ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αλγόριθμοι Για Εξισορρόπηση Πολυδιάστατου Φορτίου

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Σπαντιδάκη Ιωάννη

Επιβλέποντες:	Δημήτρης Φωτάκης	Πάρης Κούτρης
	Επίκουρος Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Wisconsin

Αθήνα, Ιούνιος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΛΟΓΙΚΗΣ ΚΑΙ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Αλγόριθμοι Για Εξισορρόπηση Πολυδιάστατου Φορτίου

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Σπαντιδάκη Ιωάννη

Επιβλέποντες: Δημήτρης Φωτάκης, Επίκουρος Καθηγητής Ε.Μ.Π. — Πάρης Κούτρης, Επίκουρος Καθηγητής Wisconsin

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 19 Ιουνίου 2017.

....	....	....
Δημήτρης Φωτάκης	Παράσχος Κούτρης	Αριστείδης Παγουρτζής
Επίκουρος Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Wisconsin	Αναπληρωτής Καθηγητής Ε.Μ.Π .

Αθήνα, Ιούνιος 2017

....
Σπαντιδάκης Ιωάννης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τους επιβλέποντες καθηγητές, κύριους Δημήτρη Φωτάκη και Πάρη Κούτρη, για τη συμβολή, το χρόνο και τη συνεχή καθοδήγηση τους στη διεκπεραίωση αυτής της διπλωματική εργασίας. Η ενθάρρυνση, ο ενθουσιασμός και οι καίριες συμβουλές τους, με βοήθησαν να γνωρίσω την ερευνητική διαδικασία εκ των έσω, δίνοντας μου συνεχώς κίνητρο για να συνεχίσω. Ειδικά θα ήθελα να ευχαριστήσω τον κ Κουτρή για την εμπιστοσύνη που μου έδειξε, και τις πολλές συναντήσεις που είχαμε που γαλούχησαν τον τρόπο σκέψης και προσέγγισης μου στην έρευνα. Ακόμα τον κ.Φωτάκη που υπήρξε πρότυπο καθηγητή και επιβλέποντα, τόσο για την επιλογή του ερευνητικού πεδίου των Αλγορίθμων, όσο και για την βοήθεια του στην εύρεση των προσωπικών μου στόχων αλλά και στην επίτευξή τους.

Επιπλέον, οφείλω ένα μεγάλο ευχαριστώ σε όλους τους φίλους που γνώρισα μέσα στη σχολή και είχα την ευκαιρία να συζητήσω και να συνεργαστώ κατά τη διάρκεια των 5 αυτών χρόνων. Το πάθος για τη γνώση που μοιραζόμασταν, ο ενθουσιασμό τους αλλά και οι προσωπικότητες τους συνέβαλλαν καθοριστικά σε αυτά τα παραγωγικά και γεμάτα υπέροχες αναμνήσεις φοιτητικά χρόνια.

Τέλος, ένα μεγάλο ευχαριστώ στην οικογένεια μου, τόσο στην Αθήνα όσο και στα Χανιά, που τόσα χρόνια με στηρίζει αμέριστα και στα εύκολα και στα δύσκολα. Η συμπαράσταση τους ήταν και είναι ανεκτίμητη.

Περίληψη

Σε όλα τα μοντέρνα συστήματα βάσεων δεδομένων, οι πληροφορίες διαμοιράζονται σε πολλούς υπολογιστές ή επεξεργαστές, και οι υπολογισμοί γίνονται παράλληλα με την ελάχιστη δυνατή επικοινωνία μεταξύ αυτών. Για το σκοπό αυτό, έχουν αναπτυχθεί πολλά μοντέλα παράλληλου υπολογισμού, καθώς και πολλοί αλγόριθμοι διαμοιρασμού δεδομένων σε επεξεργαστές. Ο συνήθης στόχος των αλγορίθμων αυτών είναι να ελαχιστοποιήσουν το φορτίο που θα λάβει ο υπολογιστής με τα περισσότερα δεδομένα, καθώς ο χρόνος ανταπόκρισης του συστήματος είναι αντιστρόφως ανάλογος του φόρτου εργασίας των επεξεργαστών του.

Στην εργασία αυτή βασιζόμαστε σε ένα Μαζικό και Παράλληλο Υπολογιστικό μοντέλο, MPC, με σκοπό τον υπολογισμό queries χωρίς ενδιάμεση επικοινωνία των υπολογιστών μεταξύ τους. Ενδιαφερόμαστε για τη μελέτη αλγορίθμων εξισορρόπησης πολυδιάστατων φορτίων, οι οποίοι γενικεύουν τους κλασσικούς αλγόριθμους εξισορρόπησης, αφού αντί να έχουμε βαθμωτά μεγέθη να διαμοιράσουμε, έχουμε διανυσματικά. Παρουσιάζουμε τα πρόσφατα αποτελέσματα τόσο σε deterministic όσο και σε randomized αλγορίθμους και μελετάμε την απόδοση τους σε σχέση με τη μορφή που έχουν τα δεδομένα εισόδου και το query που πρέπει να υπολογίσουν. Στη συνέχεια αποδεικνύουμε ένα νέο αποτέλεσμα για έναν από τους αλγόριθμους αυτούς, εξασφαλίζοντας τη σχεδόν βέλτιστη κατανομή των δεδομένων, σε περιπτώσεις που προηγουμένως δεν είχαν λυθεί.

Λέξεις-Κλειδιά: Πολυδιάστατη Εξισορρόπηση Φορτίου, Αλγόριθμοι Διαμοιρασμού Δεδομένων, Προσεγγιστικοί Αλγόριθμοι, Βελτιστοποίηση.

Abstract

In all the modern database systems, information is spread among multiple computers or processors, and the computation needed is done in parallel, using the minimum possible communication between them. To serve that purpose, numerous models of parallel processing and many distribution algorithms have been developed. In most of the cases, the goal of those algorithms is to increase the efficiency of the system by minimizing the maximum load among these computers. This is happening because the response time of a systems is inversely proportional to the workload of its computational units.

In this thesis, we work under a Massive Parallel Computational Model (MPC) to achieve the computation of a certain kind of queries, without any communication between the processors. We are interested in studying multidimensional load balancing algorithms, which are more general than the classic load balancing algorithms, since instead of distributing scalar values, they distribute vectors. We present the most recent results in deterministic and randomized algorithms that solve that problem, and we study their performance in relation with the shape of the input data and the query that has to be computed. Finally, we prove a new result about one of those algorithms, guaranteeing an almost optimal distribution of the data, in cases that were not previously solved.

Keywords: Multidimensional Load Balancing, Distribution Algorithms, Approximation Algorithms, Optimization.

Contents

Chapter 1

Introduction

It is a common fact that we are living in the information era. There is an exponential expansion of data that are generated constantly from multiple sources, and this situation is something that is not going stop. Many believe that data is to this century what oil was to the previous one. Flows of data have created new infrastructures, new businesses, new politics and in general new economies. These data define aspects of our every day lives and have become an inseparable part of our selves. This data explosion comes along with many technological challenges concerning how are we able to store the data in order to use them. Rising to the challenge, many technological achievement were made in the last 20 years both in hardware and in software. Specifically, we developed systems that could store those information and many sophisticated algorithms in order to distribute the data optimally and use them efficiently. In this thesis, we will discuss some distribution algorithms that are trying to balance the load of the data between the computers in order to maximize their efficiency.

Nowadays, even the more advanced super computers are not fast enough on their own to rise up to the technological demands. The number of data that needs to be computed cannot be stored by a single computer and even if they can, their size makes even the most powerful machines slow to produce results. For that reason we have created distributed systems and clusters with many different computers that work in parallel on the same task. That way, we decrease both the load of data every computer has to hold and the computation time needed to complete each task. This necessary change to a network of computers that collaborate and communicate in order to produce the desirable result, gave rise to new algorithms that tried to optimize the way those data will be distributed. Their goal most of the times is to balance the total load of the input data among the many processors in a way that there would not be some idle machines and others doing almost all the work.

Distribution and load balancing algorithms are encountered in many other aspects of our lives that have nothing to do with computers and networks. Fields like human resource planning, production planning and transportation are only few of the numerous examples. It was these problems that started the motivated mathematician to develop tools to analyze them, before the field of computer science burgeoned into what it is in the 21st century. Nowadays, their application in both computer science, like networking and data bases, and operational research like optimization, constantly give motives to scientist to experiment with new idea and tools. Last but not least, besides the many applications that are closely related with businesses, those problems are also important from a purely algorithmic viewpoint, as the have rich and highly interesting combinatorial structure, tempting contemporary computer scientist and mathematicians to study them.

In order to develop these algorithms for finding optimal distributions, we first need to define the model we are working on. Many different models have been created since the early 90's, each one with some similarities and differences depending on what was the application that it aimed to model. Some famous models are the Parallel Random Access Machine (PRAM), Bulk Synchronous Parallel (BSP), LogP and others. The model we are going to work is called Massive Parallel Computation model (MPC) and is has many similarities with the BSP model. The main difference is that the complexity of an algorithm takes account only the load of each server and the communication required, not the computation task needed by each processor. The reason behind this modeling choice is that nowadays the main bottleneck in distributed systems like these is the communication between them rather than the computations that need to be done. We will discuss more about this model later.

When it comes to load balancing and scheduling, those two fields are different sides of the same coin, there have been many research and countless results even before the modern bloom of computer science. Many approximation ratios have been proven in different cases and a great number of heuristic have been studied both in theory and in practice. In this thesis, we will focus on algorithms about multidimensional load balancing. This means that instead of the classic case where we have to balance scalar values among a number of machines, we will study a more general one where we must balance vectors instead. Many jobs in real life involve more than two components and cannot be represented by just one scalar value. Applications like parallel query optimization, resource allocation and cutting stock are efficiently solve using vector load balancing. Our personal motivation for dealing with that problem comes from the field of Data Bases.

In this thesis, we are studying algorithms to compute multi-way joins, which is a central task in large scale data analytics systems. More specifically, we are analyzing a new parallel multi-way join algorithm named HyperCube or Shares algorithm which computes the join in a singe step by organizing the available servers in the cluster on a multi-dimensional hypercube. We focus on on the following question: Can we construct an efficient algorithm that distributes the tuples of a relation such that the load is always as close to the optimal value as possible?

We are taking two different approaches. In both cases we split the data into many processors, have them compute the joins locally and then we aggregate the results. The first is by using deterministic algorithms like vector load balancing. We will discuss how existing algorithms that balance vector, can be used to distribute the tuples into the processors achieving a deterministic approximation to the best possible load. The second, is through hashing functions, that map each of the tuples of the relation to one processor located in the hypercube and achieving a near optimal result with high probability. Both the deterministic and the randomized approaches offer insight to the problem and both their analysis and the ideas behind it are of great theoretical interest.

Through the analysis we will present in the next chapters, the correlation of the problems we study with classical problems like balls into bins and simple load balancing will be obvious. For that reason we study these problem and their results, before presenting the algorithms, in order to use them for our analysis when there is a chance for a reduction. Furthermore, we introduce an important mathematical tool that is used widely in analyzing the performance of randomized algorithms, the Chernoff bound. Later we use that bound along with some other tools to prove the efficiency of the algorithms in place.

After presenting the work that is done in this direction, we focus on how to generalize it for any kind of input. So far, the main result assume that the data are of some uniformity and there are not skew among them. An informal definition of skew is when there is a single or a group of values, that appear in the input data more times than a certain threshold. Ideally, we would like each computer to receive the same amount of data. Formally, if we have M bits of data and p processors or computers, we would wish the load of every one of them would be $\frac{m}{p}$ which is the more equal distribution we can achieve. Unfortunately, this total balance in not always feasible in all the cases, and usually, the reason why is skew. When there exists skew in our data we have to change the way we treat them if we want to be equally effective. As you can begin to understand, that kind of an input has a huge effect on the efficiency of the algorithms and needs to be dealt with.

In the algorithms we are studying, in the final chapter, we present some new proofs for the cases of skew in low dimensions of the hypercube. During these proofs many new problems arise and yield interested questions with both practical and theoretical aspects. The answer to these questions is the key in order to generalize those algorithms to work for any input with or without skew. We will now present the organization of the Thesis.

1.1 Organization of the thesis

In the 2^{nd} Chapter we are going to discuss some basic concepts that facilitate the studying of the rest of the thesis. These concept are:

- The MapReduce and the MPC model that provide enough knowledge about the model we are working on.
- Some basic introduction about conjunction queries in order to get familiar with what are the applications of the algorithm and what we are trying to solve .
- The definition and the importance both Load Balancing and Vector Load Balancing and how exactly this applies to our algorithm.

In the 3^{rd} Chapter we present some mathematical tools. More specifically, the Chernoff that is used to prove the approximation ratio of the randomized algorithms and the Balls into Bins problem, which is a classic probability problem with many application including one in the Load Balancing in this thesis.

In the 4^{th} Chapter we give a formal definition to the problem we are trying to solve. We first present the main algorithm that we are going to work with: the HyperCube algorithm. We explain how it works, and what is our motivation for the problems studying. Furthermore, we explain the effect of skew in the input and we present the lower bounds of the problem that we are trying to solve.

In the 5^{th} Chapter we provide two deterministic algorithms for distributing the tuples. One for the case of 1 dimension and one for the case of 2. The second one is based in a combination of the first one and an algorithm for solving the vector load balancing problem. Furthermore, it points out the difficulties that the randomized algorithm comes to solve.

In the 6^{th} and last Chapter we give a formal presentation of the HyperCube algorithm with the use of hashing functions. This solves the general case where there is no skew in the data. We then provide a proof for the case of 1 and 2 dimensions that gives near optimal result even for the case that there is skew in the data. Finally, we present the problems we are trying to solve now to generalize it for higher dimensions and some other open problems that come with that effort.

Chapter 2

Basic Concepts

The purpose of this chapter is to introduce the reader to Basic Concepts that will be important to understand the work that follows. For that reason we present basic algorithms and models, explaining why they are significant for today's applications.

2.1 MapReduce

2.1.1 MapReduce Definition

A MapReduce algorithm is a programming model that implements 2 parts, the mapping part and the reducing part, thus the name MapReduce. First the Map job takes the input data and process them to produce key-value pairs. Then the Reduce job takes those key-value pairs as an input and then combines or aggregates them to produce the final results. The Reduce job is always performed after the Map job. So MapReduce parcels out work to various nodes within the cluster or map, and it organizes and reduces the results from each node into a cohesive answer to a query.

As we mention before, due to the huge amount of data modern companies have to deal with, parallel algorithms that work in distributed systems are much more efficient and sometimes the only practical solution. For that reason, models like MapReduce are very popular, and used widely in modern systems like Apache Hadoop and Google's MapReduce. Finally other than being faster and more practical it is also fault-tolerant. Each node periodically reporting its status to a master node. So if a node doesn't respond as expected, the master node re-assigns that piece of the job to other available nodes in the cluster. That way faults that may happen in a single machine have no significant result to the whole system. For that reason MapReduce is resilient and can be run on inexpensive commodity servers

MapReduce is composed of several components, including:

- JobTracker : the master node that manages all jobs and resources in a cluster
- TaskTrackers : agents deployed to each machine in the cluster to run the map and reduce tasks
- JobHistoryServer : a component that tracks completed jobs, and is typically deployed as a separate function or with JobTracker

To get a better understanding of how the MapReduce model works we will provide a simple example.

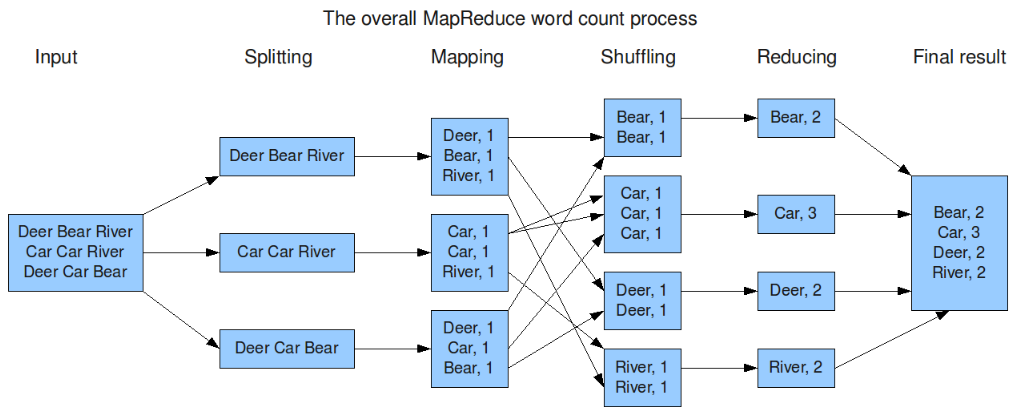
2.1.2 MapReduce Example

Imagine that you want to count the number of instances of each word in a book of 100 pages. Of course you want to do it fast, lets say in an hour. How are you going to do that. It is easy to see that it is not efficient to do it all by yourself, page by page, because it will take you way too long and you are gonna get tired and possibly make mistakes in the process. So this is what you are going to do.

1. You are going to ask 26 of your friends to come and help you.
2. To each one of your friends, you give a page of the book and ask him to put each word on an other page, one word per page.
3. When someone finishes a page you give him another one. If he gets tired you take back his page and give it to another friend who is more willing to do it.
4. When all pages have been done, you put on a table 26 boxes. Each box represents a letter of the alphabet, you ask your friends to put all the words that begin with the same letter in the same box.
5. You ask each one to take a box and sort alphabetically all the pages in the boxes.
6. When the sort is done, you ask your colleagues to count the number of pages that have the same word and to give you the results.

After those 6 steps, if everyone has followed the rules, you have the number of instances of each word in less than one hour.

In the next figure, we see the processes to make a word count distributed on many hosts like we described.



First of all, you input is the number of pages with words in them that is going to be split. In this instance we have 3 pages with 3 words each page. Each page will be ”mapped” by the Jobtracker, in this case you, to the TaskTrackers, your friends. The mapping step will produce many associations of <key,value> pairs, in this case the key will be the word and the value 1 because it is one word. In a second time, all the key-values pairs produced, will be sorted according to the key. Finally, each node will perform a ”reduce” task. They are going to count all the key value pairs with the same key. As you can see, the final result that you are going to get is the number of instances of each word in the file.

This method is not new. In the Roman times, the census was conducted in a similar way. The census bureau would dispatch its people to each city in the empire. Each census taker in each city would be tasked to count the number of people in that city and then return their results to the capital. In order to determine the overall population of the empire, these results would be reduced to a single count, the sum of all cities. As you can see, this mapping of people to cities, in parallel, and then combining or reducing the results to a single figure, is much more efficient than sending a single person out to count every person in the empire, in a serial fashion, all by himself.

Finally it is clear why this model is fault-tolerant. If a single person fails to count a single page or city then you just reassign the job to another. JobHistoryServer, as we mentioned, keeps track of all the updates and let only the complete tasks to be taken into account. That way the chance of getting a wrong result is minimized with little to nothing cost in the speed of computation.

2.2 MPC and BSP models

The model we are going to work is more general than the MapReduce. To get a better understanding we will introduce the Massively Parallel Computation model (MPC) and the Bulk Synchronous Parallel model (BSP) which share many similarities. These models allows us to analyze algorithms in massively parallel environments. We will begin by introducing the MPC first and then point out the differences it has with the BSP.

