5.5 Experiment

We conduct the experimental evaluation on our local cluster with 128 nodes. Each node consists of a X3430 4(4) @ 2.4GHZ CPU running Centos 5.4 with 8GB memory and 2X 500GB SATA disks. Our evaluation is based on Hadoop [2], an open source equivalent implementation of MR. The detail configuration of the cluster is provided in

Table 5.2: Cluster configuration

Parameter	Value
Hadoop Version	Hadoop-1.1.1
Mappers per Node	2
Reducers per Node	2
Replication Factor	3
io.sort.fact	20
JVM Size per Task	2GB
Size of Data Chunk	256MB
Default Node Number	64

Table 5.2.

We perform our experimental studies on two kinds of datasets including one real Facebook dataset and a set of synthetic datasets.

Facebook Dataset. This dataset contains a sampled Facebook data collected in April-May 2009 [40]. In the dataset, each vertex is one Facebook user and each edge indicates the relationship between two users. We further extract four dimensions for each vertex: *TotalFriends*, *School*, *Region* and *Affiliation*. And we extract one dimension for each edge: *Type* with three different values (Schoolmates, Colleagues and Friends). This dataset includes 957,359 vertices and 4.5 million edges.

Synthetic Dataset. For synthetic data, we use SNAP to generate the underlying graph structure without dimensions [3]. Since SNAP is slow in generating large attributed graphs, we develop a MR-based parallel synthetic attributed graph generator to add dimensions to each vertex and edge in parallel to the graph structure generated by SNAP.

5.5.1 Effectiveness

We first show the effectiveness of Hyper Graph Cube as a powerful decision making tool over the Facebook data. We present some interesting findings by issuing OLAP queries.

First, we want to identify the communities of active users in Facebook. Note that

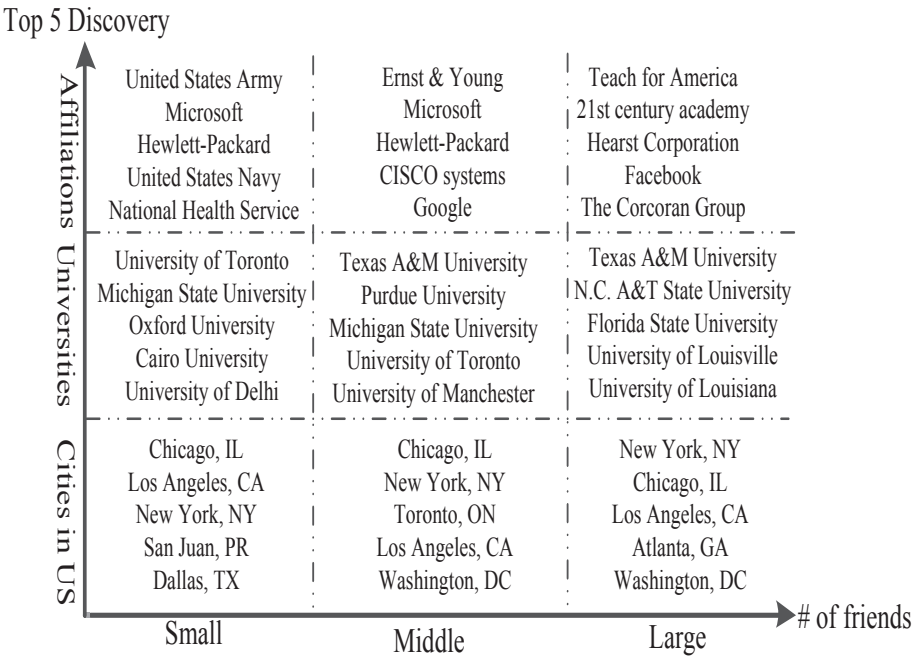


Figure 5.8: OLAP query on V-Agg Cuboids

we classify the number of total friends into three categories: small (the number of friends is less than 150 friends), middle (between 150 to 800) and large (larger than 800). Then based on the V-Agg cuboid $< \{$Region,TotalFriends$\}, * >$, $< \{$School, TotalFriends$\}, * >$ and $< \{$Affiliation, TotalFriends$\}, * >$, we can easily obtain the top 5 communities which has the largest number of people in each category as shown in Fig. 5.8. We can see that "Teach for America", "Texas A&M University" and "New York, NY" have the largest number of people with more than 800 friends on Facebook. The result also indicates that "New York" belongs to the top 5 in each category which shows that people living in New York are very active in Facebook.