2.2.1 The MPC Model

The MPC Model was introduced in [5] [4] and it consist of p servers, or processors, connected by a complete network of private channels. All the computations are performed by these servers and each one of them can communicate with any other in an indistinguishable way. The servers run the parallel algorithm in communication steps, or rounds, where each round consists of two distinct phases.

- **Communication Phase:** The servers exchange data, each by communication with all other servers. They both send and receive data.
- **Computation Phase:** Each server performs computation on the local data it has received during all previous rounds.

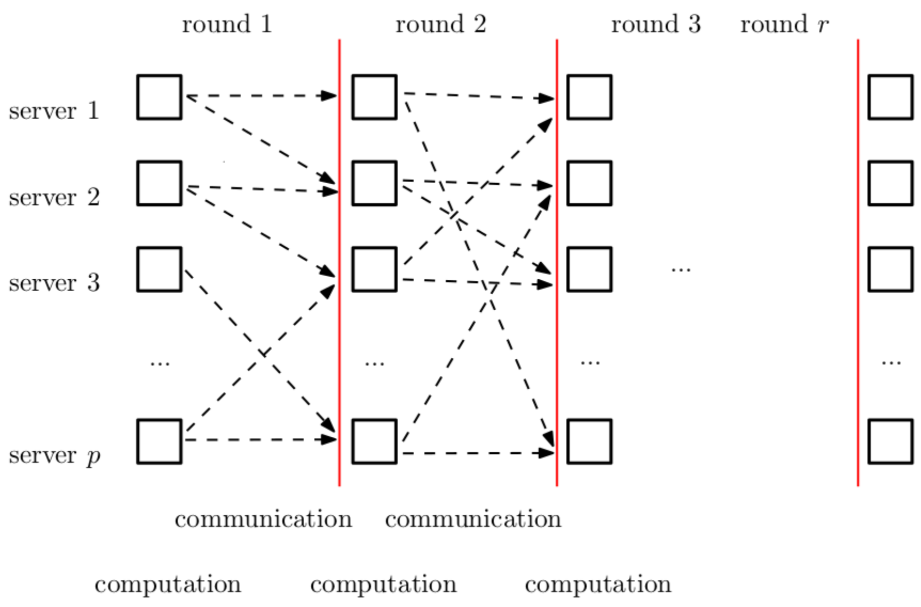
The input data of size M (in bits) is initially uniformly partitioned among the p servers, i.e each server stores M/p bits of data. Since there are not any particular assumptions on whether the data is partitioned according to a specific scheme, any parallel algorithm must work for an arbitrary data partition and any lower bound can use a worst-case initial distribution.

After the computation is competed, the output data is presented in the union of the output of the p servers. The complexity of a parallel algorith in the MPC model is captured by two basic parameters in the computation:

- **The number of rounds** r**.** This parameter captures the number of synchronization barriers that an algorithm requires during execution. The smaller the number is, the less synchronization the algorithm requires.
- **The maximum load** L**.** This parameter captures the maximum load among all servers at any round, where the load is the amount of data (in bits) received by a

server during a particular round. Let $L_{s,k}$ denote the number of bits that server s receives during round k. Then, the formal definition of L is

$$L = \max_{k=1,\dots,r} \{ \max_{s=1,\dots,p} L_{s,k} \}$$



It is worth noticing that the model does not restrict or captures the running time of the computation at each server. In other words the server can be as computationally powerful as we would like. This modeling choice means that the lower bounds must be information-theoretic, since they are based on how much data is available to each server and not on how much computation is needed to output the desired result.

The reason behind this modeling is that in practice, the main bottleneck, when it comes to efficiency of modern parallel algorithms, is indeed the communication part and the maximum load. Modern systems have a vastly amount of servers available for computation. This only makes the computational work easier but increases the time needed to exchange data between them and synchronize them. Furthermore, the MPC captures the computation part as a function of the size of the data. Since, for a round to end all the computations must be over, we basically care for the maximum load only, assuming of course that all the servers are equally powerful.

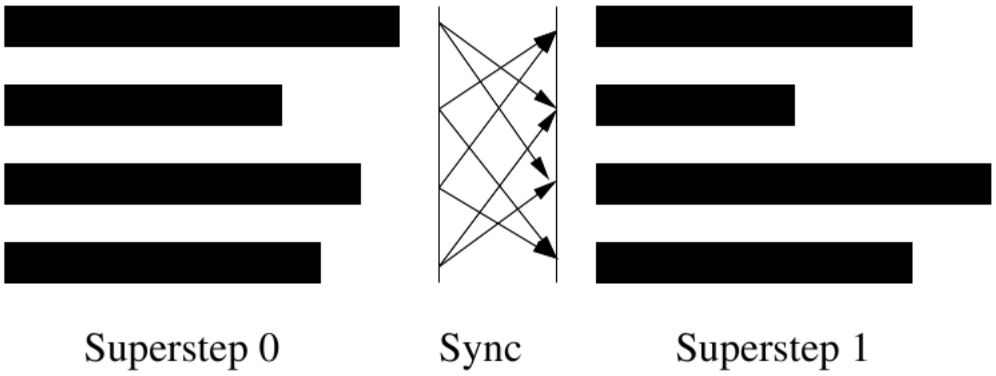
The MPC model, as it was defined in the previous paragraphs, allows arbitrary communication among servers at every round, which makes the theoretical analysis of multi-round algorithmms a very hard task. For that reason we simplifying the model by restricting the form of communication. In the new restricted model called the tuple-based MPC model the messages that each servers sends and receives are tuples instead of any bits. This restriction is natural and it captures a wide variety of algorithms including those based on MapReduce. Finally, it can be proven that this model is equivalent to the fully general MPC model with only a slight loss in efficiency since tuples can be used as Boolean functions, but we will not get into more details about that since it will not be relevant to the purpose of this thesis.

2.2.2 The BSP Model

The Bulk synchronous parallel (BSP) Model was introduced by Valiant [34] to address some issues of the Parallel Random Access Machine (PRAM) model. A BSP algorithm runs in supersteps: each superstep consists of local computation and asynchronous communication, followed by a synchronization barrier. The BSP model abstracts the communication in every superstep by introducing the notion of h-relation, which is defined as a communication pattern where the maximum number of incoming or outgoing messages per machine is h. The cost of a superstep i consists of three components:

- **The synchronization cost**: a constant l
- **The communication cost** h_i: the size of the $h - i$ relation
- **The computation cost** w_i: the longest running local computation

The total cost of a BSP algorithm is computed as a weighted sum of these terms over all steps: $\sum_i \left(l + gh_i + w_i\right)$



The BSP model and the MPC models are very similar to each other. As we mentioned the main differences are that the MCP model has removed the computational cost from consideration, and also does not allow any form of asynchronous communication. The second restriction allows us to prove lower bounds on the cost by using information-theoretic arguments. Finally, instead of measuring the total communication $\sum_i h_i$ over all supersteps, the MPC model considers the largest value of h_i, which as we defined is the load L.

The model we are working is the MPC model but the algorithms we are going to analyze run in a single step. This means that there will be no communication cost, and the only variable we want to minimize is the maximum load L. More specifically we are going to use the hypercube algorithm to compute conjunctive queries in a single step and we are going to optimize the distribution of tumples in order to minimize the variable L. But before we proceed to define what is a conjunctive query and what does the hypercube algorithm we will present an example of a single step algorithm in the MPC model and its analysis of the maximum load.

2.2.3 MPC Example: Set Intersection

The input consists of three sets R,S and T, each with m elements. The goal is to compute the intersection $R \cap S \cap T$ of those three sets. To distribute the elements across the p servers, we will use a hash function h: Dom $\to \{1, ...p\}$. As we mentioned, the algorithm will run in a single round. During the communication phase, each element $R(a)$ will be sent to server $h(a)$, each element $S(b)$ to server $h(b)$ and each element $T(c)$ to server $h(c)$ respectively. During the computation phase, each server s will perform a local intersection of the elements received from the three relations. It is clear that an element $a \in R \cap S \cap T$ will be in the output, since $R(a)$,$S(a)$ and $T(a)$ will be received by the server $h(a)$.

In order to compute the maximum load L, we will use a probabilistic argument to analyze the behavior of the hash function. It is well known that assuming h to be a perfectly random hash function, it can be shown that the load, (the number of elements of the relations, not bits) will be $O(m/p)$ with high probability when $m \gg p$.

2.3 Conjunctive Queries

The problem we assume we have to solve is the computation of a query in a Relational Database (RDB). An RDB is a collective set of multiple data sets organized by tables, records and columns. It establishes a well-defined relationship between database tables.

Tables communicate and share information, which facilitates data searchability, organization and reporting. We assume that the input of our problem is the database with all the relations it has. We define the arity of a relation as the number of columns in its table and we consider these information given at the input. Our algorithm is going to distribute the relations and compute a special, but very widely used, case of queries, the conjunctive queries (QC). We start by giving the definition of a QC.

Definition 2.1. A **Conjuctive Query** is a First Order Logic expression without negations of disjunctions. A CQ is an expression of the following form:

$$Q(x_1, ...x_n) \leftarrow R(a_1, ..., a_m), ..., S(c_1, ..., c_k)$$

It consist of two parts: The **head** and the **body**.

The set of variables $X = (x_1, ...x_n)$ before the arrow are called the head and the set of variables and constants $V = (a_1, ..., a_m, ..., c_1, ..., c_k)$ are called the body. Note that, all the variables appearing in the head must also appear in the body. Furthermore, the head doesn't necessarily have to contain only variables, it might as well contain constants. These constants will be returned in the results.

$R(a_1, ..., a_m)$ is called an atom where R is the name of the relation in the database schema. We denote the arity of the relation R_j with a_j as the number of variables it contains. If an atom does not contain any variables, only constants, it's a fact. So we can view a database as a set of facts.

A CQ q is full if every variable in the body of the query also appears in the head of the query. A CQ q is boolean if the head of the query is empty.

For instance, $q(x) = R(x, y)$ is not full, since the variable y does not appear in the head. The query $q() = R(x, y)$ is boolean.

A CQ q is sefl-join-free if every relation name appears exactly once in the body of the query.

The query $q(x, y, z) = R(x, y), S(y, z)$ is self-join-free, while the query $q(x, y, z) = R(x, y), R(y, z)$ is not, since the relation R appears twice in the query.

The result $q(D)$ of executing a conjunctive query q over a relational database D is all the assignments of values to the the vector $x \in X$ such that it satisfies all the atoms at the body of the query at the same time.

While in the general case we are going to compute full conjunctive queries, we can focus only on algorithms and bound for self-join free queries without any loss of generality. To prove that, we can do a simple transformation. Lets say we have a query q with repetitions of the same relation on its body. By giving distinct names to the repeated occurrences of the same relations we transform it into a join-free query. For instance, consider the query $q = S(x, y), S(y, z), S(z, x)$, which computes all triangles in a directed graph with edge set S. By renaming the atoms we get $q\prime = S_1(x, y), S_2(y, z), S_3(z, x)$ which is a self-join free query. In order for the algorithms to get the right result when executing $q\prime$ we must "copy" the relations that appear in multiple times in the body of the query. This increases the input size at most by a factor l where l is the number of relations. If this factor is constant then we can consider that the load of the processors will not change significantly. In the example above, the input for query $q\prime$ is 3 times larger and we get the correct result as if we were executing the original query q.

2.3.1 Conjunctive Query Example

Before we discuss about algorithms to compute conjunctive queries, lets give an example in order to get a better understanding. Assume we have the following database of students with two relations, R and S with arities two and one. We can translate relation $R(x, y)$ as student x is willing to collaborate with student y and the relation $S(z)$ as student z knows programming. We want to find a pair of students (x, y) with the following characteristics: y must be willing to collaborate with Alice, y must know programming and x must be willing to collaborate with y. We will write a query Q to return all those pairs that satisfy these conditions.

R		S
Tim	George	George
George	Helen	John
George	Alice	
Peter	John	
John	Alice	
Alice	Peter	

$$Q(x, y) \leftarrow R(x, y), R(y, Alice), S(y)$$

This query will return two tuples: (Tim,George) and (Peter,John).

We will not get into details how to compute the answer here but in the worst case we have to check all the tuples of a relation. In this example, it is simple to find the answer because the database is small and the query relatively simple. As you can imagine, this is not always the case in modern database systems. We must split the database to many processors and compute the result in parallel if we want to be efficient. This is exactly what we are going to do. Using the MPC model, we will distribute the vectors of data in the relational database into multiple processors, and we will compute the query in parallel. The way we are going to distribute the data among the processors is very crucial since the response time of the system, and thus its efficiency, is inversely proportional to the workload of its computational units. This means that we opt for an even distribution of data among these units if one is possible.

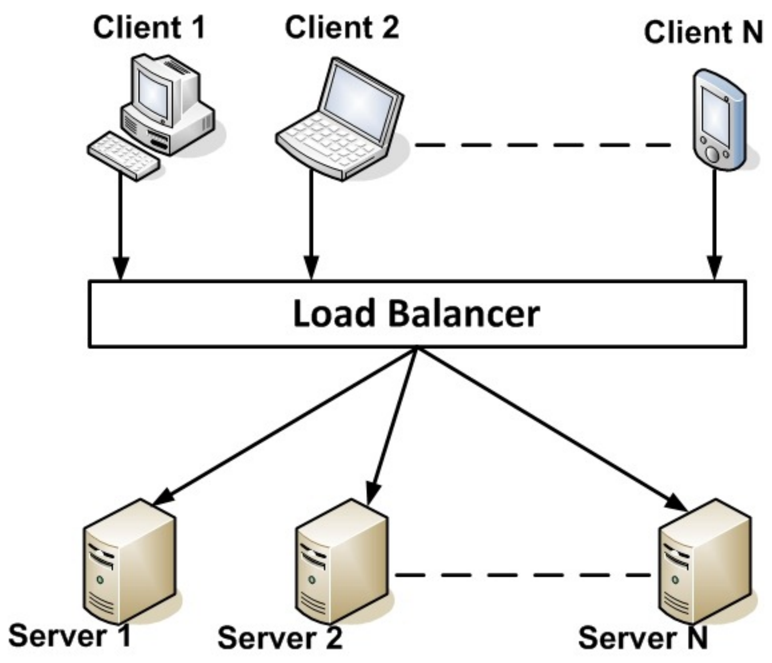
Having said that, it is getting clear that we have to solve a load balancing problem. The tuples can be concerned as loads with a unit of weight and the processors as servers that we want to distribute them to. The main difference here is that we do not only have one dimension, because the number of variables in the head of the query is more than one, and because we have to take skew into consideration the problem is not a classic scalar load balancing problem. Each tuple is a vector $\mathbf{x}$ and for that reason we have vector load balancing. But we must first introduce the scalar load balancing to get intuition about the vector and its difficulties.

2.4 Load Balancing

As we mentioned before, modern companies and websites deal with huge amount of data everyday. They must store them, process them and serve hundreds of thousands, if not millions, of concurrent requests from users or clients. Furthermore, they must return the correct text, image, video or application data, all in a fast and reliable manner. This complicated job cannot be done by a single computer even if that machine is of extreme capabilities and computational power. Therefore, in order to cost-effective scale to meet these high volumes, modern companies use a huge amount of servers, where they store their data and share the computations that need to be done on them. The way the distribution is done across the servers is a crucial factor for the performance of the system. The more Balanced the Load is, the more efficient the system becomes.

2.4.1 Load Balancing of Scalar Values

Load balancing is the procedure that aims to improve the distribution of workloads across multiple computing resources, such as computers, clusters or any processing units. Its main goal is to optimize the resource use, maximize throughput, minimize the response time and avoid overload of any single resource.



A load balancer acts as the ”traffic cop” sitting in front of the servers, routing client requests across all servers and deciding which workload or data will be distributed where. The algorithms behind the load balancer, are designed to maximize the speed and the capacity utilization and ensure that no one server is overworked, which could degrade performance. Furthermore, they increase reliability through redundancy and provide flexibility to add or subtract servers as demand dictates. We will not elaborate on the last two functions of the load balancer because the main focus or our work is in algorithms that are trying to approach optimal distributions.

Our objective to the load-balancing problems is to minimize the maximum load that a single server will receive. So the more even the distribution is the better the result would be. We aim to have the least possible amount of load to the server that has the most among the others. The key world here is approach. Load Balancing problems of that type have been proved to be NP-hard, which means that efficient algorithms, that run in polynomial time, cannot always find the best distribution. For that reason we are trying to build algorithms that find a near optimal solution in polynomial time.

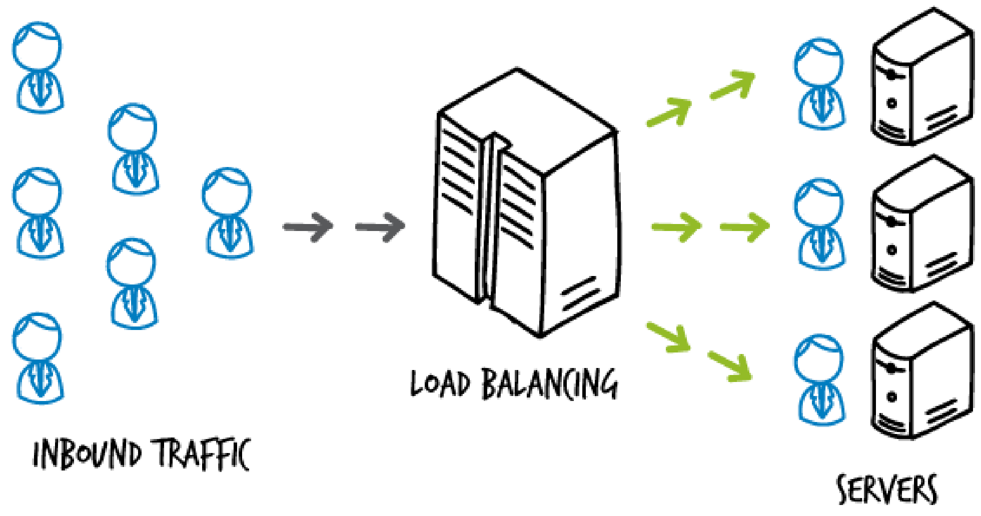
Load balancing problem has a strong correlation with scheduling problems. Instead of loads that need to be distributed to servers, we have jobs that need to be scheduled to machines. The size of the load is the amount of time it takes from a machine to finish a job. We consider all the machines identical which means that the each job takes the same amount of time to be finished in all machines. In addition, as we cannot split a load into more than one server, we cannot slit a job into two machines. Finally, minimizing the maximum load that a single server will receive is the same objective as to minimize the total makespan of all the jobs. From now on we may refer to loads as jobs and to servers as machines and vice versa. We can now give a formal definition for the scalar load balancing problem.

Definition 2.2 (Scalar Load Balancing)**.** There exist n machines and m jobs. A job j is associated with a scalar value a_j that denotes the load placed by job j. Let l_i denote the sum of a_j over all jobs j that are assigned to machine i. The goal is to assign jobs to machines such that we minimize the makespan $max_i l_i$.

This form of the problem is actually well studied in the literature since it is a classic optimization problem in both computer science and operations research. Many approximation algorithms have been developed from both theoretical scientist and practitioners, since it is of great importance in many technological and everyday life applications. The online load-balancing problem is also very interested for the same reason. At the online setting, we have to do a dynamic scheduling which means that the decision of scheduling a job can only be made online, when the job is presented to the algorithm.

2.4.1.1 Scalar LB Example

To get a better understanding we shall introduce a real life example of load balancing inspired by the academic life. Suppose that you are a professor in a university and the spring semester has just finished. Now that you are free of teaching courses you want to start doing more research with your 3 phd students. But before you do that, you need to grade all the exam-papers from the courses you were teaching this semester. Since you have 3 teaching assistants to assist you will split the job in order to grade them in parallel. You are interested in finding the best allocation of the exams papers to your phd students in order to finish all the exam papers as soon as possible under the restriction that each student must be assigned to a single grader. Assuming you can assess how much time each exam paper needs to be corrected based of the number of pages and all the phd students are the same as fast as grading speed concerns, then you need to solve a load balancing problem with the objective to minimize the time needed for the phd student with the most working load to finish. For instance if you have distributed all the papers but one and the load of your three phd students is 17,16 and 14 hours of expected grading. Since know that the last paper needs 2 hours to be graded, you will give this to the last phd student because you want them all to finish in 17 hours instead of 19 or 18. This an minimizing the makespan objective, where the load is the exam papers and the machines are the phd students.



Problem like these come up a lot in everyday application like queues in the bank, networks and job scheduling at the cpu of a personal computer. Furthermore, there exist many variations concerning the type of the machines, the possible constrains and the objective function you may have. But as we mention in the previous section, in this thesis we are not interested in just balancing scalar values, but instead we will try to balance vectors. The reason why is that while using the hypercube algorithm, the problem we have to solve trying to distribute the tuples to servers, is a multidimensional scheduling.

2.4.2 Vector Load Balancing

Many jobs in real life often involve more than two components and cannot be represented by just one scalar value. In applications like parallel query optimization, resource allocation and cutting stock we cannot apply the scalar Load Balancing algorithms and get satisfying results. In that case we have to use a vector to represent the job and the load balancing problem changes. We distribute vectors instead of scalar values and the machines have many components each with its own load.

Definition 2.3 (Vector Load Balancing)**.** There exist n machines each with d components. A job j is associated with a d-dimensional vector a_j, where the k^{th} coordinate ${a_j}^k$ that denotes the load placed on component k by job j. Let ${l_i}^k$ denote the sum of ${a_j}^k$ over all jobs j that are assigned to machine i. The goal is to assign jobs to machines such that we minimize the makespan $max_{i,k}{l_i}^k$.

The above definitions state that we no longer have scalar values but vectors of dimension d. The machines have also a vector where each coordinate i is the sum of all the coordinates i of the jobs that have been allocated to that machine. Suppose we take the maximum coordinate from each machine. Now we want to minimize the maximum value from those we just receive. This is the definition of the makespan for the vector load balancing problem.

2.4.2.1 Vector LB Example

In order to get the intuition behind that problem, we are going to extend the example with the professor and the phd students from the previous page. Suppose that now each student does not have only a final exam that needs grading, but also many assignments during the semester and a programming project to which you have to check upon the correctness and the efficiency of the code. Basically, every student that took the course is now represented by a load vector of dimension 3, where the three coordinates indicates the workload for grading the final exam, the assignments and the code respectfully. Furthermore, each phd student has 2 friends to help them with the grading. One is only grading assignments and the other is only checking and running the code. So basically, you now have to divide the vectors to the three groups of graders,each consist of a phd student and two of his friends, and you want them to finish all the grading as soon as possible. For instance you opt for a distribution of the load such as the group will finish the final projects after 14 hours the assignments in 13 and the coding in 15, than for one in which they will finish them in 12, 11 and 18 respectfully. This implies that when you assign the tasks you have to keep in mind not just a scalar value as to how much load each machine has but a whole vector, which complicates the problem further and demands sophisticated algorithms.

Chapter 3

Mathematical Tools

In this Chapter we will explain some of the mathematical tools we are going to use in the next chapters. More specifically, we are going to present some useful inequalities in probabilities like the Markov and the Chernoff bounds. These tools are widely used in randomized algorithms to bound the probability that a random variable will deviate far from its expectation. After that, we will introduce the classical balls into bins problem in different variations. This problem has an amazing property; despite its simplicity, it has a numerous of important application in computer science like Online Load Balancing and Hashing. We will present the problem, analyze it, and then point out the correlation it has with the multidimensional problem we are trying to solve.

These mathematical tools are crucial for understanding the randomized algorithm proposed in chapter 6 as a solution to the problem we have been studying. Furthermore, by using these tools, we manage to extend the analysis of the algorithm and prove our novel results. In this chapter, we assume that the reader has some familiarity with basic probability theory like independence, expectation, random variables and their distribution and moments.

3.1 Expectation and Deviation

Random variables and probabilities are witnessed in everyday life. There are many things in nature and in our society that we cannot predict exactly how there going to behave and this usually introduce uncertainty in our systems and decisions. Randomness is sometimes introduced because of some truly random things, but most of the times it is a reflection of our ignorance about the thing being observed, rather than something inherent to it.

Nevertheless, since we have not figure out a model that can predict the outcome of such events both in science and in everyday life, we have accepted the concept of uncertainty. While trying to get a better understanding of that uncertainty and deal with it, we developed some mathematical tools to analyze the random variables and get as much information as possible out of them. The effects that randomness has often create chaos when we try to analyze and predict what will happen in a system, but when it comes to designing algorithms, we sometimes embrace this chaos and we use it in order to make things simpler, faster and more efficient.

Without a doubt, the first thing we are interested in learning about a random variable is its expected value. For instance you cannot be sure how many heads of tails you will have if you flip a fair coin a 100 times, but you expect it to be around 50. You cannot be sure how much time it will take you to drive to your work due to the unpredictable traffic or the fact than an accident may occur. Nevertheless, based on your experience you expect to take around the same time it took you all the previous times. If you are a basketball manager and you are looking for a shooting guard for your team, you will try to get one with high probability of scoring shoots. The expected value may affect our decisions. For instance lets say you have to pick between two shooting guards, one that has 50% chance of scoring a shoot and an other with 60%. You will get the second one. This does not guarantee though that he will score a bigger percentage of basket-shots than the other player, but in the long run and after many games, you expect his true colors to show through.

Although the expected value is a key feature when we either make a decision or designing an algorithm, it is not the only thing we care. We want to know how likely is for the random variable we care to deviate for the expected or mean value. To get a better understanding, lets say you have am important meeting at work today and you do not want to be late. The expected time it takes you to go to work is 40 minutes but you also want to know the probability that you today's trip to work will deviate far from its expectation time. In the head or tails example, after throwing the coin a 100 times you want to know how likely is to get heads more than 60 times.

The deviation of random variable is also important when we design systems and algorithms. For instance, when setting up servers for an internet site, you have an expected traffic value in mind. You will always have some extra servers in case your traffic increases for a short period of time, but it is crucial for you to know what is the probability of passing the threshold you site can serve. While designing a randomized algorithm, we are interested in proving not only that the expected value of the objective function will be low or high, depending on what we are aiming for, but also that it will have a small deviation, meaning that its behavior is good almost all the time. In our problem, we seek to opt for a balanced distribution. Thus we are trying to prove that with high probability, the maximum load of our algorithm will not deviate far from the expected value. The tools we are going to introduce now will help us do exactly that.

3.2 Markov and Chebyshev Inequalities

Before we introduce the Inequalities of Markov and Chebyshev we shall explain what is a tail of a distribution.

The tail of a distribution isn't a precisely defined term. In other words, there is not some specific place where you stop being in the middle of the distribution and start being in the tail. With that said, it is a very important concept nevertheless. The tail basically refers to the part of the distribution that is really far away from the mean. Distributions like the normal have very "skinny" tails because the density decreases like $e^{(-x^2)}$ for $|x|$ much greater than the mean. Other distributions like the Cauchy distribution have very "fat" tails because they decrease much more slowly. The Cauchy decreases like $x^{(-2)}$.

In most of the cases, a lot of problem-specific analysis may be involved in order to find the probability that a random variable will deviate far from its expectation. Often one can avoid such detailed analysis by resorting to general inequalities on tail probabilities such as the ones we are going to present next.

Theorem 3.1. ***Markov Inequality**: Let Y be a random variable assuming only non-negative values. Then for all $t \in \mathbb{R}^+$,*

$$Pr[Y \geq t] \leq \tfrac{E[Y]}{t}.$$

Equivalently

$$Pr[Y \geq kE[Y]] \leq \tfrac{1}{k}.$$

We should note that this is the tightest possible bound when we know only that Y is non-negative and has a given expectation. Unfortunately, the Markov inequality by itself is often too weak to yield useful results. Assuming we have more information about the distribution we will show that we derive better bounds on the tail probability.

Based on the knowledge of the variance of the distribution we can use the Chebyshev inequality to get a stronger bound.

Theorem 3.2. ***Chebyshev Inequality**: Let X be a random variable with expectation μ_X and standard deviation σ_X. Then for all $t \in \mathbb{R}^+$,*

$$Pr[X - \mu_X \geq t\sigma_X] \leq \tfrac{1}{t^2}.$$

Proof. The proof comes directly by applying the Markov inequality for the random variable $Y = (X - \mu_X)^2$ with expectation σ_X^2 □

The Chebyshev Inequality is an improvement to the the simple Markov inequality because we used more information to derive these bounds. There are some applications where these bounds suffice to prove what we are interested in, but most of the time we need something even sharper. For that reason, in the next section, we are going to introduce the Chernoff or exponential bound. We will focus on techniques for obtaining considerably sharper bounds on such tail probabilities. The Chernoff bound, is one of the most useful and thus widely used in algorithms design.

3.3 Chernoff Bound

Before we present it and apply it to some examples in order to get a better understanding, we shall introduce the type of random variables that we are most concerned to analyze and bound. This random variable is the sum of independent Bernoulli trials; for example the outcomes of tosses of a coin. Let $X_1, ..., X_n$ be the independent Bernoulli trials such that, for $1 \leq i \leq n$, $Pr[X_i = 1] = p$ and $Pr[X_i = 0] = 1 - p$. Let $X = \sum_{i=1}^{n} X_i$; then X is said to have the binomial distribution. More generally when for $1 \leq i \leq n$, $Pr[X_i = 1] = p_i$ and $Pr[X_i = 0] = 1 - p_i$ then we referred to these trials as Poisson trials. We will discuss the random variable $X = \sum_{i=1}^{n} X_i$ for the general case, but the result of course apply also for the binomial distribution.

We consider two questions regarding the deviation of X from its expectation $\mu = \sum_{i=1}^{n} p_i$. First, for a real number $\delta > 0$, what is the probability that X exceeds $(1 + \delta)\mu$?. An answer to this type of question is useful in analyzing an algorithm, showing that a chance it fails to achieve a certain performance is small. Second, when designing an algorithm we want to know : how large must δ be in order that the tail probability is less than a prescribed value ϵ?. Both of these questions can receive tight questions by using a technique known as the Chernoff bound.

For a random variable X, the quantity $E[e^{tX}]$ is called the moment generating function of X. This is because $E[e^{tX}]$ can be written as a power-series with terms of the form

$t^k E[X^k]/k$. The basic idea behind the Chernoff bound technique is to take the moment generating function of X and apply the Markov inequality to it. The sum of independent random variables appears in the exponent, and this turns into the product of random variables whose expectation we then bound using inequalities deriving from their McLaurin expansion.

Theorem 3.3. *Let $X_1, ..., X_n$ be the independent Poisson trials such that, for $1 \leq i \leq n$, $Pr[X_i = 1] = p_i$ and $Pr[X_i = 0] = 1 - p_i$ where $0 \leq p_i \leq 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = E[X] = \sum_{i=1}^n p_i$ and any $\delta > 0$.*

$$Pr[X > (1+\delta)\mu] \leq \left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^\mu$$

We will emit the actual proof since it is not correlated with the problem we are going to use this theorem and for the reason that it can easily be found in the bibliography.

For future reference, we will define the right part of the inequality proved in the above theorem as $F^+(\mu, \delta) = \left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^\mu$. This way we can refer to that quantity easier in the examples that follow.

Example 3.1. *The university basketball team win each game they play with probability $1/3$. Assuming that the outcomes of the games are independent, derive an upper bound on the probability that they have a winning season in a season lasting in n games.*

Let X_i be 1 if the university team win the ith game and 0 otherwise; let $Y_n = \sum_{i=1}^n X_i$ Applying the theorem 3.3 to Y_n, we find that $Pr[Y_n > n/2] < F^+(n/3, 1/2) \leq (0.965)^n$. Notice that this bound is not tight but is an upper bound. Nevertheless, we can see that the probability the university team having a winning season in n games is exponentially small in n. This means that the longer they play the more likely it is that their true colors show through.

Notice that the $F^+(\mu, \delta)$ is always strictly less than 1. Since the power μ is always positive, we will always get an upper bound that is less than 1.

We will now present the Chernoff bound for the deviation of X below the its expectation μ.

Theorem 3.4. *Let $X_1, ..., X_n$ be the independent Poisson trials such that, for $1 \leq i \leq n$, $Pr[X_i = 1] = p_i$ and $Pr[X_i = 0] = 1 - p_i$ where $0 \leq p_i \leq 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = E[X] = \sum_{i=1}^n p_i$ and $0 < \delta \leq 1$.*

$$Pr[X > (1-\delta)\mu] \leq e^{-(\mu\delta^2/2)}.$$

The proof of this theorem is similar to the one in 3.3. There reason we get a "cleaner" closed form here is because the expansion for $ln(1-\delta)$ allows us to.

As before, we will define the right part of the inequality as $F^{-}(\mu,\delta) = e^{-(\mu\delta^2/2)}$. As you can see $F^{-}(\mu,\delta)$ is always kess than 1 for possitive μ and δ.

Example 3.2. *The university hires a new coach for the basketball team and critics now revise their estimates of the probability of their winning each game to* 0.75. *What is the probability that they are going to suffer a losing season assuming the critics are right and the outcomes of their games once again are independent of one another?*

Setting up the random variable Y_n *as before, we find that* $Pr[Y_n < n/2] \leq F^{-}(0.75n, 1/2)$, *which evaluates to* $\leq (0.9592)^n$. *Thus, this probability is also exponentially small in n giving them a high probability of having a winning season the longer they play.*

Here we should note that the bounds in theorems 3.3 and 3.4 do not depend on n, but only on μ and δ. This means that they do not distinguish, for instance between 1000 trials each with $p_i = 0.02$ and 100 each with $p_i = 0.2$, even though the distributions of X are different in the two cases. Thus, even if the actual tail probabilities are different in these cases, the estimates we are going to get are the same.

The formula for the $F^{+}(\mu,\delta)$ is often unwieldy in practice. For that reason we sum up the Chernoff bounds we mentioned, with the following looser, but more convenient formula.

$$Pr[X > (1+\delta)\mu] \leq e^{-(\delta^2\mu/3)}, \qquad 0 < \delta < 1$$

$$Pr[X > (1+\delta)\mu] \leq e^{-(\delta\mu/3)}, \qquad 1 < \delta$$

$$Pr[X > (1-\delta)\mu] \leq e^{-(\delta^2\mu/2)}, \qquad 0 < \delta < 1$$

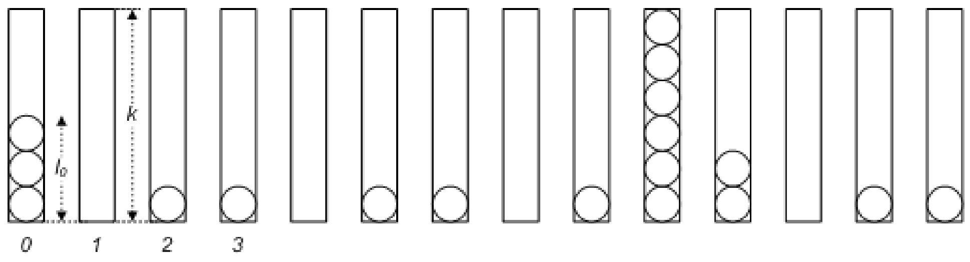
In order to answer the question concerning what must be the value of δ if we want to have a bound ϵ to the probabilty, we need to do the analysis of the bound and solve for δ. We will not do that here since it is not correlated to the problem we are going to solve. We will just introduce a rule of thumb. Usually, we want to have an ϵ of the order of n^{-c} where c is a constant. The value n^{-c} arise often in algorithmic application. We need to have δ of the order of polylogarithm of $\frac{1}{\epsilon}$ providing that μ is $\Omega(logn)$. When μ is smaller, we must do the analysis to obtain the tightest possible estimate.

We are going to apply this rule of thumb to the Balls into Bins problem that we are going to present next and compare the results we are going to get with the tightest possible.

3.4 Balls into Bins

In this section we are going to discuss a problem which is very simple but equally important. We have m indistinguishable objects ("balls") and we throw them randomly into n bins. Each ball is placed in a bin chosen independently and uniformly at random. We are interested in questions such as:

- What is the maximum number of balls in any bin?
- What is the expected number of bins with k balls in them?



Such problems are at the core of the analyses of many randomized algorithms ranging from data structures to routing in parallel computers. Thus, this very simple mode, has also many applications in computer science. Here we name just two of them.

- **Hashing**: The balls into bins model may be used to analyze the efficiency of hashing algorithms. In the case of the so called separate chaining, all keys that hash to the same location in the table are stored in a linked list. It is clear that the lengths of these lists are a measure for the complexity. For a well chosen hash-function (i.e. a hash-function which assigns the keys to all locations in the table with the same probability), the lengths of the lists have exactly the same distribution as the number of balls in a bin.
- **Online Load Balancing**:With the growing importance of parallel and distributed computing the load balancing problem has gained considerable attention during the last decade. A typical application for online load balancing is the following scenario: consider n database-servers and m requests which arise independently at different clients and which may be handled by any server. The problem is to assign the requests to the servers in such a way that all servers handle (about) the same number of requests. Of course, by introducing a central dispatcher one can easily achieve uniform load on the servers. However, within a distributed setting the use of such

a central dispatcher is highly undesired. Instead randomized strategies have been applied very successfully for the development of good and efficient load balancing algorithms. In their simplest version each request is assigned to a server chosen independently and uniformly at random. If all requests are of the same size the maximum load of a server then corresponds exactly to the maximum number of balls in a bin in the balls-into-bins model introduced above.

There are many variations of the problem depending on the number of balls. We will present the results for four different cases . When m is smaller than n, when m is equal to n, when m is equal to $nlogn$ and when m is greater than $nlogn$.

Lets use the Chernoff bound to get an estimation for the case of $m = n$. We throw an n balls into n bins. Let Y be the random variable that denotes the number of balls that fall into the first bin. We are interested in determining the quantity k such that $Pr[Y \geq k] \leq 1/n^2$.