Next, we investigate how people are connected with each other between the cities in California: LA (Los Angles), OC (Orange City), SFO (San Francisco) and SV (Silicon Valley). We perform such a query based on the VE-Agg cuboid <Region, Type>. Fig. 5.9 provides an aggregate graph which gives us a much simpler and clearer picture of the original Fackbook data. We observe that there are only 4 cross-city colleague relation-

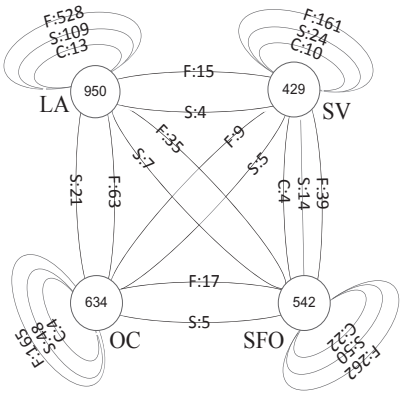
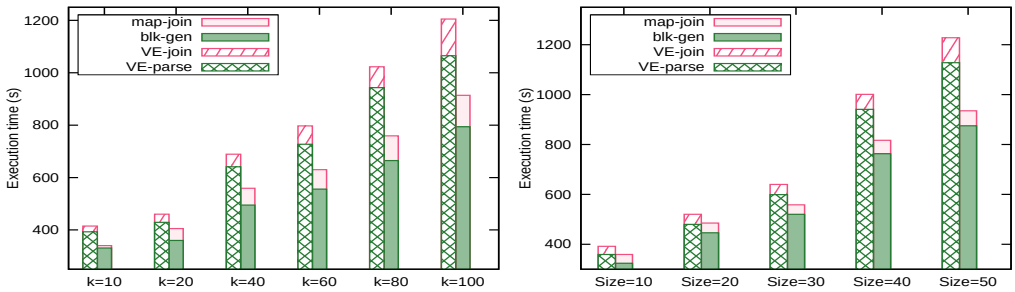


Figure 5.9: OLAP query on VE-Agg cuboid <Region, Type>

ships (denoted as C:4 in the figure) between SV and SFO. Such a relationship suggests that some companies may be operating in these two cities. By drilling down, we discover that three of the relationships were built among the people working in Tellme Inc. which is a big company operating in both cities. Another observation is that cross-city schoolmate relationships (denoted as S:21) appear the most between LA and OC. Our drill-down operation allows us to figure out that these belong to the Universities in California like UCLA. From the results, it is clear that our Hyper Graph Cube model offers an effective mechanism to study and explore the characteristics of large graphs.



(a) Impact of graph degree (b) Impact of V-Dim size

Figure 5.10: Evaluation for self-contained join

For the following experiments, we use the synthetic dataset. Each graph contains 3 V-Dims and 3 E-Dims. By default, the average size of each V-Dim or E-Dim is 7 bytes. Meanwhile, we choose COUNT(·) as the measure function for both the vertex and edge

aggregation. Note that if a graph contains N edges with degree K, there will be N*2/K vertices in the graph.

5.5.2 Self-Contained Join Optimization

In this experiment, we compare the proposed self-contained join optimization with the naive approach. For our proposed self-contained join optimization, we have two MR jobs: *blk-gen* job which reads the vertex and edge attribute tables and generates a series of self-contained files, and *map-join* job which reads the self-contained files, conducts the map side join and ends after the join operation in the map phase. For the naive approach, we also have two MR jobs: *VE-join* job which joins the vertex with edge attribute tables, and *VE-parse* job which reads the joined data and ends when it finishes parsing each joined data in the map phase.

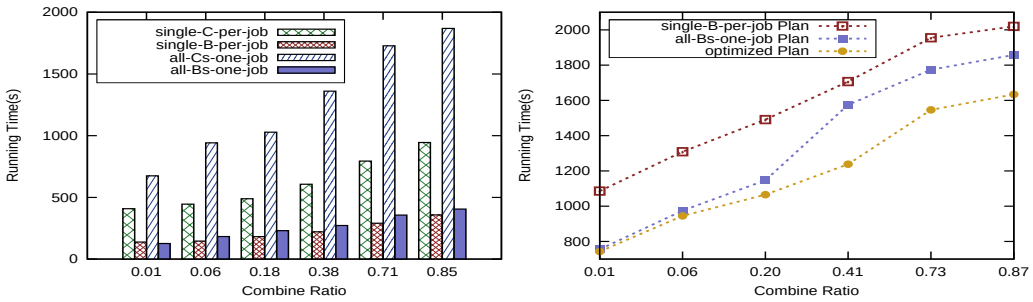
Figure 5.10 (a) shows the performance comparison on datasets with 200 Million(M) vertices when we vary the graph degree from 10 to 100. From the results, we can see that our optimization performs, on average, 30% faster than the naive scheme.