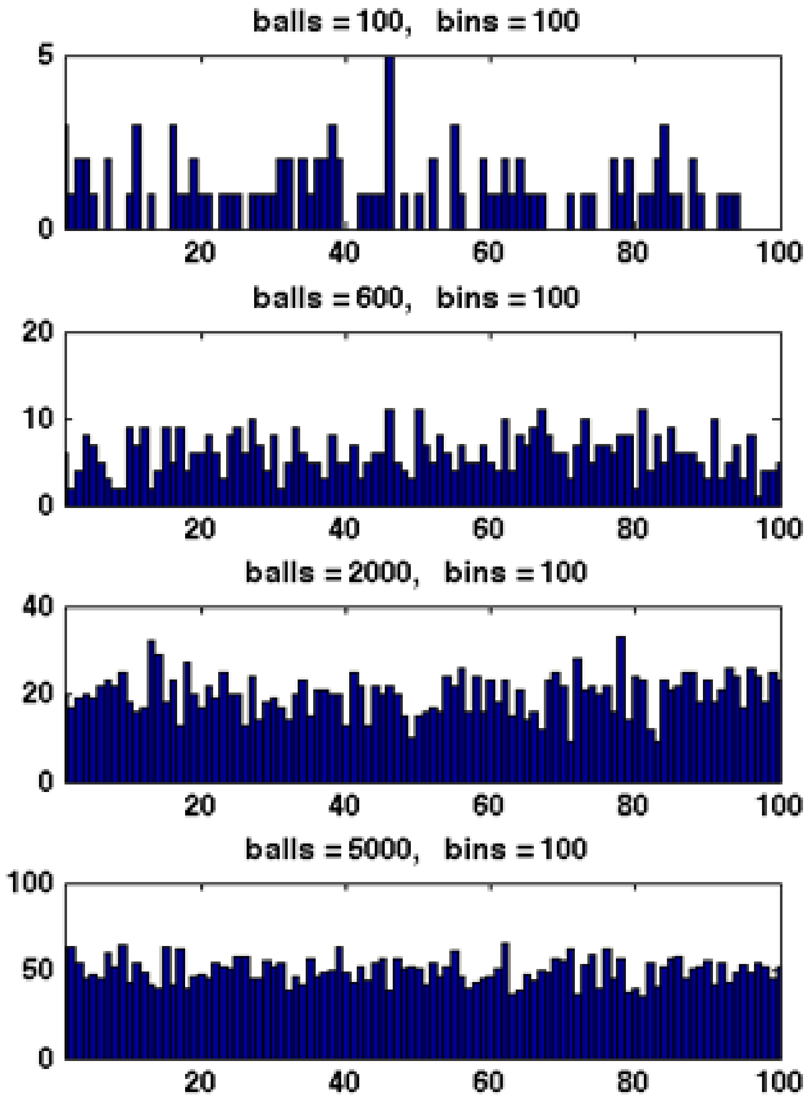
Consider the Bernoulli trials indication whether or not the ith ball falls into the first bin. Each of the p_i is this $1/n$. This means that $\mu = 1$. By using the rule of thumb we get that the valye of m we are looking for is $2logn$. We translate this result as that with high probability, the number of balls in each bin will not exceed $logn$.

Unfortunately, this is the tightest possible answer in this case. As it has been proven by Gonnet [14] the expected value of the maximum load for this case is $\Gamma^{(-1)}(n) - \frac{3}{2} + o(1)$, where $\Gamma^{(-1)}$ is the inverse the Gamma function, and it is know that $\Gamma^{(-1)}(n) = \frac{logn}{loglogn}\big(1 + o(1)\big)$. He also proved that this bound is tight.

We will now present the results for all the cases of m. We will not prove them since there are many proofs in papers like [28] and [26] . We will instead include a figure at the end which illustrates the results of several random experiments showing the distribution of the loads of the bins.

With high probability we get the following bounds:

- Case : $m < n/logn$ the maximum load is $\Theta\big(\frac{logn}{log(n/m)}\big)$
- Case : $m = n$ the maximum load is $O\big(\frac{logn}{loglogn}\big)$
- Case : $m \leq nlogn$ the maximum load is $O(logn)$
- Case : $m > n \cdot polylog(n)$ the maximum load is $O\big(\frac{m}{n}\big)$



From the figure above we can clearly see that the distribution of the loads of the bins becomes more even as the number of balls grow large than the number of bins.

To conclude this chapter we shall present shortly an other variation of the balls in to bins problems. This one is called weighted balls into bins and is a weighted case of the classic problem, where each ball i has a weight w_i. Now the balls are no longer uniform and indistinguishable. We define the load of a bin to be the sum of the weights of the balls allocated to it.

The problem of weighted balls is of practical relevance. Balls-into-bins games are frequently used to conveniently model load balancing problems. Here, weights can be used to model resource requirements of the jobs,i.e.,memory or running time.

One may think intuitively and assume that the bounds will be different. But this is not the case. In [7] the authors compare the maximum load of weighted balls into bins with the maximum load of corresponding the uniform case. They compared the maximum load of a game with m weighted balls with maximum weight 1 and a total weight $W = w_1 + w_2 + ... + w_m$ to the uniform case with approximately $4W$ uniform balls. Basically, they show that the maximum load of the weighted case is not larger than the load of the uniform one.

We should state here that these results are for the case of weighted balls where their individual weight does not exceed a certain value. Otherwise we can think of the extreme case of having only one ball with weight W. Then the expected load of each bin will be W/n but the maximum load would be W because the whole ball has to go into a single bin. We will further study that case and provide a short proof for than in the analysis of the randomized algorithm in chapter 6.

Chapter 4

Problem Definition

From now on we will focus on the main problem of this thesis. We study the problem of distributing the tuples of a relation to a number of processors organized in an r-dimensional hypercube which is an important task for parallel join processing.

In the next section we will explain what is an r-dimensional hypercube and how the algorithm with the same name works. Furthermore, we will give the motivation of this thesis, explaining how solving this problem, will enable us to compute the outcome of conjunctive queries in only one round of communication. After we have explained why we are trying to solve that problem, we will then provide a formal definition of the problem and introduce the work that will follow in the next chapters.

At the end of this Chapter we will explain what is skew in the input data, and how it can affect the lower bounds of our algorithm and thus the optimal distribution we can hope to achieve.

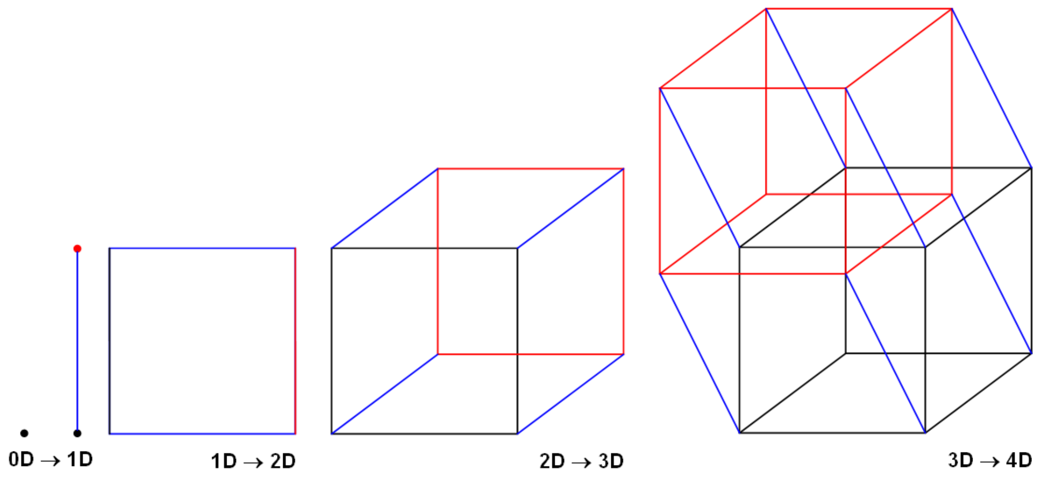
4.1 Hypercube Algorithm

In this section we will present the Hypercube algorithm to compute a conjunctive query in the MPC model in a single round of communication and computation. The main analysis of the algorithm will be done in the chapter 6 but here we will explain the main idea behind it. Before we give a definition we must state that we assume that the input servers know the cardinalities $m_1, ..., m_l$ of the relations $S_1, ..., S_2$ in the body of the query. We denote **m**$= (m_1, ..., m_l)$ the vector of cardinalites and **M**$= (M_1, ..., M_l)$ the vector of the sized expressed in bits, where $M_J = a_j m_j logn, n$ is the size of domain of each attribute, and a_j the arity of each atom.

The question is given the size of the information, how well can an algorithm do in a single communication round? The Hypercube algorithm can achieve the optimal load for any conjunction query q under the assumption that the data we have are without skew. We will not provide a formal definition of skew at this section but intuitively, skew means that some values in the input data appear many times. Finally even in the cases with some skew the algorithm can give near optimal load distribution as we gonna prove in chapter 6.

The Hypercube algorithm was introduced by Afrati and Ulman [2] in a MapReduce setting, and is similar to an algorithm by Suri and Vassilviskii [33] to count the number of triangles in graphs. The idea can be traced much earlier in time, to a work by Ganguly [13] on parallel provessing of Datalog programms. Note that the algorithm can be found in the literature as the Shares algorithm [2]. We will present what the algorithm does and then give a formal definition.

The idea behind it is simple in principle. The Hypercube algorithm performs communication by doing a smart routing of the input tuples. The communication phase in the algorithm is highly distributed, since the destination of each input tuple depends only on the content of the specific tuple, specifically the size M of the relations and the query q that we want to compute. For that reason, it can be easily implemented in almost any distributed or parallel computing environment.



Definition 4.1. The HyperCube Algorithm (HC) . We initially assign to each variable x, where $i = 1, ..., k$, a share p_i, such that $\prod_{i=1}^{k} p_i = p$. Each server is then represented by a distinct point $\mathbf{y} \in P$, where $P = [p_1] \times ... \times [p_k]$. In other words, all the servers are mapped into points of a $k-$dimensional hypercube. Thus the name of the algorithm.

- **Communication:** We use k independently chosen hash functions $h_i : [n] \to [p_i]$ and send each tuple t of relation S_j to all servers in the destination subcube of t:

$$D(t) = \{\ \mathbf{y} \in P | \forall m = 1, ..., a_j : h_{i_m}(t_{i_m}) = \mathbf{y}_{i_m}\ \}$$

- **Computation:** Each server locally computes the query q for the subset of the input that is has received and return its result.

The final result will be the union of the results of all servers.

The reason why the HC algorithm computes the correct result lies in the observation that, for every potential tuple $(a_1, ..., a_k)$ the server $(h_1(a_1), ..., h_k(a_k))$ contains all the necessary information to decide whether it belongs in the answer of not. We must state here that the choice if the size of $p_1, .., p_k$ its crucial and gives a different parametrization of the HC algorithm.

We will not do an extended analysis on the algorithm here, since this will be done in another chapter. We will state two things. First, what is the maximum load per server when we use the algorithm and second, how the algorithm chooses the shares. We will prove that under some skew conditions, and with high probability the maximum load per server is:

$$\tilde{O}\left(max_j \frac{M_j}{\prod_{i:i \in S_j} p_i} \right)$$

This means that we get a poly-logarithmic approximation of the optimal distribution we can have with this formation. As we explained, choosing the best shares is an important factor for a successful distribution of the load over the processors. In their paper [2], Afrati and Ullman, compute the shares by optimizing the total load $\sum_j / \prod_{i:i \in S_j} p_i$ subject to the constraint $\prod_{i:i \in S_j} p_i = p$, which is a non-linear system that can be solved be using Lagrange multipliers. In the paper [6] Beame, Koutris, and Suciu introduce a different approach which is to optimize the maximum load per relation, $L = max_j \frac{M_j}{\prod_{i:i \in S_j} p_i}$ where the total load per server is $\leq lL$. This leads to a linear optimization problems which solution is the shares. We will not write the linear program here and from now on we will consider that the shares are given by this method. What we are going to do is focus on the analysis of the distribution the algorithm does. To get a better understanding lets illustrate how the HC algorithm will compute the triangle query C_3.

4.1.1 HC triangle query example

Lets say we have the following query to compute:

$$C_3(x_1, x_2, x_3) = S_1(x_1, x_2), S_2(x_2, x_3), S_3(x_3, x_1).$$

We will consider the choice of shares given: $p_1 = p_2 = p_3 = p^{1/3}$. Each one of the p servers is uniquely identified by a triple (y_1, y_2, y_3), where $y_1, y_2, y_3 \in [p^{1/3}]$. In the communication round, the input servers that are storing the S_1 send each tuple $S_1(a_1, a_2)$ to all servers with coordination $(h_1(a_1), h_2(a_2), y_3)$ for all $y_3 \in [p^{1/3}]$. At this point we must observe that each tuple is replicated $p^{1/3}$ times since all the coordinates of the missing variable must receive each tuple of the relation. The same procedure happens for the input servers that hold the tuples for the relations S_2 and S_3. At the end of the computation round, any three tuple$S_1(a_1, a_2), S_2(a_2, a_3), S_3(a_3, a_1)$ that are an output of the query will be computed and then returned by the server **y** $= (h_1(a_1), h_2(a_2), h_3(a_3))$.

4.2 Motivation

Our motivation for that is to compute conjunctive queries, using the hypercube algorithm we presented in the previous section. In order to achieve this, we must find a way to distribute not just one relation, but many, in a way that the tuples will satisfy the constrains of the hypercube model. But in most of the cases, the distribution is far from the ideal, because we have to duplicate some tuples of a relation to more than one processor to guarantee that we will get the right result. The Triangle query example in the section above is a case like this. We want to compute a query $C_3(x_1, x_2, x_3) = S_1(x_1, x_2), S_2(x_2, x_3), S_3(x_3, x_1)$. The head of the query has of dimension 3 so we will need a cube. On the other hand, each relation that this query consist of, has only a dimension of 2. This means that we will have to duplicate all the tuples of each relation to the 3^{rd} dimension that this tuple is missing. Queries like these have a structure that forces us to increase the load L of the processors by a factor of $p^{d-v/d}$ where d is the dimension of the head and v is the smallest dimension of a relation in the body of the query. In the case of the triangle query we have to increase the load by a factor of $p^{1/3}$ because each relation in the body has dimension of 2 and the head has dimension of 3. We will not focus on that ”problem” because it is imposed by the model.

To make things worse, finding an efficient way to duplicate the tuples when it is necessary, is not always easy. The randomized algorithm, that we will analyze in chapter 6, uses a single hash function for every dimension and apply that to all the variables of that dimension, even if they belong to different relations. If a relation is missing one dimension, it broadcast its tuples to all the processors of that dimension. This way it satisfies the conditions of the hypercube algorithm and yields near "optimal" distributions at the same time. For the deterministic algorithm, this is not always the case. We have not found a general strategy for the case of d dimension that can satisfy these restrictions. For the case of 1 dimension it is easy since you treat all your relations as if they were the same one with only one variable. Unfortunately, this cannot be done that simply when you have 2 or more dimensions in the hypercube. This constitutes another open problem for the deterministic case.

From now on, and for the rest of the thesis, we will study how to distribute a single relation with dimension r. All the lower bounds and the approximation ratios that we will prove will be for a single relation. Nevertheless, we have to keep in mind that when it comes to distributing more relations, with different dimensions, many problems arise and the maximum load increases significantly.

4.2.1 Problem Statement

The core problem is how to distribute a relation $R(X_1, ..., X_a)$ among p available servers. As we mentioned earlier in this chapter, these servers have been organized in an r-dimensional hypercube with dimensions $p_1, p_2, ..., p_r$, such that $\prod_{i:i \in S_j} p_i = p$. We state again that the choosing of the shares in the hypercube is done by the join algorithm by solving the LP or a non linear system. In either case, we assume that the $p_i's$ are given. The constrain we have is that the distribution of the tuples of R must be performed in such a way that for any two tuples $t, t\prime$, if $t.X_i = t\prime.X_i$, then $t, t\prime$ must be located in servers that have the same $i-$th coordinate. This restriction means that if two tuples agree on an attribute X_i, then the must go to servers that share the same i coordinate. Following that rule for all the variables we get the guarantee that for every assignment that satisfies the query, there will be a server that will return it.

There are multiple ways that we can distribute the tuples such that the above condition is satisfied. Our goal is to find one that minimizes the maximum number of the tuples that each server receives. We call this the maximum load L. The more balanced the load is the smaller the value of L. If the relation R has m tuples and we have p available servers, ideally we would like to achieve the optimal load $L = m/p$. Nevertheless, this is not always feasible. There are cases where the lower bound is above that value. For instance, assume that we have a binary relation $R(X_1, X_2)$ to distribute and only a single value appears at the X_1 attribute of all m tuples. It is easy to see that the problem becomes equivalent to distributing a unary relation $R\prime(X_2)$ over p_2 servers. Consequently, the load L cannot be better than m/p_2 in this case.

In the next chapters we will carefully analyze two kind of algorithms. One that constructs efficient deterministic distributions and one that proposes a randomized distribution strategy. Before we explain these algorithms and compare them, we will present some general lower bounds of the load for any dimension. These bounds do not depend only on the size of the relation, but also on the skew of the data, which is the maximum frequency of each value in the relation.

In the 5^{th} chapter we will present two deterministic algorithms for the case of 1 and 2 dimensions respectfully. The first with an approximation factor of 2 and the second with an approximation ration of $log^2 p$. The 2-dimension algorithm shares an interested connection with the vector load balancing that we mentioned in the 2^{nd} chapter.

In the 6^{th} chapter we will analyze the randomized algorithm that yields an $log^r p$ approximation ratio for the general case of r dimensions under certain skew conditions. Finally, in the case of 2 dimensions we will extend the analysis and provide a proof that this approximation ratio holds with high probability, even for the case where we have high skew.

4.3 Skew

The example above reveals the main reason why finding optimal distribution for every case can be hard.That reason is skew. When some single values or some group of values have a great number of occurrences, beyond the expected number, then designing algorithms to ensure a near optimal distribution is highly challenging. We will give a formal definition to the skew, also know as the degree of values, and then we are going to prove some lower bounds for every algorithm that satisfies the above conditions of the hypercube model. Our goal is to prove that our algorithms yield good approximation ratios to those lower bounds.

Let $R(X_1, ..., X_a)$ be a relation of arity a. Throughout this thesis, we will use m to denote the number of tuples in R and p for the number of servers. We denote by adom(X_i) the active domain of X_i, which is all the possible values that attribute X_i can take in R. For a given value a of attribute X_i, we define the degree of a, denoted $d_i(a)$ as :

$$d_i(a) =\mid \{t \in R \mid t.X_i = a\} \mid$$

More generally, given a tuple $\vec{a}$ over a set of attributes $U \subseteq X_1, ..., X_r$, we define the degree of $\vec{a}$ as :

$$d_U(\vec{a}) =\mid \{t \in R \mid \forall X_i \in U : t.X_i = a.X_i\} \mid$$

We assume that each server s in the hypercube is identified with a point in the r-dimensional space $P = [p_1] \times [p_2] \times ... \times [p_r]$.

Definition 4.2. Distribution Strategy. A distribution strategy D is a mapping from each tuple of R to P, such that for any two tuples $t_1, t_2 \in R$, if for some $i = 1, ..., r$ we have $t_1.X_i = t_2.X_i$ then $D(t_1)[i] = D(t_2)[i]$.

As we said before, this restriction of the distribution strategy is necessary in order for the join queries to be performed correctly like the example in the hypercube algorithm of chapter 2. That way the two tuples from different relations that join on some attribute will meet in a common server. Finally, we use only the first r attributes without any loss of generality.

Given a distribution strategy D, the load of servers, denoted L_s is defined as:

$$L_s =| \{t \in R \mid D(t) = s\} |$$

Definition 4.3. Maximum Load. The maximum load L of a distribution strategy D is defined as the maximum number of tuples that are assigned to any server.

$$L = max_s\{L_s\}$$

We are trying to answer the following question. Given a relation R can we compute an efficient distribution strategy over and r-dimensional hypercube P that achieves the minimum possible load L under the skew or no skew conditions? Before we start presenting the deterministic and the randomized algorithms for that task we will provide some lower bounds that hold for any distribution strategy.

4.4 Lower Bounds

No matter what algorithm we are going to use there are some lower bounds about the best distribution we can hope to achieve. These bounds depends on the size of the relation m, the number of the servers p and as we shower in the previous section, the maximum degrees of each value in the relation.

For any subset of attributes U, let us denote $d_U = max_{\vec{a}} d_U(\vec{a})$. Using only the size of the relation and the number of servers we can come up with an obvious lower bound for the maximum load for any $r \geq 1$ which is $L \geq m/p$. However, as we have seen before, by using the maximum degree information that captures the amount of skew, we can obtain better lower bounds.

Lemma 4.4. *Any distribution strategy on an r-dimensional hypercube with dimensions $p_1, ..., p_r$ requires load*

$$L \geq max\big(\tfrac{m}{p}, \max_{U} \tfrac{d_U \cdot P_U}{p}\big)$$

Proof. Consider any tuple $\vec{a}$ on an attribute set $U \subseteq \{X_1, ..., X_r\}$. Notice that the number of tuples in R that have $\vec{a}$ can be as large as the maximum degree d_U. These d_U tuples have the same coordinates on the U attributes, thus, they can only be distributed across the dimensions of the hypercube in $X_1, .., X_r$". This fact implies that :

$$L \geq \frac{d_u}{\prod_{i:X_i \notin U} p_i} = \frac{d_U \cdot P_U}{p}$$

And since L is always greater or equal to m/p we get that:

$$L \geq max\big(\tfrac{m}{p}, \max_{U} \tfrac{d_U \cdot P_U}{p}\big)$$

□

For the case of one dimension,where we distribute a relation $R(X, ...)$ using only the attribute X, the lemma gives a lower bound $L \geq max(m/p, d_X)$. We will refer to that quantity as $Opt1 = max(m/p, d_X)$. For the case of the 2-dimensional hypercube the distributes the first two attributes of the relation $R(X, Y, ...)$, the above lemma give similarly a lower bound of $L \geq max\big(\frac{m}{p}, \frac{d_X}{p_Y}, \frac{d_Y}{p_X}, d_{XY}\big)$. We will refer to that quantity as $Opt2 = max\big(\frac{m}{p}, \frac{d_X}{p_Y}, \frac{d_Y}{p_X}, d_{XY}\big)$.

Lemma 4.5. *Any distribution strategy on an r-dimensional hypercube with dimensions $p_1, ..., p_r$ requires load $L \geq \frac{m}{min_i pi}$. Moreover, there exist a relation R for which the above lower bound is tight.*

Proof. The first statement is a corollary of the previous lemma. To prove the second statement, assume w.l.o.g that p_1 is the smallest dimension. Then, consider the relation R that contains all tuples of the form $R(c, 1, 1, ..., 1)$, where c takes m different values. It is clear that we can distribute this instance of R only across attribute X_1, which means that the best possible load we can achieve is m/p_1 □

As far as the complexity class of the problem concerns, it is obvious for the case of 1 dimension that the problem is NP-hard.

Lemma 4.6. *Computing the optimal strategy for distributing R along a 1-dimensional hypercube is NP-hard.*

The reduction is clear from the load balancing problem we mentioned at the second chapter.

For the case of two dimension or more, we can show an even stronger result, which is that is not possible to approximate the optimal solution within a constant.

Lemma 4.7. *Unless $NP = ZPP$, for any constant $c > 1$, there is no deterministic polynomial algorithm that approximates the optimal value $Opt2 = max\left(\frac{m}{p}, \frac{d_X}{p_Y}, \frac{d_Y}{p_X}, d_{XY}\right)$ within a factor of c.*

We prove this result by a reduction from the vector load balancing problem.

Chapter 5

Deterministic Load Balancing

In this chapter we will present and analyze two deterministic distribution algorithms that can optimally load balance the input relation. The work presented here is based on the [21]. The first is for the case of 1 dimension and is a fast algorithm called GreedyBalance. This algorithm is optimal within a constant factor 2 of the lower bound. The second is for the case of 2 dimensions and is called 2-Balance. This algorithm yields an $O(log^2 p)$ approximation ratio of the lower bound and it combines the GreedyBalance algorithm with an algorithm for the vector load balancing problem. We remind that for that case there exist no polynomial algorithm that yields a constant approximation factor.

5.1 GreedyBalance

In this section we present the deterministic algorithm GreedyBalance. As we mentioned before, this algorithms achieves a load that is only a constant factor away from the lower bound in the 1-dimension. It will also be useful later, for the algorithm 2-Balance for 2-dimensions. GreedyBalance takes as input a relation $R(X_1, ...)$ of size m, and distributes the tuples along a 1-dimensional hypercube where dimension X_1 has size p.

This greedy algorithm iterates in an arbitrary order through all $a \in adom(X_1)$. For each a, it allocates all tuples with $t.X_1 = a$ to the first X_1-coordinate with load $< m/p$.

Algorithm 1 GreedyBalance

```
$i \leftarrow 1$
for all $a \in adom(X_1)$ do
  if $L_{(i)} \leq m/p$ then
    for all $t \in R$ s.t $t.X_1 = a$ do
      $D(t) \leftarrow (i)$
    end for
  else
    $i \leftarrow i + 1$
  end if
end for
```

Lemma 5.1. ***GreedyBalance** allocates all tuples in R to some processor and achieves maximum load $L \leq \frac{m}{p} + d_{X_1}$.*

Proof. In the algorithm, we keep allocating data until the load of the machine reaches $\frac{m}{p}$ and then proceed onto the next machine. Hence, at any point, before allocating a new set of tuples to a row, the load is $< \frac{m}{p}$. The maximum number of tuples that can be associated with a single $a \in adom(X_1)$ is d_{X_1}. This means that after the tuples are allocated, the load alloted to the processor is bounded by $\frac{m}{p} + d_{X_1}$. $\square$

Since $\frac{m}{p} + d_{X_1} \leq 2max(\frac{m}{p}, d_{X_1}) = 2 * Opt1$, GreedyBalance achieves a load that is a factor 2 away from the lower bound. The example below shows that this factor is essentially tight.

The idea behind this algorithm is simple. You get all the tuples grouped by the variable x and you now see them as a single value with weight equal to their number. Since you cannot always achieve an optimal distribution you start distributing the values to one processor and you stop as soon as you exceed the optimal value of $\frac{m}{p}$. Using that logic you cannot diverge far from the expected value. We will next present the worst case example you can have which proves that the factor of 2 we mentioned before is tight.

5.1.1 Worst Case Example

Consider an instance with the following structure. We have X_1 taking $p + 1$ different values, so $|adom(X_1)| = p + 1$, where the first p values have degree $\frac{m}{p} - \frac{m}{p^2}$ and the last value has degree $\frac{m}{p}$. Observe that the total number of tuples adds up to m and $Opt1 = \frac{m}{p}$. Greedy Balance will achieve a load of at most $2 * Opt1$. According to the pigeon hole principle, since we have $p + 1$ values and p servers, there must be a server that will have at least two of these values. This means that the load cannot be smaller than $2 * (\frac{m}{p} - \frac{m}{p^2}) = 2 * \frac{m}{p}(1 - \frac{1}{p}) = 2 * Opt1(1 - \frac{1}{p})$. Notice that as p grows, this lower bound in the limit becomes $2\frac{m}{p} = 2 * Opt1$ which implies that GreedyBalance is asymptotically optimal.

An other way of thinking this is by having an almost ”full” processor with current load $\frac{m}{p} - 1$ and then you add one last value to that processor that happens to have a $d_X = \frac{m}{p}$. This case is the worst that could happen and the load of that processor would be $2\frac{m}{p} - 1$.

5.2 Vector Load Balancing

In this section, we will introduce an algorithm for the vector load balancing problem called $\Lambda - VLB$. More specifically, we are going to use a restricted version of the online algorithm of [25]. The problem was first introduced by Chekuri and Khanna [10] and later work [3],[17], [25] approached the vector load balancing from an online perspective.

5.2.1 VLB problem correlation

Before we introduce that, let us point out how the VLB is related to the problem we are trying to solve in the 2 dimensions case. Suppose we have a 2-dimensional hypercube and we have already partitioned the tuples of $R(X, Y)$ according to the Y attribute into p_Y coordinates. To complete the distribution strategy, we need to specify how to partition the values in $adom(X)$ to the X coordinates of the hypercube. We can now view each value $j \in adom(X)$ as a job of the VLB and each one of the p_X coordinates as a machine. In this case, the k-th coordinate a_j^k of job j, is the number of tuples of the form $R(j, b, ...)$ for which b is assigned to the k-th Y coordinate. The objective function here is the same. By solving the problem we seek to minimize the makespan which is the maximum load L as described before.

In other words, we use the distribution of one variable to decide the distribution of the other. For instance, when we have to decide in which or the p_X columns one value of X must be distributed, we take into account the distribution of values of Y to decide. We consider different values of Y as being only one if they have been distributed in the same row. The reason why is because the algorithm does not separate the variables Y_1 and Y_2 after the have been distributed in the same row. The tuples (X_1, Y_1) and (X_1, Y_2) are now consider as having to distribute two tuples (X_1, Y_1) according to its X_1 variable. This means that after having distribute the Y variable, the VLB algorithm considers only p_Y

different Y variables. The distribution of those creates the vector of weights for each X value, and these vectors are the ones that the VLB algorithm uses to make the distribution.

Now that the correlation between the two problems is clear, we will define a parameter such that:

$$\Lambda \geq max\left(\max_{n,k} a_j^k \, , \max_k \left(\sum_j a_j^k\right)/n\right)$$

This choice of Λ for the $\Lambda-$VLB algorithm is not arbitrary. In particular, Λ is the lower bound for the best possible load any algorithm can achieve. We will use this bound as an estimation of the optimal load for the algorithm we are going to present next. Let $\beta = (1+\frac{1}{\gamma})^{1/\Lambda}$ for some constant $\gamma > 1$. We use $l_i^k(j)$ to denote the load on component k of machine i once the jobs up to job j have been assigned.

Algorithm 2 Online Algorithm for Λ-VLB

```
$\beta \leftarrow (1+\frac{1}{\gamma})^{1/\Lambda}$
**for all** jobs $j$ **do**
    assign job $j$ to machine $s = \underset{i}{argmin} \sum_{k=1}^{d} \left[\beta^{l_i^k(j-1)+a_j^k} - \beta^{l_i^k(j-1)}\right]$
    **for all** k **do**
        let $l_s^k(j) \leftarrow l_s^k(j-1) + a_j^k$
    **end for**
**end for**
```

Theorem 5.2. *The Online Algorithm for Λ- VLB always achieves a makespan of $\Lambda \cdot O(logn d)$ for the Λ- VLB problem.*

Proof. The proof of the above theorem will be based on a lemma similar to that of [25]

Lemma 5.3. *During any point in the execution of the Λ- VLB algorithm $\sum_{k=1}^{d} beta^{l_i^k} \leq \frac{\gamma}{\gamma+1} nd$.*

Proof. We will provide a proof sketch for that lemma, using a potential function argument. Let $\Lambda^k(j) = \frac{1}{n\Lambda}\sum_{t=1}^{j} a_t^k$. By definition, $\Lambda^k(j) \leq 1$. For each job j, we define the potential function :

$$\Phi(j) = \sum_{k=1}^{d} \sum_{i=1}^{n} \beta^{l_i^k(j)}(\gamma - \Lambda^k(j)).$$

At the beginning, $l_i^k(0)$ is zero, so the value of the potential function is at most γnd. Using one inequality and the way the algorithm chooses the machine s that will allocate the job, it can be proved that $\Phi(j) - \Phi(j-1) \leq 0$. This means that for any j we have that $\Phi(j) \leq \Phi(0) \leq \gamma nd$. □

The previous lemma provides the bound for the performance of the algorithms. Indeed, since $L = \max_{i,k} l_i^k$, we have: $\beta^L = \max_{i,k} \beta^{l_i^k} \leq \frac{\gamma}{\gamma+1} nd$ which implies that

$$L \leq \Lambda \cdot log(\frac{\gamma}{\gamma+1} nd)$$

But $n \cdot d = p_X \cdot p_Y = p$ since n and d are the dimensions of the hypercube, we get :

$$L \leq \Lambda \cdot log(\frac{\gamma}{\gamma+1} p)$$

□

Since we have explained how the Λ-VLB algorithm works given the vector of weights for one of the two dimensions, we now have to figure out a smart way of distributing the first variable to get the near optimal result using the Λ-VLB for the second one. Such a way is provided in the next section that presents the 2-Balance algorithm.

5.2.2 2-Balance

In this section we are going to present the deterministic algorithm called 2-Balance which provide a distribution strategy that yields a polylogarithmic approximation ratio for the case of two dimensions. Since we distributing a relation $R(X, Y, ..)$ across a 2-dimensional hypercube, we can consider a rectangle or a grid with dimension p_X and p_X such as $p_X \cdot p_Y = p$ where p is the total number of processors.

We will denote the maximum degree of attribute X with d_X and the maximum degree of attribute y with d_Y. Two assumptions about the input relation must be made. First, $m \geq p^{3/2}$, in other words the number of tuples is much larger than the number of processors. Second, $d_{XY} = 1$, this means that they attributes X and Y form a key in the input relation because each pair (a,b) for those attributes appear exactly once. It is like assuming that R has the form of $R(X, Y)$ and it is a set with no duplicates. Although the first assumption is natural for modern size databases, the second one creates a problem of generalizing the algorithm into n-dimensions as we are going to mention later.

Let us assume without loss of generality that $p_X \geq p_y$. The algorithm 2-Balance first splits the values in $adom(X)$ into two disjoing subsets:

- **Heavy** (R^H): values for which $d_X(a) \geq p_Y$.
- **Light** (R^L): values for which $d_X(a) < p_Y$.

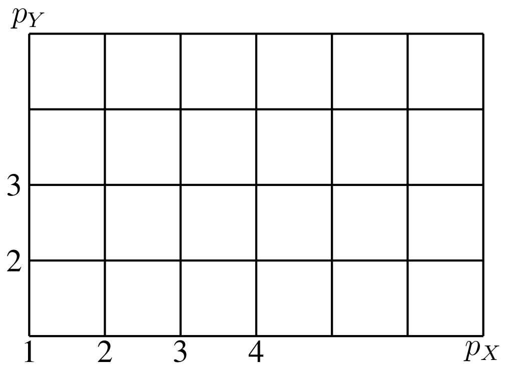
The reason why the we need to split will be obvious after we explain the algorithm, but the idea is that the Heavy and the Light values must be distributed in a different way in order to get satisfactory results. We will present the steps of the algorithm so it would be easier to follow the proofs and understand how they yield the polylogarithmic approximation factor.

1. We apply GreedyBalance to the Heavy values according to the X variable.
2. We apply GreedyBalance to the Light values according to the Y variable.
3. We use the distribution we got from the step 2 in order acquire the values of a_j^k and apply Λ'-VLB with the X values of the R^L as jobs.
4. We use the distribution of X we got from the steps 1 and 3 in order to acquire the values of a_j^k and apply Λ-VLB with Y values of all tuples as jobs. This is the final distribution that the algorithm returns as a result.

Algorithm 3 2-Balance

```
split $X$ to Heavy and Light values.
**for all** $a \in$ Heavy$(X)$ **do**
  apply Greedy Balance
**end for**
**for all** $b \in$ Light$(X)$ **do**
  **for all** $Y \in$ Light$(X)$ **do**
    apply Greedy Balance
  **end for**
  $\forall X \in$ Light$(X)$ get the weights $a_j^k$
  apply $\Lambda$-VLB to the Light $X$ Values
**end for**
$\forall$ $X$ get the weights $a_j^k$
Apply $\Lambda$-VLB to $X$ value.
```

Before we proceed into the analysis of the algorithm we will give the intuition behind it. In the case of 2-dimensions, we have to distribute vectors instead of scalar values. Thus we cannot just use an algorithm that takes into account only one variable and always hope to get near optimal results. We have to consider the relation and the degrees of both X and Y in order to get to a satisfactory distribution. We want to distribute the X in such a way that when it comes to apply the Λ-VLB algorithm the value of Λ, the values of a_i^k and the $\sum_j a_j^k$, would be as low as possible. To achieve this, we break the tuples into two sets and we treat those sets differently. The heavy values cannot be more than $\frac{m}{p_Y}$ by definition so we can apply the GreedyBalance and get satisfactory results. The light values on the other hand can me up to m, thus, before we distribute them we get a "peak" to see how an optimal distribution of Y values would look like. We then use that information to distribute the light X values with VLB. After that is done, we utilize the distribution of all the X attributes to acquire values for the a_j^k and apply Λ-VLB as we present it in the previous section.



5.2.2.1 Distributing Heavy X

As we mentioned before, we apply GreedyBalance to assign each tuple to a specific row based on its X attribute. This way, all tuples sharing the same X value are assigned to the same row. According to the algorithm, the load on each row will be at most $2max(\frac{m}{px}, d_X)$. Moreover:

Lemma 5.4. *If we apply GreedyBalance to distribute R^H according to the X attribute, every value in adom(Y) appears at most $2max(\frac{m}{p}, \frac{d_X}{p_Y})$ times in each row.*

Proof. Since we distribute values a in R^H, for all the $d_X \geq p_Y$.Furthermore, the maximum load for a particular row is $2max(\frac{m}{p}, \frac{d_X}{p_Y})$. This implies that the values from adom(X) that occur in this row can be at most $\frac{2}{p_Y}max(\frac{m}{p}, \frac{d_X}{p_Y})$. Thus, every $b \in adom(Y)$ can appear at most $2max(\frac{m}{p}, \frac{d_X}{p_Y})$ times at any row. □

5.2.2.2 Distributing Light X

As we noted in the previous section we first apply the GreedyBalance according to the Y attribute. This instance can now be seen as a VLB one, where each job j corresponds to a value in $adom(X)$ and each component to a column of the rectangle. In addition, the values a_j^k of a job vector is the number of tuples t such that $t.X = j$ and t is distributed to column k. The bound on the total load of each column guaranteed by GreedyBalance implies that $\forall k, \sum_j a_j^k \leq 2max(\frac{m}{p_Y}, d_Y)$. The following lemma shows that a_j^k will be appropriately bounded as well.

Lemma 5.5. *If we apply GreedyBalance to distribute R^L according to the Y attribute, every value in $adom(X)$ appears at most $2max(\frac{m}{p}, \frac{d_Y}{p_X})$ times in each column.*

Proof. By definition of the light tuples, every value of X appears at most p_Y times. Since $p_y \leq max\{\frac{m}{p}, \frac{d_Y}{p_X}\}$ and $p_X \geq p_Y$ it holds that $p_Y \leq p^{1/2}$. According to our assumption $m \geq p^{3/2}$ and thus $max\{\frac{m}{p}, \frac{d_Y}{p_X}\} \geq \frac{m}{p} \geq p_Y$. □

According to the lemma above, we have that $\forall j, k\ a_j^k \leq 2max\{\frac{m}{p}, \frac{d_Y}{p_X}\}$. The number of machines for the VLB is not $n' = p_X$ and the number of components $d' = p_y$. Consequently, we have an instance of Λ'-VLB problem where $\Lambda' = 2max(\frac{m}{p}, \frac{d_Y}{p_X})$. By running the algorithm for Λ'-VLB to distribute the light X values we obtain that the maximum load will be

$$L = \Lambda' \cdot O(log(n'd')) = max\big(\frac{m}{p}, \frac{d_Y}{p_X}\big) \cdot O(logp).$$

This result implies that after solving the Λ'-VLB problem for the R^L, every value in $adom(Y$ appears at most $max\big(\frac{m}{p}, \frac{d_Y}{p_X}\big) \cdot O(logp)$ times in each row. Also, the total load per row will naturally be at most $max\big(\frac{m}{p_X}, \frac{d_Y}{p_X} \cdot p_Y\big) \cdot O(logp)$.