Figure 5.10 (b) provides the comparison results by using the graphs with 200M vertices and with degree 40 when we vary the average size of each V-Dim from 10 bytes to 50 bytes. The results indicate that our scheme has an average performance improvement of 21% over the naive scheme as well.

The insights we gain are: 1) The self-contained join is superior over the naive join strategy over all the evaluated datasets; 2) The performance gain is more significant when the graph degree is high and the vertex attribute is large.

5.5.3 Cuboids Batching Optimization

Next, we study the benefit of our proposed cuboid batching optimization. We implement four algorithms. The first two are the baselines without batching the cuboids:



(a) Impact of batch processing (b) Batch execution plans

Figure 5.11: Evaluation for batch processing

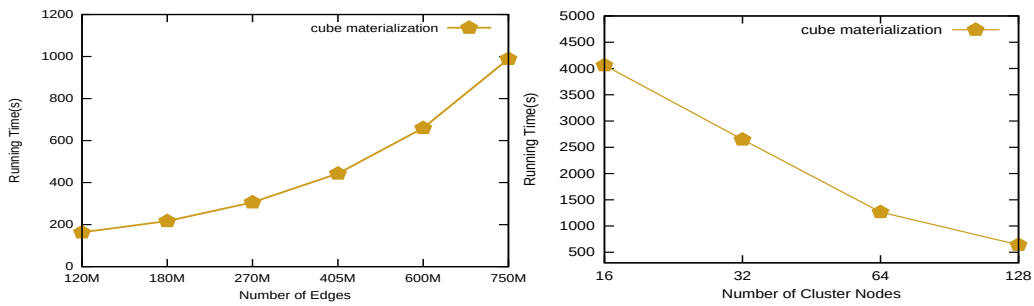
single-C-per-job (computes each cuboid independently in one MR job) and all-Cs-one-job (computes all cuboids in one MR job). The next two are with our batching optimization: 1) single-B-per-job (compute each batch in one MR job) and all-Bs-one-job (compute all batches in one MR job).

Figure 5.11 (a) provides the comparison results based on the datasets with 600M edges and with degree 60 when we vary the average combine ratio of the graph. The results indicate that the algorithms with batching optimization are 2.5X and 4X faster than single-C-per-job and all-Cs-one-job respectively. Meanwhile, we observe that the larger the intermediate data (with a bigger combine ratio) are, the higher the performance gain by the batching optimization.

5.5.4 Batch Execution Plan Optimization

We next evaluate the proposed batch execution plan optimizer. We compare the performance between the plan (*optimized plan*) generated by the optimizer with two basic plans: single-B-per-job and all-Bs-one-job. Fig. 5.11 (b) shows the results on the datasets with 600M edges with degree 60 while varying the combine ratio from 0.01 to 0.87. The execution time is the complete graph materialization time including the first Blk-Gen job. For the optimized plan approach, the graph information is collected by sampling 1% tuples in the memory in the Blk-Gen job.

The results show that the execution time of the optimized plan and all-Bs-one-job is much shorter than single-B-per-job when the combine ratio is low. When the combine ratio equals to 0.01, the optimized plan is the same as the all-Bs-one-job. This is reasonable, since when the map output size is small, combining all batches in one job is less costly. It also shows that the optimizer can generate much better plan than the two basic ones when combine ratio increases. From this experiment, we gather the following insights: 1) The in-memory graph information collection for cost model incurs very low overhead; 2) The optimizer is able to find reasonable execution plan in all cases even with a simple sampling approach.



(a) Impact of graph size (b) Impact of parallelism

Figure 5.12: Evaluation of the plan optimizer and scalability

5.5.5 Scalability

Finally, we evaluate the scalability of our MRGraph-Cubing approach from two different angles. The first set of experiments studies the scalability of the scheme with respect to the size of the datasets. Fig. 5.12 (a) shows the execution time of cube computation on the datasets with degree 60 when we vary the number of edges from 120M to 750M. We observe that when the dataset is 1.5X bigger, the execution time becomes almost 1.5X slower. This indicates the algorithm scales well in terms of dataset size.

The second set of experiments aims at studying the impact of parallelism. Fig. 5.12 (b) provides the execution times of cube materialization on a dataset with 750M vertices

with degree 60 when we vary the cluster size from 16 to 128 nodes. The results indicate that when the computation power is doubled, the execution time almost reduces to half from 16 to 64 nodes. This confirms that the algorithm scales linearly up to 64 nodes. Beyond 64 nodes, the benefit of parallelism decreases a little. This is reasonable since the MR framework setup time may reduce the benefits of increasing the computation resources when the cube computation time is not long enough.

5.6 Summary

In this work, we extended the OLAP techniques to attributed graphs towards a better query and decision making support. To support graph OLAP, we proposed a new conceptual graph cube model-Hyper Graph Cube. It is an effective approach to aggregate the graphs for users to better understand the characteristics of the underlying massive graphs from different granularities and levels. In addition, we also provided, to the best of our knowledge, the first parallel MapReduce-based graph cube computation solution over large-scale attributed graphs with various optimization techniques. Experimental results showed that the optimizations perform at least 2.5X to 4X faster than the naive approach.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Analyzing large-scale data has become one of the main challenges in various enterprises. The size of data in many of today's applications has been dramatically increasing, such as scientific data, financial data and social network data etc. These data resources contain a wealth of information that is of benefit to different communities. A better understanding of them may help us have a better insight of the world, better target marketing campaigns and provide a better decision making support in industries. However, due to the increasing size of data, analyzing these data becomes quite difficult as well.

In this thesis, we have categorized the big data analysis into two types (computation intensive and data intensive) and proposed new scalable, efficient and practical parallel data processing algorithms, frameworks and systems based on the MapReduce computation paradigm. We tackled the problem by studying two types of data analyses: Combinatorial Statistical Analysis (CSA, as an representative example of computation intensive analysis to finding the significant associations that is measured by statistical methods) and OLAP cubes analysis (as an representative example of data intensive anal-

ysis to materialize the data in support of decision making in traditional data warehousing over relational data and graph warehousing over attributed graphs).

6.1 Thesis Contributions

Our first contribution is to introduce a generic MapReduce-based CSA framework: COSAC-COmbinatorial Statistical Analysis on Cloud platforms. In particular, we proposed an efficient and flexible object combination enumeration framework with good load balancing and scalability for large scale of datasets using the MapReduce paradigm. Two schemes are developed in the framework: Exhaustive Testing- enumerating the entire set of objects and Semi-Exhaustive testing- enumerating a subset of objects. Our framework is suited for any application that needs to enumerate the object combinations. We also proposed a technique for efficient statistical analysis using IRBI (Integer Representation and Bitmap Indexing) which is both CPU efficient with regard to statistics testing, and storage and memory efficient. The approach we adopted can be a promising solution to speed up the statistical testing in many other applications where statistics methods have been used, e.g. data mining, machine learning. We further proposed an optimization technique of computation sharing to salvage the computation among the combinations during statistical testing with significant performance savings, instead of conducting the testing for each combination independently. Our experimental results demonstrated that our framework is able to conduct analysis in hours where the task normally takes weeks before, if not months [60]. To the best of our knowledge, none of the existing framework has such a computation capability.

Our second contribution is to introduce a scalable parallel data cube analysis system, `HaCube` on big data, integrating a new data cubing algorithm and an efficient view maintenance scheme for traditional OLAP and data warehousing. `HaCube`, an exten-

sion of MapReduce, modifies the Hadoop MapReduce framework while retaining good features like ease of programming, scalability and fault tolerance. It also has a user-friendly interface layer for effective data cube analysis. We also proposed a new cubing algorithm which is able to incorporate sort feature of MapReduce to batch the cuboids processing to salvage partial work done. In the cubing algorithm, we designed a general and effective load balancing scheme LBCCC (short for Load Balancing via Computation Complexity Comparison) to ensure that resources are well allocated to each batch. We further adopted a new computation paradigm, MMRR(MAP-MERGE-REDUCE-REFRESH), to support efficient view updates for both distributive measures such as SUM, COUNT and non-distributive measures such as MEDIAN, CORRELATION. In so doing, `HaCube` is able to support more applications with data cube analysis in a data center environment. To the best of our knowledge, this is the first work to address data cube view maintenance in MapReduce-like systems. The experimental results showed that `HaCube` has significant performance improvement over Hadoop.