5.2.2.3 Distributing Y

In the previous steps of the algorithm we have provided a distribution of both the heavy and the light values in $adom(X)$. Now we will just use this distribution and apply VLB along the columns. In this case, each job is a value $adom(Y)$, each component of the job is a row of the rectangles (hence $d = p_X$), and a_j^k is the number of tuples t such that $t.Y = j$ and t is distributed to row k. We have n machines so $n = py$. Let $\Lambda = max\big(\frac{m}{p}, \frac{d_Y}{p_X}\big) \cdot O(logp)$. When we combine the distributions of the heavy and the light values, we can easily see according to our analysis that $a_j^k \leq \Lambda$ and $\sum_j a_j^k \leq \Lambda \cdot p_Y$. We can now apply the Λ-VLB algorithm and obtain that after we run the algorithm and distribute the Y values, the maximum load will be $\Lambda \cdot O(log(nd))$ = $max\big(\frac{m}{p}, \frac{d_Y}{p_X}\big) \cdot O(log^2 p)$.

Combining all the above steps we have proved the following theorem:

Theorem 5.6. *Suppose we have a binary relation $R(X, Y)$ with size $m \geq p^{3/2}$. Then, 2-Balance describes a deterministic distribution strategy that achieves maximum load*

$$L = max\left(\frac{m}{p}, \frac{d_Y}{p_X}, \frac{d_X}{p_Y}\right) \cdot O(log^2 p)$$

Following directly from the above theorem and taking into account that $d_{XY} = 1$ we proved that :

$$L = Opt2 \cdot O(log^2 p)$$

5.3 Conclusion

In this chapter we presented deterministic algorithms to solve the problem of distributing a relation to a number of processors organized in an r-dimensional hypercube. Progress was made providing with efficient algorithms for the case of 1 and 2 dimensions. Studying those problems we came up with several open questions. For example, can we give a more general algorithm that does not require assumption on the degrees of the relation $R(X, Y)$? Furthermore, can we extend our techniques to construct an algorithm that works for dimensions $\geq 3?$. Finally, although these algorithms are very efficient, they still require centralized decisions in order to distribute data. This raises the challenge of designing deterministic strategies that work well in a distributed setting, where the coordination must be minimal.

In the next chapter, we are going to study a randomized algorithm that gives answers to the problems like generalization to more dimensions, dependencies from the degrees and the decentralization.

Chapter 6

Randomized Load Balancing

In this chapter we will study and analyze a simple randomized distribution. As it became clear from the previous chapter the deterministic algorithms for distributing relations, although they can always guarantee a good approximation ratio, they have some major disadvantages. First of all it is hard to generalize them for more than 2 dimensions. Second they need a great level of coordination to be executed which is a major drawback in distributed settings and finally, the are not skew resilient. Trying to deal with all those problems while still demanding efficiency good approximation ratios we proposed the Hypercube algorithm. Since it is a randomized algorithm we can not always be absolutely sure that the result will always be satisfactory. On the other hand, we can prove that with high probability this is going to be the case, which is acceptable and thus implementable for modern businesses and database applications.

We will first shortly present the algorithm again and the intuition behind why it is efficient and preferable to others so far. Next we are going to discuss the previous work and analyses that has been done to the algorithm up until now. The approximation ratios and the effect of skew on them will be presented and then we will explain what are the challenges and the difficulties of generalizing this to be a load-optimal one round algorithm for any input. The work is based on the [5] and [6].Finally, we are going to present the contributions of this thesis to the algorithm. More specifically, in an attempt of generalizing in for any input we will focus on distributing a general 2 dimensional relation, without skew restriction, and prove that the algorithm is optimal in any case.

We will now present a table that sums up the results of this Chapter. This table includes both previous results and the ones that are first shown in this thesis.

TABLE 6.1: HyperCube algorithm approximations depending on skew

HyperCube Algorithm	
Skew and dimension	**Approximation ratio**
Without skew dimension d	$O(log^d p)$
With skew dimension 1	$O(logp)$
With skew dimension 2	$O(log^2 p)$

6.1 HyperCube without skew

We have introduced the Hypercube algorithm in the forth chapter. The strategy of the algorithm picks independently of each $i = 1, ..., m$ a perfectly random hash function $h_i : adom(X_i) \rightarrow 1, ..., p_i$. Then, we send each tuple $R(a_1, ..., ar, ...)$ to the server with coordinates specified by the hash function: $(h_1(a_1), ...h_r(a_r))$. In other words. For a d dimensional hypercube, we define d perfectly random hash functions and we hash each coordinate of the tuple to decide to which server on the hypercube this tuple will be distributed. In this section we will present and prove the approximation ratio that this algorithm achieves, explain the correlation with the balls into bins problem and get a good intuition what are the problems created by skew.

Theorem 6.1. *Let $R(A_1, ...A_r)$ be a relation of arity r of size m. Let $p_1, ..., p_r$ be integers and let $p = \prod_i p_i$ the total number of servers mapped into points of a $r-$dimensional hypercube. Suppose that we hash each tuple $(a_1, ..., a_r)$ to the server with coordinates $(h_1(a_1), ...h_r(a_r))$, where $h_1, ...h_r$ are independent and perfectly random hash functions from the domain n to $p_1, ...p_r$ respectively. Suppose further ,that for every tuple J over $U \subseteq [r]$, we have $d_J(R) \leq \frac{m}{a^{|U|}\prod_{i \in U} p_i}$ for some constant $a > 0$. Then the probability that the maximum load exceeds $\tilde{O}(m/p)$ is exponentially small in p.*

The notation $\tilde{O}(m/p)$ hides $logp$ factors

According to the theorem above, for input tuples that have less skew than the expected load of every server times some constant, this randomized distribution achieves maximum load $O(\frac{m}{p} log^r(p))$ with high probability. This means that this algorithm will give an almost optimal load (up to a polylogarithmic factor) if the degrees of each value or combination of values is small enough.

Before we begin the analysis of the algorithm and the proof of the theorem 6.1, we will introduce an example to understand why skew plays such a crucial role to the maximum load.

Example 6.1. *Suppose we want to compute a simple join query $q(x, y, z) = S_1(x, z)S_2(z, y)$ where both relations have cardinality m. The optimal share are $p_1 = p_2 = 1$, and $p_3 = p$.*

This allocation of shares reduce the cube to a standard parallel hash-join algorithm, where both relations are hashed on the join variable z. When the data has no skew,(in particular, when the frequency of each variable z in both S_1 and S_2 is at most $\frac{m}{p}$), then according to the theorem 6.1, the maximum load is $O(\frac{m}{p}logp)$

However, under the presence of skew, the maximum load can be as large as $O(m)$. Indeed, consider the case where relation S_1 is of the form $(1, 1), (2, 1, ...(m, 1)$. In other words, the frequency of value 1 in S_1 is m. In that case, it is clear that the HC algorithm would hive a maximum load of $O(m)$.

We should note here that the lower bounds we proved in the chapter 5 still hold. This means that for the case where the lower bound is dominated by the term $O(\frac{m}{p})$, as it is when the skew condition we mention in the theorem holds, then the result is near optimal comparing to the lower bound. If the skew condition does not hold, then we have to compare the distribution of the algorithm with the domination term of the lower bound.

The polylogarithmic approximation ration achieved by the algorithm for the cases when skew is bellow the threshold of theorem 6.1 is very satisfactory. Naturally, we are interested in learning what distributions this algorithm yields for the cases that exist values with degrees the surpass this threshold. As we have proved in the previous chapter, you cannot achieve the expected load in these cases, because the lower bounds are depended upon the degrees as well. Thus, when having values with degrees higher that the threshold, we want to know what is the approximation ratio this algorithm achieves.

In this thesis, for the case of 1 and 2 dimensions, we extend the analysis and prove that you can have an polylogarithmic approximation for any input, compared to the lower bounds. In the next sections we give the proofs and aggregate the results into one.

We are interested in learning what will happen in the case of 1 dimension, when the relation $R(X)$ we want to distribute have skew, meaning that there some values of X with degrees $d_X > m/p$. To answer that question we will study the Weighted Balls into Bins problem we presented in chapter 3 and then make a reduction to our problem.

Theorem 6.2 (Light Weighted Balls into Bins)**.** *Let S be a set where each element i has weight w_i and $\sum_{i \in S} w_i \leq m$. Let $K > 0$ be an integer. Suppose that for some $\beta > 0$, $\max_{i \in S} w_i \leq \beta m/K$. We hash-partition S into K bins. Then for any $\delta > 0$,*

$$\Pr\left(L_1 \geq (1+\delta)m/K\right) \leq K \cdot e^{-h(\delta)/\beta}$$

where L_1 is the maximum weight of any bin and $h(x) = (1+x)\ln(1+x) - x$.

The above theorem proves that, in the case where every ball has weight less than $\beta m/K$, $\beta > 0$, then with high probability the maximum load of any bin would be O($\frac{m}{K} \log K$). In that case, it is clear that the optimal distribution we hope to achieve is $\frac{m}{N}$ so the hashing algorithm is an $O(\log K)$ approximation algorithm.

Before we proceed in proving the theorem above, we shall explain how this exponential bound leads to a $O(\log K)$ approximation ratio to the load. This is happening either by allowing the load to grow to $m/Kln(K)$ or by requiring the maximum degree to be $< m/(K \cdot ln(K))$.

Let us denote the maximum weight of a ball d. Since $d \le \beta m/K$ we can state that $\beta \ge \frac{m}{K\beta}$. We can now rewrite the inequality like this:

$$\text{Pr}\left(L_1 \ge (1+\delta)m/K\right) \le K \cdot e^{-\frac{m}{Kd}h(\delta)}$$

Now we get the following analysis for different values of d.

1. If $d = 1$(the degree of every value is 1), then for any $\delta < 1$,

   $$Pr\left(L_1 \ge (1+\delta)m/K\right) \le K \cdot e^{-\frac{m\delta^2}{3K}}$$

   this probability decreases exponentially fast in the input size m.

2. If $d \le m/(3Kln(K))$, then for any $\delta > 1$, Pr $\left(L_1 \ge (1+\delta)m/K\right) \le K^{1-\delta}$; this probability decreases polynomially in the number of bins K.

3. If $d \le m/K$, Pr $\left(L_1 \ge (1+\delta)m/K\right) \le K^{-1}$; this probability also decreases polynomially in the number of bins K.

Proof. Item (1) follows from the fact that for $\delta < 1$ we have that $h(\delta) \ge \delta^2/3$. Item (2) follows from $h(\delta) \ge \delta/3$ and $Ke^{-ln(K)\delta} = K^{1-\delta}$. Item (3) follows from setting $1 + \delta = lnp$. Then since $(1+\delta)ln(1+\delta) - \delta > 2(1+\delta)$ whenever $ln(1+\delta) > 3$ we have $Ke^{-2ln(K)} = K^{-1}$.

□

We will show the proofs for the theorems 6.2 and 6.1 which are the basic results of this section. As a matter of fact we will first prove a stronger version of theorem 6.2 where we replace $h(\delta) by K \cdot D\left(\frac{1+\delta}{K}||\frac{1}{K}\right) where D(q'||q) = q'ln\left(\frac{q'}{q}\right)ln\left(\frac{1-q'}{1-q}\right)$ is the relative entropy (also know as KL-divergence) of Bernoulli indicator variables with probabilities q' and q. This strengthening is immediate from the following theorem with $t = 1 + \delta$ and $m_1 = m$, which implies the bound for a single bin, together with a union bound over all K of bins. The statement of theorem 6.2 follows from a weaker form of the next theorem that can also be derived using Bennett's inequality. We will follow the proof that can be found at [6].

Theorem 6.3. *Let $K \geq 2$ and $w \in \mathbb{R}^n$ such as $w \geq 0, ||w||_1 \leq m_1$, and $||w||_\infty \leq m_\infty = \beta m_1/K$. Let $(Y_i)_{i\in[n]}$ be a vector of i.i.d. random indicator variables with $Pr(Y_i = 1) = 1/K$ and $P(Y_i = 0) = 1 - 1/K$. Then*

$$P\Big(\sum_{i\in[n]} w_i Y_i > tm_1/K\Big) \leq e^{-K\cdot D(t/K||1/K)/\beta}$$

Proof. The proof follows along similar lines to constructive proofs of Chernoff bounds in [18] : Choose a random $S \subseteq [n]$ by including each element i independently with q_i :

$$Pr(i \in S) = q_i = 1 - (1-q)^{\frac{w_i}{\beta m_1}/K}$$

Let ε denote the event that $\sum\limits_{i\in[n]} w_i Y_i \geq tm_1/k$. Then :

$$\mathbb{E}\Big[\bigwedge_{i\in S} Y_i = 1\Big] \geq \mathbb{E}\Big[\bigwedge_{i\in S} Y_i = 1|\varepsilon\Big] \cdot Pr(\varepsilon)$$

We bound both expectation. First we see that

$$\begin{aligned}
\mathbb{E}\Big[\bigwedge_{i\in S} Y_i = 1\Big] &= \sum_{S\subseteq[n]} \big(1/k^{|S|}\big) \prod_{i\in S} q_i \prod_{i\notin S} \big(1-q_i\big) \\
&= \sum_{S\subseteq[n]} \prod_{i\in S} \big(q_i/K\big) \prod_{i\notin S} \big(1-q_i\big) \\
&= \prod_{i\in[n]} \big(q_i/K + (1-q_i)\big) = \prod_{i\in[n]} \big(1/K + (1-1/K)(1-q_i)\big) \\
&= \prod_{i\in[n]} \big(1/K + (1-1/K)(1-q)^{\frac{w_i}{\beta m_1/K}}\big) \\
&\leq \prod_{i\in[K/\beta]} \big(1/K + (1-1/K)(1-q)\big) \\
&= \big(1 - K + (1-1/K)(1-q)\big)^{K/\beta}. \\
&= \big(1 - q(1-1/K)\big)^{K/\beta}.
\end{aligned}$$

The inequality follows from the fact that the function $f(w) = 1/K + (1-1/K)(1-q)^{\frac{w_i}{\beta m_1/K}}$ is logconvex and therefore $\prod_i f(w_i)$ is maximized on a vertex of the polytope given by $0 \leq w_i \leq \beta m_1/K$ and $\sum_i w_i \leq m_1$.

Second for any outcome of $Y_1, ..., Y_n$ that satisfies ε, the probability that S misses all indices i such that $Y_i = 0$ is

$$\prod_{i:Y_i=0}(1-q)^{\frac{w_i}{\beta m_1/K}} = (1-q)^{\frac{\sum_{i:Y_i=0} w_i}{\beta m_1/K}} \geq (1-q)^{\frac{m_1 - tm_1/K}{\beta m_1/K}}$$

since ε implies that $\sum_{i:Y_i=0} \leq m_1 - \sum_{i:Y_i=1} w_i \leq m_1 - tm_1/K$. Hence

$$\mathbb{E}\left[\bigwedge_{i\in S} Y_i = 1|\varepsilon\right] \geq (1-q)^{(K-t)/\beta}$$

Combining the last two inequalities we get that:

$$P(\varepsilon) \leq \left(\frac{1-q(1-1/K)}{(1-q)^{1-t/K}}^{K/\beta}\right)$$

As noted in [18], by looking at its first derivative, one can show that the function $f_{\delta,\gamma}(q) = \frac{1-q(1-\delta)}{1-q)^{1-\gamma}}$ takes its minimum at $q = q* = \frac{\gamma-\delta}{\gamma(1-\delta)}$ where it has value $e^{D(\gamma||\delta)}$. Plugging in $\delta = 1/K$ and $\gamma = t/K$,we obtain:

$$P(\varepsilon) \leq e^{-K\cdot D(t/K||1/K)/\beta}$$

□

We will now introduce a lemma that we are going to need for the proof of the theorem 6.1. We will not prove this lemma. For readers interested in the proof we suggest they look at the appendix of [6].

Lemma 6.4. *Let* $K \geq 2$ *and* $w_j^{(j)}$ *be a sequence of vectors in* $\mathbb{R}^n$ *satisfying* $w^(j) \geq 0$, $||w^{(j)}||_1 \leq m_1$ *and* $||w^{(j)}||_\infty \leq m_\infty = \beta m_1/K$. *Suppose further that* $||\sum_j w^{(j)}||_1 \leq km_1$. *Let* $(Y_i)_{i\in[n]}$ *be a vector of i.i.d. random indicator variables with* $Pr(Y_i = 1) = 1/K$ *and* $Pr(Y_i = 0) = 1 - 1/K$. *Then*

$$Pr\big(\sum_{i\in[n]} w_i^{(j)} Y_i > (1+\delta)\tfrac{m_1}{K}\big) \leq 2k \cdot e^{-h(\delta)/\beta}$$

6.1.1 Partition with Promise

Before we begin the proof of theorem 6.1, some notation is need. Even though in this thesis we assume the relations are sets, here we will give a more general analysis for bags.

Let a U-tuple J be a function $J : U \to [n]^{|u|}$, where $[n]$ is the domain and $U \subseteq [r]$ a set of attributes. If J is a V-tuple and $U \subseteq V$ then $\pi_U(J)$ is the projection of J on U. Let S be a bag or $[r]$-tuples. Define:

- $m(S) = |S|$ the size of the bag S, counting duplicates

- $\Pi_U(S) = \{\pi_U(J) \mid J \in S\}$ duplicates are kept, thus $|\Pi_U(S)| = |S|$
- $\sigma_J(S) = \{K \in S \mid \pi_U(K) = J\}$ bag of tuples that contain J
- $d_J(S) = |\sigma_J(S)|$ the degree of the tuple J

Theorem 6.5. *Let S be a bag of tuples of $[n]^r$, and suppose that for every U-tuple J we have $d_J \leq \frac{\beta^{|U|} \cdot m}{p_U}$ where $\beta > 0$. Consider a hypercube hash-partitioning of S into p bins. Then, for any $\delta \geq 0$:*

$$Pr\left(L > (1+\delta)^r \frac{m(S)}{p}\right) \leq f(p, r, \beta) \cdot e^{-h(\delta)/\beta}$$

where the bings refer to the HyperCube partition of S using shares $p_1, .., p_r$ and

$$f(p, r, \beta) = 2p \sum_{j=1}^{r} \prod_{u \in [j-1]} (1/\beta + 1/p_u) \leq 2p \frac{(1/\beta + \epsilon)^r - 1}{1/\beta + \epsilon - 1}$$

where $\epsilon = 1/min_{u \in [r-1] p_u}$.