Our third contribution is to introduce a new graph OLAP model and the first distributed graph cube materialization scheme. We first proposed a new graph cube model, Hyper Graph Cube over the attributed graphs for graph OLAP and graph warehousing. On the basis of Hyper Graph Cube, we further illustrated how it supports different categories of queries and supports a new set of OLAP Roll-Up/Drill-Down operations. We then proposed several optimization techniques to tackle the problem of performing an efficient graph cube computation under the MR framework: a), our self-contained join strategy can reduce I/O cost. It is a general join strategy applicable to various applications which need to pass a large amount of intermediate joined data between multiple MR jobs. b), we combine cuboids to be processed as a batch so that the intermediate data and computation can be shared. c), a cost-based optimization scheme is used to further group batches into bags (each bag is a subset of batches) so that each bag can be pro-

cessed efficiently using a single MR job. d), a MR-based scheme is designed to process a bag. Furthermore, we proposed a cube materialization approach, MRGraph-Cubing, that employs the aforementioned techniques to process large scale attributed graphs. To the best of our knowledge, this is the first parallel graph cubing solution over large-scale attributed graphs under the MR-like framework. Finally, we conducted extensive experimental evaluations based on both real and synthetic data. The experimental results showed that our parallel Hyper Graph Cube solution is effective, efficient and scalable.

6.2 Future Research Directions

The continued growth of data sizes and advent of novel applications ensures that the area of big data analysis has many interesting research challenges. We discuss some of these interesting directions.

Graph OLAP on High Dimensional Attributed Graphs. Hyper Graph Cube faces the similar challenge as traditional data warehousing and OLAP does while handling the high dimensional datasets. For graph cube materialization, our current schemes mainly focus on the full cube materialization to precompute all the views in advance. Full materialization provides the best query response, but takes a large amount of storage space. Therefore, due to the storage limitation in different systems, the existing solutions for traditional high-dimensional OLAP (e.g. partial cube materialization [94][71][32][34], shell-fragment [45]) can be extended to tackle the challenge here. However, given the unique feature of the graph OLAP, there remain a lot of challenges of extending these techniques to support graph OLAP on high dimensional attributed graphs which will be an interesting future work.

View Allocation. As the size of data increases, the size of the materialized views increases as well, especially for graph data warehousing. We have not designed a technique

to specially allocate the views properly in a distributed data warehousing environment. Given a distributed system, the views should be partitioned and allocated to different machines in a manner such that user's queries can be efficiently supported. For instance, if all the hot views are allocated to the same machine, these machines will be frequently visited and the performance may be significantly reduced. Therefore, an efficient and effective view allocation strategy is needed to balance the query load across the system nodes. Meanwhile, the view allocation may also effect the view update performance. Thus, the view allocation scheme should also ensure view update efficiency.

Indexing on Attributed Graphs. Due to the astounding growth of property graphs, it is very costly to answer the query by scanning. Our current graph OLAP solution does not explicitly take into consideration of indexes. The indexing techniques can be further integrated into our solution to highly improve the graph retrieval efficiency. There are a lot of existing works focusing on graph indices [86][87][76]. However, most of the existing works focus on indexing on non-attributed graphs where graph has no attributes with the vertices and edges. Very few techniques are for attributed graphs. Indexing attributed graphs is challenging, especially considering to building index with regard to both graph structure and attributes. Therefore, designing effective indexing techniques on attributed graphs could be an interesting future work.

BIBLIOGRAPHY

[1] Graphlab. http://graphlab.org/.

[2] Hadoop. http://hadoop.apache.org/.

[3] Snap stanford network analysis platform. Available at:http://snap.stanford.edu/.

[4] Tacc longhorn cluster. https://www.tacc.utexas.edu/.

[5] Tpc-h , ad-hoc, decision support benchmark. Available at: www.tpc.org/tpch/.

[6] Alberto Abelló, Jaume Ferrarons, and Oscar Romero. Building cubes with mapreduce. In *DOLAP*, pages 17–24, 2011.

[7] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *VLDB*, pages 506–521, 1996.

[8] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[9] Daniel Archambault, Tamara Munzner, and David Auber. Topolayout: Multi-level graph layout by topological features. *IEEE TRANS. VISUALIZATION AND COMPUTER GRAPHICS*, 13:2007, 2007.

[10] David J. Balding. A tutorial on statistical methods for population association studies. *Nature Reviews Genetics*, 7:781–791, 2000.

[11] Kevin S. Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD*, pages 359–370, 1999.

[12] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: Mapreduce for incremental computations. In *SOCC*, pages 7:1–7:14, 2011.

[13] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *WWW*, pages 595–602, 2004.

[14] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[15] Chee-Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD conference*, SIGMOD '98, pages 355–366, 1998.

[16] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, March 1997.

[17] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S. Yu. Graph olap: Towards online analytical processing on graphs. In *ICDM*, pages 103–112, 2008.

[18] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S. Yu. Graph olap: a multi-dimensional framework for graph data analysis. *Knowl. Inf. Syst.*, 21(1):41–63, 2009.

[19] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009.

[20] Heather J. Cordell. Detecting gene-gene interactions that underlie human diseases. *Nature Reviews Genetics*, 10:392–404, 2009.

[21] N. G. de Bruijin. Asymptotic methods in analysis. pages 102–109. New York: Dover, 1981.

[22] Jeffrey Dean and Sanjay Ghemawa. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th symposium on operating systems design and implementation*, OSDI '04, pages 137–150, 2004.

[23] Iman Elghandour and Ashraf Aboulnaga. Restore: Reusing results of mapreduce jobs. *PVLDB*, 5(6):586–597, 2012.

[24] Kelly A. Frazer, Dennis G. Ballinger, David R. Cox, et al. A second generation human haplotype map of over 3.1 snps. *Nature*, 449:851–861, 2007.

[25] Rainer Gemulla. *Sampling algorithms for evolving datasets*. PhD thesis, 2008.

[26] Sanjay Goil and Alok N. Choudhary. High performance olap and data mining on parallel computers. *Data Min. Knowl. Discov.*, 1(4):391–417, 1997.

[27] Sanjay Goil and Alok N. Choudhary. High performance multidimensional analysis of large datasets. In *DOLAP*, pages 34–39, 1998.

[28] Sanjay Goil and Alok N. Choudhary. A parallel scalable infrastructure for olap and data mining. In *IDEAS*, pages 178–186, 1999.

[29] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, OSDI'12, pages 17–30, 2012.

[30] Benjamin J Grady, Eric Torstenson, Scott M Dudek, Justin Giles, David Sexton, and Marylyn D Ritchie. Finding unique filter sets in plato: A precursor to efficient interaction analysis in gwas data. In *Pacific Symposium on Biocomputing*, pages 315–326, 2010.

[31] Jim Gray, Adam Bosworth, Andrew Layman, Don Reichart, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *ICDE*, pages 152–159, 1996.

[32] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[33] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD*, pages 1–12, 2001.

[34] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. Revisiting the partial data cube materialization. In *Proceedings of the 15th international conference on Advances in databases and information systems*, ADBIS'11, pages 70–83, 2011.

[35] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *ACM SIGMOD*, pages 205–216, 1996.

[36] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Incremental recomputations in mapreduce. In *CloudDB*, pages 7–14, 2011.

[37] Tony Kam-Thong, Darina Czamara, Koji Tsuda, Karsten Borgwardt, Cathryn M. Lewis, Angelika Erhardt-Lehmann, Bernhard Hemmer, Peter Rieckmann, Markus Daake, Frank Weber, Christiane Wolf, Andreas Ziegler, Benno Ptz, Florian Holsboer, Bernhard Schlkopf, and Bertram Mller-Myhsok. Epiblaster-fast exhaustive two-locus epistasis detection strategy using graphical processing units. *European Journal of Human Genetics*, 2010.

[38] Tony Kam-Thong, Benno Ptz, Nazanin Karbalai, Bertram MllerMyhsok, and Karsten Borgwardt. Epistasis detection on quantitative phenotypes by exhaustive enumeration using gpus. *Bioinformatics*, 2011.

[39] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.

[40] M Kurant, M. Gjoka, Yan Wang, Zack W. Almquist, Carter T. Butts, and Athina Markopoulou. Coarse-Grained Topology Estimation via Graph Sampling. In *WOSN*, Helsinki, Finland, 2012.

[41] Laks V. S. Lakshmanan, Jian Pei, and Jiawei Han. Quotient cube: How to summarize the semantics of a data cube. In *VLDB*, pages 778–789, 2002.

[42] Laks V. S. Lakshmanan, Jian Pei, and Yan Zhao. Qc-trees: An efficient summary structure for semantic olap. In *SIGMOD Conference*, pages 64–75, 2003.