Proof. We prove the theorem by induction on r. The base case $r = 1$ follows immediately from Theorem 6.2 since an empty product evaluates to 1 and hence $f(p, 1, \beta) = 2p$. Suppose that $r > 1$. There is one bin for each r-tuple in $[p_1] \times ... \times [p_r]$. We analyze cases based on the value $b \in [p_r]$. Define

$$S^{r \to b}(h_r) = \bigcup_{i \in [n] : h_r(i) = b} \sigma_{r \to i}(S)$$

and

$$S'(b, h_r) = \Pi_{[r-1]}(S^{r \to b})$$

Here $r \to i$ denotes the tuple (i). $S^{r \to b}(h_r)$ is a random variable depending on the choice of the hash function h_r that represents the bag off tuples sent to bins whose first projection is b. $S'(b, h_r)$ is is essentially the same bag where we drop the last coordinate, which, strictly speaking, we need to do to apply inductio. Then $m(S'(b, h_r)) = m(S^{r \to b}(h_r))$.

Since the promise with $U = \{r\}$ implies that $d_r(S) \leq \beta \cdot m/p_r$, by 6.2

$$P(m(S'(b, h_r) > 1 + \delta m(S)/p_r) \leq e^{-h(\delta)/\beta}$$

We handle the bins corresponding to each value of b separately via induction. However, in order to do this we need to argue that the recursive version of the promise on coordinates holds for every $U \subseteq [r-1]$ with $S'(b, h_r)$ and $m' = (1+\delta)m(S)/p_r$ instead if S and m. More precisely, we need to argue that, with high probability, for every $U \subseteq [r-1]$ and every U-tuple J,

$$d_J(S'(b,h_r)) \leq \frac{\beta^{|U|} \cdot m'}{p_U} = (1+\delta)\frac{\beta^{|U|} m}{p_U p_r}$$

Fix such a subset $U \subseteq [r-1]$. The case for $U = \emptyset$ is precisely the bound for the size $m(S'(b,h_r))$ of $S'(b,h_r)$. Since the promise of the theorem statement with $U = \{r\}$ implies that $d_{\{r\}}(S) \leq \beta m/p_r$, by theorem 6.2, we have that

$$P(m(S'(b)) > m') \leq e^{-h(\delta/\beta}.$$

Assume next that $U \neq \emptyset$.Observe that $d_J(S'(b,h_r))$ is precisely the number of tuples of S consistent with (J,i) such that $h_r(i) = b$. Using the lemma 6.4 we upper bound the probability that there is some U-tuple J such that the above inequality fails. We apply the lemma with $k = k(U)$, $m_1 = m/K(U)$ and $m_\infty = \beta m_1/p_r$ to say that the probability that there is some U-tuple J such that $d_J(S'(b)) > (1+\delta)m_1/p_r = (1+\delta)m/p_r k(U)$ is at most $2k(U) \cdot e^{-h(\delta)/\beta}$. It is easy to see that the conditions to apply the lemma hold.

For a fixed b, we now use a union bound over the possible set $U \subseteq [r-1]$ to obtain a total probability that the inequality fails for some set U and some$U$-tuple J of at most

$$2 \sum_{U \subseteq [r-1]} \beta^{-|U|} p_U \cdot e^{-h(\delta)/\beta}$$
$$= 2(p/p_r) \prod_{U \subseteq [r-1]} (1/\beta + 1/p_u) \cdot e^{-h(\delta)/\beta}$$

If $m(S'(b)) \leq m'$ and the inequality above holds for all $U \subseteq [r-1]$ and U-tuples J, then we apply the induction hypothesis to derive that the probability that some bin that has b in its last coordinate has more than $(1+\delta)^r m/p$ tuples is at most $f(p/p_r, r-1, \beta) \cdot e^{h(\delta)/\beta}$.

Since there are p_r choices for b, we obtain a total failure probability at most $f(p,r,\beta) \cdot e^{h(\delta)/\beta}$ because from the union bound we get that this probability is less than the sum of the two failure probabilities. We will omit the exponential part of the probabilities and focus on proving the factor $f(p,r,\beta)$

$$Pr(failure) \leq \left(2p_{[r-1]} \prod_{u \in [r-1]} (1/\beta + 1/p_u) + f(p/p_r, r-1, \beta)\right)$$
$$= 2p \prod_{u \in [r-1]} (1/\beta + 1/p_u) + p_r f(p/p_r, r-1, \beta)$$

$$
\begin{aligned}
&= 2p \prod_{u\in[r-1]} (1/\beta + 1/p_u) + p_r(2p/p_r) \sum_{j=1}^{r-1} \prod_{u\in[j-1]} (1/\beta + 1/p_u) \\
&= 2p \sum_{j=1}^{r} \prod_{u\in[j-1]} (1/\beta + 1/p_u) \\
&= f(p, r, \beta)
\end{aligned}
$$

$\square$

6.2 Hypercube with skew

The results that we have presented for the case without skew are very satisfactory and near optimal. This raises the question what will happen to this algorithm if we have skew. As we have pointed out many times to this thesis, it is no longer possible to achieve the same efficient distributions since the lower bounds are different for the case of skew. This however does not mean that we cannot get near optimal results. This time the term optimal does not refer to the expected value $\frac{m}{p}$ of course but to the term we defined as $Opt2$. This means that we are not trying to approximate the lower bounds and get near optimal results compared to them. In this section we prove just that. This is the main contribution of this thesis. We first provide a proof for the case of 1 dimension and then we provide a proof for the case of 2 dimensions. We wish to generalize these proofs for the case of n dimensions where $n \geq 3$ but this work is still in progress and will be discussed in the conclusion.

Now we can begin presenting the proofs for the case with skew. First we present our proof for a theorem that can be found as weighted balls into bins in the bibliography. The proof is very natural and intuitive from the balls into bins problem we presented in the 3^{rd} chapter. We later proceed with the case of 2 dimensions where the things start to get a little more complicating.

6.2.1 1 dimension

Assume there exist a ball with weight more than $\frac{m}{N}$ then we cannot apply the theorem 6.2. Let W_{max} be the maximum weight. Since $W_{max} > \frac{m}{K}$ and that ball has to be put into one bin, the maximum load of all the bins cannot be less than W_{max}. According to the theorem we are going to prove next, in that case, the maximum load of any bin using the hashing algorithm would be $O(W_{max} \log N)$ with high probability.

Theorem 6.6 (Heavy Weighted Balls into Bins)**.** *Let S be a set where each element i has weight w_i and $\sum_{i \in S} w_i \leq m$. Let $K > 0$ be an integer. Suppose that $\forall i \in S$ $w_i \leq W_{max}$ where $W_{max} > \frac{m}{K}$. We hash-partition S into K bins. Then for any $\delta > 0$*

$$\Pr\big(L_1 \geq (1+\delta)W_{max}\big) \leq K \cdot e^{-h(\delta)}$$

Proof. The worst case scenario for the Load of the bins according to [22] is when all the weights have value equal to the maximum weight W_{max}. So we will prove that the argument holds for that case and thus for all the cases. Since all the balls have weight more than $W_{max} > \frac{m}{K}$ and the sum of the weights is m, there cannot be more than K balls. Since we have less than K balls to distribute into K bins, we can apply the result from the classic balls into bins problem, and get that the maximum number N of balls into every bin is $O(\log K)$ with high probability. To prove our argument we can apply Theorem 6.2 with $\beta = 1$ and $m = N$ and $w_i = 1$ $\forall i \in S$. Then for any $\delta > 0$

$$\Pr\big(N \geq (1+\delta)N/K\big) \leq K \cdot e^{-h(\delta)}$$

Because $K \geq N$ we can substitute N/K with 1. Furthermore $L_1 = N \cdot W_{max}$ because all the balls have the same weight. So we multiply both parts of the inequality with W_{max} and we get the final result.

$$\Pr\big(L_1 \geq (1+\delta)W_{max}\big) \leq K \cdot e^{-h(\delta)}$$

□

To conclude, let Opt be the optimal distribution we can hope for. In the general case $Opt = max(W_{max}, \frac{m}{K})$. Using the hash-partition algorithm we proved with Theorem 6.2 and Theorem 6.6 that with high probability we have an $O(\log K)$ approximation to the optimal solution. To be exact, for any $\delta > 0$

$$\Pr\big(L_1 \geq (1+\delta)Opt\big) \leq K \cdot e^{-h(\delta)}$$

We can reduce the tuples distribution problem of one dimension to balls into bins. The reductions is obvious. Each different tuple can be considered as a ball with weight equal to the number of times the tuple appears. The total number of tuples is m. For the one dimension case, lets assume that we assign values to X and instead of K bins we have p processors that we want to distribute those values. Using this reduction we can conclude that the hashing-algorithm for distributing these tuples to the p processors will give an $O(\log p)$ approximation ratio.

Having proved the case of 1 dimension with skew we will proceed to expand the proof for 2 dimensions using the theorems above that we have just shown.

6.2.2 2 dimensions

We will now study the case of 2 dimensions. This time we will focus directly to our problem and trying to figure out what is happening for the case of skew in 2 dimensions. As we mentioned again, we will provide a novel result, that the hypercube algorithm is optimal within a polylogarithmic factor even for the case of skew in 2 dimensions. Note that the optimality is always compared to the lower bound we proved in the chapter 5.

Assume we have a relation $R(X, Y)$ with m tuples. Let d_X be the maximum degree of the X variable, d_Y the maximum degree of the Y variable and d_{XY} the maximum degree of tuples that has same values in both variables. We have p processors again and we organize them in a 2-dimensional grid with p_X columns and p_Y rows. Thus is obvious that $p = p_X \cdot p_Y$. Finally, let L_2 be the maximum load of any processor after distributing all the tuples. What we want to prove is that the algorithm that is hashing each variable independently, yields a poly logarithmic approximation ratio with high probability. The algorithm is hashing each tuple to one of the p_X columns according to its X variable and to one of p_Y rows according to its Y variable. As we mentioned before, the optimal distribution we can hope for for the 2 dimensions case is the $\max(\frac{m}{p}, \frac{d_X}{p_Y}, \frac{d_Y}{p_X}, d_{XY})$, which we will call $Opt2$. So basically $L_2 \geq \max(\frac{m}{p}, \frac{d_X}{p_Y}, \frac{d_Y}{p_X}, d_{XY})$. The reason why was explained in the previous chapter in the section of lower bounds. The next Theorem is an extensive proof that proves the poly-logarithmic bound for all the possible degree cases where we have a Set, meaning $d_{XY} = 1$.

Theorem 6.7. *Let S be a set of m tuples with 2 dimensions each. Let d_X and d_Y be the degrees of variables X and Y and p_X and p_Y the dimensions of the grid of processors such as $p = p_X \cdot p_Y$. Because it is a set we have $d_{XY} = 1$. Consider a hash-partition of S into those p processors. Then, for any $\delta \geq 0$:*

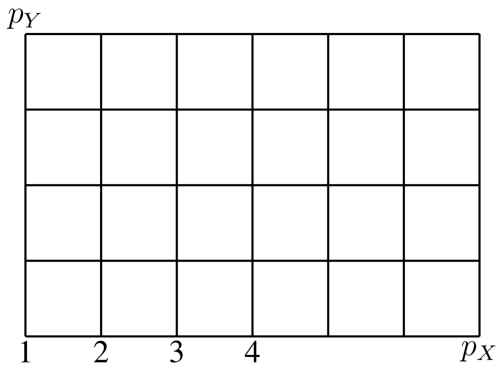
$$\Pr\left(L_2 > (1+\delta)^2 Opt2\right) \leq 8p \cdot e^{-h(\delta)}$$

Proof. Let us assume w.l.o.g that $\frac{d_X}{p_Y} > \frac{d_Y}{p_X}$, where d_X is the maximum degree of X and d_Y of Y. We will call each variable i of X heavy, if $d_{Xi} > \frac{m}{p_X}$ and light if $d_{Xi} \leq \frac{m}{p_X}$. The same goes for Y, we call heavy Y those with $d_{Yi} > \frac{m}{p_Y}$ and light Y those with $d_Y \leq \frac{m}{p_Y}$. We will first prove that those tuples that have both variables being light are well distributed with high probability. Then we are going to prove the same for tuples that have at least one of the variables being heavy. Finally, doing a Union Bound we will get our result.

Case i (light): We only have light values of X and Y. That case is proven by 6.5. For $r = 2$ and $b = 1$ we get that

$$\Pr\left(L_2 > (1+\delta)^2 \cdot \frac{m}{p}\right) \leq 5p \cdot e^{-h(\delta)}$$

where $\frac{m}{p}$ is the dominating term in that scenario and thus is equal to the $Opt2$.



Case ii(heavy) : Now, we have tuples that have at least one heavy variable. Lets assume that we have to distribute tuples that have heavy X values. The key observation here is that we can only have a small number of different heavy values. For instance, the number of heavy values in X cannot be more than p_X by definition because $d_X > \frac{m}{p_X}$. Combining that with the fact that we work with a set and we have $d_{XY} = 1$ we get to the conclusion that the maximum degree of Y cannot be higher than p_X. Now, we can think of doing the hashing of the X variable into p_X processors and then each processor hashing according to the Y variable and distributing his own tuples to p_Y processors that have the same X-axis. Of course the final distribution is the same if we do it all in once but it is easier to understand the proof thinking that way. Distributing according to X and using the Theorem 6.6 we get that

$$\Pr\left(L_{1x} > (1+\delta)d_X\right) \leq p_X \cdot e^{-h(\delta)}$$

where L_{1x} is the maximum number of tuples in any of the p_X columns. This means that with high probability L_{1_x} would not be more that $(1+\delta)d_X$. Now we consider the distribution of each column to p_Y processors according to the Y variable where the maximum weight is L_{1_x}. We can directly apply the Theorem 6.2 using $\beta = 1$, $K = p_Y$ and $m = L_{1_x}$. Note that in order to use the Theorem we need to ensure that $d_Y \leq \frac{(1+\delta)d_X}{p_Y}$ which is true because $d_Y = O(\log(p_X))$ after distributing the X. The reason why is because the expected number of tuples in each column is 1 value of X which means that the expected degree of y in each column is also 1. So the biggest degree of Y must be $d_Y = O(\log(p_X))$ with the same logic. By applying the 6.2 we get that:

$$\Pr\big(L_{1y} > (1+\delta)^2 \cdot \frac{d_X}{p_Y}) \leq p_Y \cdot e^{-h(\delta)}$$

where L_{1y} is the maximum number of tuples in any processor in one particular column given that the load in that column is at most $(1+\delta)d_X$. To find the total probability of all, we do a Union Bound over all p_X columns because this result is just for one. We then do an other Union Bound of the event that $(1+\delta)d_X$ would be the maximum load in all columns and that there would be a normal distribution to the Y-axis : The result is that

$$\Pr\big(L_2 > (1+\delta)^2 \cdot \frac{d_X}{p_Y}) \leq (p + p_X) \cdot e^{-h(\delta)}$$

We apply the same logic when we have to distribute values with heavy Y. There cannot be an X value with degree more than p_Y and thus while distributing the heavy Y we get an O(logp_Y) maximum degree of X and consecutively the same inequality as the one above but instead of $\frac{d_X}{p_Y}$ we have $\frac{d_Y}{p_X}$ which is less than $\frac{d_X}{p_Y}$ so we can substitute it.

$$\Pr\big(L_2 > (1+\delta)^2 \cdot \frac{d_X}{p_Y}) \leq (p + p_Y) \cdot e^{-h(\delta)}$$

Using an Union Bound to calculate the probability that both light and heavy values will be distributed well, and the fact that $p_X + P_Y \leq p$ we get that :

$$\Pr\big(L_2 > (1+\delta)^2 \cdot Opt2) \leq 8p \cdot e^{-h(\delta)} \tag{6.1}$$

To conclude we proved that with high probability, we have an $O(\log^2(p))$ approximation for the case of 2 dimensions. $\square$

The key to this proof is to manage to distribute the one variable without having a concentration of same values for the variable. We manage this by applying the theorem we proved for the 1 dimension. Furthermore, we take advantage that there cannot be many values with high degrees and thus we can handle those few in such way that they will be distributed evenly with high probability. Finally, we take advantage that if many events happen with exponentially small probability, then their union will also happen with exponentially small probability. This means that the chance of diverging from the desired approximation remains small in all the cases.

The next thing we are interested in solving is the case of 2 dimensions where $d_{XY} \neq 1$. This question is not only interesting to complete the proof for all the possible input we can have for the case of 2 dimensions, but is also a key step in order to generalize the proof for the case of r dimension. We conjecture that for the case of $d_{XY} \neq 1$ we can still get a polylogarithmic approximation to the lower bound.

So far, the proof of theorem 6.1 guarantees that for the case where $d_{XY} \leq m/p$ the bound still holds. With our proof of theorem 6.7 we have proved that even for the case where $d_{XY} > m/p$ we can guarantee the logarithmic approximation as long as the the degree d_{XY} is less that $max\{\frac{d_X}{p_y}, \frac{d_Y}{p_X}\}$.

The only case left that we have not proved that this bound holds is when $d_{XY} > \max\left\{\frac{d_X}{p_y}, \frac{d_Y}{p_X}, \frac{m}{p}\right\}$. By solving that case we would have completed the proof for any 2 dimensional input relation and we would have made a key step for generalizing the proof.

6.3 Conclusion

In this chapter we presented the main result of the algorithm for the case of no skew and when the skew is under a threshold. Furthermore, for the case of 1 and 2 dimensions we proved that we can get near optimal results even when we have values with degrees that surpass this threshold .

The question we are naturally asking is : can we generalize it to n dimensions where $n \geq 3$ and prove that we get near optimal results compared to the lower bounds of the algorithm? We conjecture that for d dimensions, an O($\log^d p$) approximation algorithm is feasible under any distribution. Can we prove that?

While trying to solve the problem where, in our input, the d_{XY} is the dominant term of the lower bound, many questions and problems are arising concerning the binary relation with a strong graph-theoretical aspect.

The binary relationship can be represented as a bipartite graph where the X variable are on the left and the Y variables on the right. Each edge in the graph represents a tuple that has the corresponding X and Y variable that the edge connects. Having said that, given a sequence of values of X and Y, what is the worst edge assignment scenario for that distribution according to a given randomized algorithm? Furthermore, out of all sequences what is the worst sequence case you can have ?

By Solving these problems we kill two birds with one stone. First we contribute to some interesting graph theoretic problems and at the same time we get intuition and one step closer on how to generalize and prove the bound we want.

Bibliography

[1] Foto N. Afrati, Jacek Sroka, and Jan Hidders, editors. *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2016, San Francisco, CA, USA, July 1, 2016*. ACM, 2016.

[2] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In Manolescu et al. [24], pages 99–110.

[3] Yossi Azar, Ilan Reuven Cohen, Seny Kamara, and F. Bruce Shepherd. Tight bounds for online vector bin packing. In Boneh et al. [8], pages 961–970.

[4] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *CoRR*, abs/1306.5972, 2013.

[5] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. *CoRR*, abs/1401.1872, 2014.

[6] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication cost in parallel query processing. *CoRR*, abs/1602.06236, 2016.

[7] Petra Berenbrink, Friedhelm Meyer auf der Heide, and Klaus Schroder. Allocating weighted jobs in parallel. *Theory Comput. Syst.*, 32(3):281–300, 1999.

[8] Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors. *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*. ACM, 2013.

[9] Eric A. Brewer and Peter Chen, editors. *6th Symposium on Operating System Design and Implementation (OSDI) 2004), San Francisco, California, USA, December 6-8, 2004*. USENIX Association, 2004.

[10] Chandra Chekuri and Sanjeev Khanna. On multidimensional packing problems. *SIAM J. Comput.*, 33(4):837–851, 2004.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Brewer and Chen [9], pages 137–150.