[43] Ralf Lämmel and David Saile. Mapreduce with deltas. In *PDPTA*, 2011.

[44] Ki Yong Lee and Myoung Ho Kim. Efficient incremental maintenance of data cubes. In *VLDB*, pages 823–833, 2006.

[45] Xiaolei Li, Jiawei Han, and Hector Gonzalez. High-dimensional olap: A minimal cubing approach. In *VLDB*, pages 528–539, 2004.

[46] Hongjun Lu, Xiaohui Huang, and Zhixian Li. Computing data cubes using massively parallel processors. In *in Proc. 7th Parallel Computing Workshop (PCW97*, 1997.

[47] Li Ma, Birali Runesha, Daniel Dvorkin, John R Garbe, and Yang Da. Parallel and serial computing tools for testing single-locus and epistatic snp effects of quantitative traits in genome-wide associatin studies. *BMC Bioinformatics*, 9:315, 2008.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[49] Jason H. Moore, Folkert W. Asselbergs, and Scott M. Williams. Bioinformatics challenges for genome-wide association studies. *Bioinformatics*, 26:445–455, 2010.

[50] Jason H. Moore and Scott M. Williams. Epistasis and its implications for personal genetics. *Am. J. Hum. Genet.*, 85:309–320, 2009.

[51] Shinichi Morishita and Jun Sese. Traversing itemset lattices wwith statistical metric pruning. In *In Proc. of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 226–236, 2000.

[52] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, pages 100–111, 1997.

[53] Seigo Muto and Masaru Kitsuregawa. A dynamic load balancing strategy for parallel datacube computation. In *DOLAP*, pages 67–72, 1999.

[54] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. Distributed cube materialization on holistic measures. In *ICDE*, pages 183–194, 2011.

[55] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. Data cube materialization and mining over mapreduce. *IEEE Trans. Knowl. Data Eng.*, 24(10):1747–1759, 2012.

[56] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *SIGMOD Conference*, pages 419–432, 2008.

[57] Raymond T. Ng, Alan S. Wagner, and Yu Yin. Iceberg-cube computation with pc clusters. In *SIGMOD*, pages 25–36, 2001.

[58] Stefano Paraboschi, Giuseppe Sindoni, Elena Baralis, and Ernest Teniente. Materialized viewsin multidimensional databases. In *Multidimensional Databases*, pages 222–251. 2003.

[59] Mee Yong Park and Trevor Hastie. Penalized logistic regression for detecting gene interactions. *Biostatistic*, 9:30–50, 2009.

[60] Shaun Purcell, Benjamin Neale, Kathe Todd-Brown, Lori Thomas, et al. Plink: a tool set for whole-genome association and population-based linkage analyses. *Am. J. Hum. Genet*, 81:559–575, 2007.

[61] Qiang Qu, Feida Zhu, Xifeng Yan, Jiawei Han, Philip S. Yu, and Hongyan Li. Efficient topological olap on information networks. In *DASFAA (1)*, pages 389–403, 2011.

[62] Sriram Raghavan and Hector Garcia-Molina. Representing web graphs. In *ICDE*, pages 405–416, 2003.

[63] Kenneth A. Ross and Divesh Srivastava. Fast computation of sparse datacubes. In *VLDB*, pages 116–125, 1997.

[64] Frank Ruskey and Carla D. Savage. A gray code for the combinations of a multiset. *Eurepean Journal of Combinations*, 17:493–500, 1996.

[65] Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-sparql: a hybrid engine for querying large attributed graphs. CIKM '12, pages 335–344, 2012.

[66] Sunita Sarawagi, Rakesh Agrawal, and Ashish Gupta. *On Computing the Data Cube*. Research report. IBM Research Division, 1996.

[67] Robert Sedgewick. Algorithms in c. In *Chapter 8*. Addison-Wesley Publishing Company, 1990.

[68] Kuznecov Sergey and Kudryavcev Yury. Applying map-reduce paradigm for parallel closed cube computation. In *DBKDA*, pages 62–67, 2009.

[69] Jayavel Shanmugasundaram, Usama M. Fayyad, and Paul S. Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In *KDD*, pages 223–232, 1999.

[70] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB'98*, pages 488–499, 1998.

[71] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98*, pages 488–499, 1998.

[72] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: shrinking the petacube. In *SIGMOD Conference*, pages 464–475, 2002.

[73] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *SIGMOD Conference*, pages 567–580, 2008.

[74] Xiang Wan, Can Yang, Qiang Yang, Hong Xue, Nelson L.S. Tang Xiaodan Fan, and weichuan Yu. Boost: a fast approach to detecting gene-gene interactions in genome-wide case-control studies. *Am. J. Hum. Genet*, 87:325–340, 2010.

[75] Wei Wang, Hongjun Lu, Jianlin Feng, and Jeffrey Xu Yu. Condensed cube: An efficient approach to reducing data cube size. In *ICDE*, pages 155–165, 2002.

[76] Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, Shanshan Ying, and Hai Jin. An efficient graph indexing method. In *ICDE*, pages 210–221, 2012.

[77] Zhengkui Wang, Divyakant Agrawal, and Kian-Lee Tan. Cosac: A framework for combinatorial statistical analysis on cloud. *IEEE Transactions on Knowledge and Data Engineering*, 25(9):2010–2023, 2013.

[78] Zhengkui Wang, Qi Fan, Huiju Wang, Kian-Lee Tan, Divyakant Agrawal, and Amr EI Abbadi. Hyper graph cube computation over large-scale attributed graphs. In *ICDE*, 2014.

[79] Zhengkui Wang, Kian-Lee Tan, Divyakant Agrawal, Amr EI Abbadi, and Xiaolong Xu. Hacube: Extending the mapreduce framework for data cube materialization and view maintenance. In *Submitted for publication*.

[80] Zhengkui Wang, Yue Wang, Kian-Lee Tan, Limsoon Wong, and Divyakant Agrawal. Ceo: A cloud epistasis computing model in gwas. In *IEEE International Conference on Bioinformatics and Biomedicine*, pages 85–90, 2010.

[81] Zhengkui Wang, Yue Wang, Kian-Lee Tan, Limsoon Wong, and Divyakant Agrawal. eceo: An efficient cloud epistasis computing model in genome-wide association study. *Bioinformatics*, 27(8):1045–1051, 2011.

[82] Jing Wu, Bernie Devlin, Steven Righquist, Massimo Trucco, and Kathryn Roeder. Screen and clean: a tool for identifying interactions in genome-wide association studies. *Genet. Epidemiol.*, 34:275–285, 2010.

[83] Tong Tong Wu, Yifang Chen, Trevor Hastie, Eric Sobel, and Kenneth Lange. Genome-wide association analysis by lasso penalized logistic regression. *Biostatistic*, 25:714–721, 2009.

[84] Dong Xin, Jiawei Han, Xiaolei Li, Zheng Shao, and Benjamin W. Wah. Computing iceberg cubes by top-down and bottom-up integration: The starcubing approach. *IEEE Trans. Knowl. Data Eng.*, 19(1):111–126, 2007.

[85] Dong Xin, Jiawei Han, Xiaolei Li, and Benjamin W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *VLDB*, pages 476–487, 2003.

[86] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *SIGMOD Conference*, pages 335–346, 2004.

[87] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.*, 30(4):960–993, 2005.

[88] Jinguo You, Jianqing Xi, Pingjian Zhang, and Hu Chen. A parallel algorithm for closed cube computation. In *ACIS-ICIS*, pages 95–99, 2008.

[89] Ning Zhang, Yuanyuan Tian, and Jignesh M. Patel. Discovery-driven graph summarization. In *ICDE*, pages 880–891, 2010.

[90] Xiang Zhang, Shunping Huang, Fei Zou, and Wei Wang. Team: efficient two-locus epistasis tests in human genome-wide association study. *Bioinformatics*, 26:217–227, 2010.

[91] Xiang Zhang, Shunping Huang, Fei Zou, and Wei Wang. Team: efficient two-locus epistasis tests in human genome-wide association study. *Bioinformatics [ISMB]*, 26(12):217–227, 2010.

[92] Xiang Zhang, Feng Pan, Yuying Xie, Fei Zou, and Wei Wang. Coe: A general approach for efficient genome-wide two-locus epistasis test in disease association study. In *RECOMB*, pages 253–269, 2009.

[93] Xiang Zhang, Fei Zou, and Wei Wang. Fastanova: an efficient algorithm for genome-wide association study. In *SIGKDD*, pages 821–829, 2008.

[94] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube: on warehousing and olap multidimensional networks. In *SIGMOD Conference*, pages 853–864, 2011.

[95] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD Conference*, pages 159–170, 1997.

[96] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, 1995.