[12] Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors. *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 2014.

[13] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. Parallel bottom-up processing of datalog queries. *J. Log. Program.*, 14(1&2):101–126, 1992.

[14] Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *j-J-ACM*, 28(2):289–304, April 1981.

[15] Venkatesan Guruswami, editor. *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. IEEE Computer Society, 2015.

[16] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In Dyreson et al. [12], pages 881–884.

[17] Sungjin Im, Nathaniel Kell, Janardhan Kulkarni, and Debmalya Panigrahi. Tight bounds for online vector scheduling. In Guruswami [15], pages 525–544.

[18] Russell Impagliazzo and Valentine Kabanets. Constructive proofs of concentration bounds. In Serna et al. [31], pages 617–631.

[19] Christos Kaklamanis and Lefteris M. Kirousis, editors. *SIROCCO 9, Proceedings of the 9th International Colloquium on Structural Information and Communication Complexity, Andros, Greece, June 10-12, 2002*, volume 13 of *Proceedings in Informatics*. Carleton Scientific, 2002.

[20] Paraschos Koutris and Dan Suciu. A guide to formal analysis of join processing in massively parallel systems. *SIGMOD Record*, 45(4):18–27, 2016.

[21] Paraschos Koutris and Nivetha Singara Vadivelu. Deterministic load balancing for parallel joins. In Afrati et al. [1], page 10.

[22] Elias Koutsoupias, Marios Mavronicolas, and Paul G. Spirakis. Approximate equilibria and ball fusion. In Kaklamanis and Kirousis [19], pages 223–235.

[23] Michael Luby, José D. P. Rolim, and Maria J. Serna, editors. *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain, October 8-10, 1998, Proceedings*, volume 1518 of *Lecture Notes in Computer Science*. Springer, 1998.

[24] Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors. *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*. ACM, 2010.

[25] Adam Meyerson, Alan Roytman, and Brian Tagiku. Online multidimensional load balancing. In Raghavendra et al. [29], pages 287–302.

[26] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001.

[27] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[28] Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In Luby et al. [23], pages 159–170.

[29] Prasad Raghavendra, Sofya Raskhodnikova, Klaus Jansen, and José D. P. Rolim, editors. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, Berkeley, CA, USA, August 21-23, 2013. Proceedings*, volume 8096 of *Lecture Notes in Computer Science*. Springer, 2013.

[30] Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 2013.

[31] Maria J. Serna, Ronen Shaltiel, Klaus Jansen, and José D. P. Rolim, editors. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 13th International Workshop, APPROX 2010, and 14th International Workshop, RANDOM 2010, Barcelona, Spain, September 1-3, 2010. Proceedings*, volume 6302 of *Lecture Notes in Computer Science*. Springer, 2010.

[32] Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors. *Proceedings of the 20th International Conference*

on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011. ACM, 2011.

[33] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In Srinivasan et al. [32], pages 607–614.

[34] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[35] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In Ross et al. [30], pages 13–24.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Το θέμα αυτής της διπλωματικής εργασίας είναι οι αλγόριθμοι διαμοιρασμού και εξισορρόπησης φορτίου. Οι αλγόριθμοι αυτοί καλούνται να λύσουν προβλήματα που εμφανίζονται τόσο σε πολλούςς τομείς της καθημερινής μας ζωής όσο και στην επιστήμη των υπολογιστών και της επιχειρησιακής έρευνας. Γνωστά σε όλους παραδείγματα είναι η διαχείριση ανθρωπίνου κεφαλαίου, η χρονοδρομολόγηση εργασιών και ρύθμιση μεταφορών αλλά και εφαρμογές σε δίκτυα υπολογιστών. Αυτές οι εφαρμογές έχουν άμεση σχέση με την αγορά εργασίας και η έρευση αποδοτικών αλγορίθμων για την γρηγορότερη αλλά και δικαιότερη επίλυση τους έχει μεγάλο και θετικό αντίκτυπο στην κοινωνία. Τέλος, πέρα από τη πρακτική τους πλευρά, η μελέτη των προβλημάτων αυτών παρουσιάζει και μεγάλο ακαδημαϊκό ενδιαφέρον καθώς είναι προβλήματα με πλούσια συνδυαστική δομή που έλκουν τους επιστήμονες της πληροφορικής και των μαθηματικών να τα μελετήσουν.

1.1 Προσεγγιστικοί Αλγόριθμοι

Στη γενική περίπτωση των προβλημάτων αυτών η έρευση της βέλτιστης λύσης δεν είναι πάντα υπολογιστικά εφικτή. Ο λόγος που συμβαίνει αυτό είναι ότι πολλά προβλήματα του τύπου αυτού έχουν εκθετικά πολλές λύσεις και είναι μη αποδοτικό ακόμα και για έναν υπέρ-υπολογιστή να ελέγξει όλες τις δυνατές λύσεις και να επιλέξει την καλύτερη. Η διαδικασία αυτή απαιτεί τόσο πολύ χρόνο που είτε την καθιστά πρακτικά άλυτη, είτε μετά από το χρονικό διάστημα αυτό οι λύσεις δεν μας είναι χρήσιμες. Για το λόγο αυτό, αντί να ψάχνουμε πάντα για τη βέλτιστη λύση βάση μιας αντικειμενικής συνάρτησης, επικεντρωνόμαστε στην δημιουργία προσεγγιστικών αλγορίθμων που θα μας επιστρέφουν μια σχεδόν βέλτιστη λύση, όμως θα το κάνουν αυτό πολύ αποδοτικά από άποψη χρόνου και πόρων.

1.2 Ορισμός Προβλήματος

Το πρόβλημα που θα μελετήσουμε στην εργασία αυτή είναι πως να διαμοιράσουμε βέλτιστα τα πολυδιάστατα δεδομένα που έχουμε σε υπολογιστικές μονάδες, ώστε ο διαμοιρασμός να είναι όσο το δυνατό πιο ίσος γίνεται. Συγκεκριμένα θα ασχοληθούμε με ένα είδος πολυδιάστατου φορτίου, τις τούπλες δεδομένων. Επιθυμούμε να τις κατανέμουμε με συγκεκριμένο τρόπο ανάμεσα στους υπολογιστές που διαθέτουμε ώστε να υπολογίσουμε παράλληλα τα αποτελέσματα διαφόρων ερωτήσεων στη βάση δεδομένων. Τέτοιου είδους ερωτήσεις συμβαίνουν με ιλιγγιώδες ρυθμό σε σύγχρονα δεδομένα και υπολογιστικά συστήματα, οπότε η σημασία στην εξοικονόμηση χώρου είναι προφανής στην απόδοση των συστημάτων αυτών.

1.3 Δύο Αλγόριθμοι Επίλυσης

Θα προσεγγίσουμε με δυο διαφορετικούς τρόπους τη λύση του προβλήματος αυτού. Ο ένας είναι ντετερμινιστικός, όπου θα χρησιμοποιήσουμε μια ντεντερμινιστικη μέθοδο για την επιλογή του υπολογιστή που θα πάει κάθε δεδομένο της βάσης. Ο δεύτερος είναι πιθανοτικός, όπου για λόγους απόδοσης, θα εισάγουμε τυχαιότητα στον αλγόριθμό μας και κατά συνέπεια η επιλογή του υπολογιστή για κάθε δεδομένο θα είναι κατά έναν αριθμό τυχαία. Και για τις δύο αυτές προσεγγίζεις παρουσιάζουμε δύο ξεχωριστούς αλγορίθμους του οποίους αναλύουμε και συγκρίνουμε βάση της αποδοτικότητας τους. Η ανάλυση θα γίνει βάση της διάστασης των δεδομένων, του όγκου τους, τις συχνότητες εμφάνισης τους αλλά και τον διαθέσιμο αριθμό υπολογιστών.

Κεφάλαιο 2

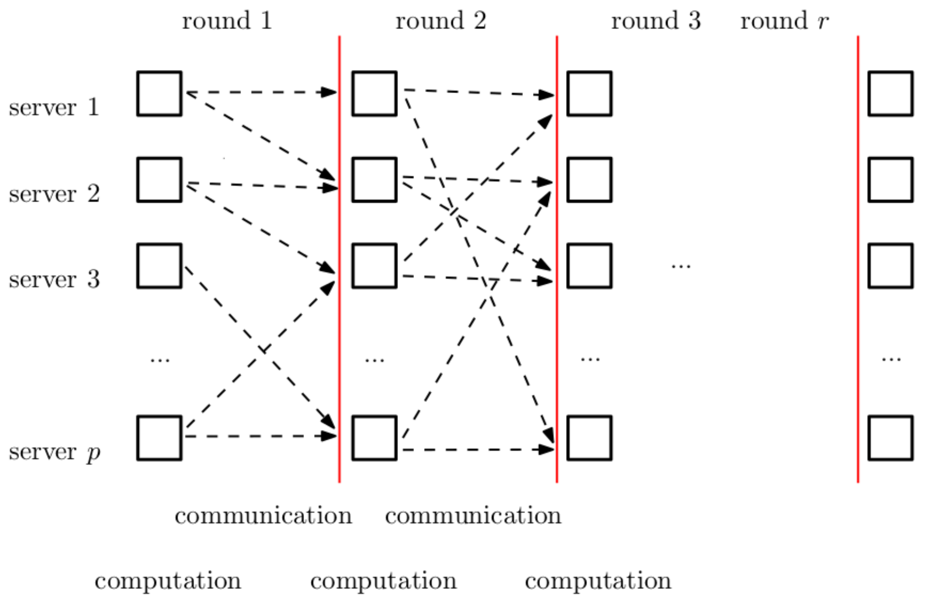
Μοντέλο

Σε αυτό το κεφάλαιο θα αναφερθούμε σύντομα στο μοντέλο το οποίο θα δουλέψουμε καθώς και γενικά στο ποιος είναι ο σκοπός των αλγορίθμων εξισορρόπησης φορτίου.

2.1 Μοντέλο MPC

Το MPC μοντέλο εισήχθη [10] [21] και αποτελείται από p υπολογιστές, η επεξεργαστές, που διασυνδέονται με ένα πλήρες δίκτυο ιδιωτικών καναλιών. Όλοι οι υπολογισμοί συμβαίνουν από αυτούς τους υπολογιστές, και ο καθένας μπορεί να επικοινωνήσει με οποιοδήποτε άλλο με έναν δυσδιάκριτος τρόπο. Αυτοί οι υπολογιστές μπορούν να τρέξουν κάποιον παράλληλο αλγόριθμο σε γύρους επικοινωνίας, όπου κάθε γύρος αποτελείται από δύο ξεχωριστές φάσεις:

- **Φάση Επικοινωνίας:** Οι υπολογιστές ανταλλάζουν δεδομένα μεταξύ τους.
- **Φάση Υπολογισμού:** Κάθε υπολογιστής εκτελεί τους υπολογισμούς του πάνω στα τοπικά δεδομένα που έχει μαζέψει από τους προηγούμενους γύρους.



Θεωρούμε δεδομένα εισόδου M σε bits τα οποία είναι αρχικά κατανεμημένα ομοιόμορφα στους p υπολογιστές. Η απόδοση ενός αλγορίθμου στο μοντέλο αυτό κρίνεται από τον αριθμό των γύρων που χρειάζεται για να εκτελεστεί αλλά και το μέγιστο φόρτο δεδομένων που θα λάβει κάποιος από τους υπολογιστές. Εμείς θα ασχοληθούμε με αλγόριθμους που τρέχουν σε αυτό το μοντέλο με μόνο ένα γύρο επικοινωνίας. Επομένως αφού έχουμε συγκεκριμενοποιήσει τον αριθμό των γύρων, μας ενδιαφέρει μόνο ποιο είναι το μέγιστο φόρτο L ενός από τους υπολογιστές για να χαρακτηρίσουμε την απόδοση τους αλγορίθμου.

Έχουμε αμελήσει το υπολογιστικό κομμάτι όταν συζητάμε για το πόσο αποδοτικός θεωρείται ένας αλγόριθμος που τρέχει το μοντέλο αυτό. Ο λόγος που έχει γίνει αυτό είναι ότι στα σύγχρονα συστήματα ο κύριος παράγοντας που προκαλεί καθυστερήσεις είναι η επικοινωνία μεταξύ των υπολογιστών και όχι το υπολογιστικό κομμάτι που γίνεται τοπικά στον καθένα. Επιπλέον, έχουμε συμπεριλάβει το υπολογιστικό κομμάτι ως συνάρτηση του όγκου των δεδομένων που διαθέτει κάθε υπολογιστής. Συγκεκριμένα, όσα περισσότερα δεδομένα έχει τόσο περισσότερο χρόνο θέλει να τα επεξεργαστεί. Για το λόγο αυτό μας ενδιαφέρει ο υπολογιστής με τα περισσότερα δεδομένα καθώς αυτός θα ολοκληρώσει του υπολογισμούς του τελευταίος και τότε θα είμαστε σε θέση να λάβουμε το αποτέλεσμα.

Κεφάλαιο 3

Ορισμός του Προβλήματος

Σε αυτό το κεφάλαιο θα ορίσουμε αναλυτικά το πρόβλημα που προσπαθούμε να λύσουμε. Πριν το κάνουμε αυτό θα παρουσιάσουμε τον αλγόριθμο του υπερκύβου και θα εξηγήσουμε γιατί αυτός βρίσκει πάντα σωστές λύσεις στη συγκεκριμένη μορφή ερωτήσεων που θέλουμε να υπολογίσουμε στη βάση δεδομένων.

3.1 Αλγόριθμος Υπερκύβου

Ο **Αλγόριθμος Υπερκύβου**, Hypercube algorithm, εισήχθη από τους Αφράτη και Ulman [1] και μοιάζει πολύ με έναν αλγόριθμο των Suri και Vassilviskii [23] για τον υπολογισμό του αριθμού τριγώνων σε γράφο. Η ιδέα πίσω από τον αλγόριθμο είναι απλή. Τοποθετούμε τους διαθέσιμους υπολογιστές σε έναν υπερκύβο και δίνουμε συντεταγμένες σε κάθε ένα από αυτούς ανάλογα με το σε ποια συντεταγμένη του υπερκύβου βρίσκεται. Στη συνέχεια διαμοιράζει τις τούπλες των δεδομένων στους υπολογιστές αυτούς ανάλογα με τη τιμή κάθε μεταβλητής κάθε τούπλας. Φροντίζει τούπλες που έχουν κοινή τιμή στην ίδια μεταβλητή να πάνε σε επεξεργαστές που η αντίστοιχες μεταβλητές των συντεταγμένων τους θα είναι και αυτές ίδιες. Με λίγα λόγια κάνει μια απεικόνιση των μεταβλητών των δεδομένων στον υπερκύβο διάστασης k.

Αυτός ο αλγόριθμος υπολογίζει παράλληλα πάντα σωστά το αποτέλεσμα των ερωτήσεων πάνω στη βάση καθώς για κάθε τούπλα-απάντηση της που θα περιέχει συγκεκριμένες τιμές, ο υπολογιστής που θα βρίσκεται στην τοποθεσία που μας δίνει η συνάρτηση απεικόνισης των τιμών αυτών θα μας επιστρέψει την τούπλα αυτή σαν απάντηση στο ερώτημα μας.

3.2 Ορισμός Προβληματος

Όπως είναι φανερό, υπάρχουν πολλοί διαφορετικοί τρόποι να διαμοιράσουμε τα δεδομένα μας ώστε να ικανοποιήσουμε τον περιορισμό που θέτει ο παραπάνω αλγόριθμος. Εμείς ψάχνουμε να βρούμε το διαμοιρασμό αυτό που θα τηρεί τον παραπάνω περιορισμό και θα έχει το ελάχιστον μέγιστο φόρτο δεδομένων ανάμεσα στους υπολογιστές. Ουσιαστικά θέλουμε να βρούμε έναν τρόπο να επιλέγουμε την λύση αυτή που ισομοιράζει όσο το δυνατόν καλύτερα τα δεδομένα. Άμα το καταφέρουμε αυτό τότε έχουμε βρει έναν πολύ γρήγορο αλγόριθμο που δεν υπερφορτώνει μερικούς υπολογιστές με δεδομένα και αφήνει άλλους σχεδόν ανενεργούς με ελάχιστα δεδομένα.

3.3 Ακραίες Συχνότητες

Όπως μπορεί να έγινε αντιληπτό, εφόσον οι τιμές των διαφόρων μεταβλητών των δεδομένων επηρεάζουν το υπολογιστή που θα διαμοιραστούν οι τούπλες, τότε η συχνότητες εμφάνισης αυτών των τιμών επηρεάζουν το αποτέλεσμα. Συγκεκριμένα έχουμε δύο περιπτώσεις, αυτή που όλες οι συχνότητες είναι μικρότερες του αναμενόμενου φόρτου $\frac{m}{p}$ όπου m οι τούπλες και p οι διαθέσιμοι υπολογιστές, και αυτή όπου υπάρχει τουλάχιστον μια τιμή μιας μεταβλητής, ή ένα σύνολο τιμών, που ξεπερνάει το αναμενόμενο φόρτο δεδομένων. Στη δεύτερη περίπτωση λέμε πως έχουμε εμφάνιση ακραίων συχνοτήτων. Το φαινόμενο αυτό θέλει ξεχωριστή μελέτη καθώς οι βέλτιστες λύσεις που μπορούμε να πετύχουμε όταν έχουμε τέτοιες τιμές στα δεδομένα εισόδου αλλάζουν δραματικά.

Κεφάλαιο 4

Ντετερμινιστικοί Αλγόριθμοι

Έχοντας παρουσιάσει το πρόβλημα το οποίο προσπαθούμε να λύσουμε, θα παρουσιάσουμε συνοπτικά στο κεφάλαιο αυτό δύο ντετερμινιστικούς αλγόριθμους που θα μας δίνουν πολύ καλές προσεγγιστικές λύσεις, ο πρώτος για τη περίπτωση της 1 διάστασης, και ο δεύτερος για τις 2 διαστάσεις.

4.1 Άπληστο Ισοζυγιο

Ο αλγόριθμος άπληστου ισοζύγιου **Greedy Balance** επιλύει το πρόβλημα της μίας διάστασης δίνοντας πάντα μια λύση προσεγγίσης που αποκλείνει το πολύ **2** φορές απο τη βέλτιστη λύση. Αυτό σημαίνει οτι σε πολύ μικρό και αποδοτικό χρόνο, ο αλγόριθμος αυτός μας δινει μια σχεδόν βέλτιστη λύση. Παρακάτω φαίνεται αναλυτικά το τι κάνει:

Algorithm 1 Άπληστο Ισοζυγιο

```
$i \leftarrow 1$
**for all** $a \in adom(X_1)$ **do**
    **if** $L_{(i)} \leq m/p$ **then**
        **for all** $t \in R$ s.t $t.X_1 = a$ **do**
            $D(t) \leftarrow (i)$
        **end for**
    **else**
        $i \leftarrow i + 1$
    **end if**
**end for**
```

Εν συντομία μοιράζει σε κάθε επεξεργαστή/υπολογιστή δεδομένα μέχρι να ξεπεράσει τον αναμενόμενο φόρτο εργασίας και έπειτα αλλάζει επεξεργαστή.

4.2 Αλγόριθμος VLB

Μπαίνοντας στις 2 διαστάσεις υπάρχει ένας αλγόριθμος που λέγεται VLB και επιλύει το πρόβλημα εξισορρόπησης πολυδιάστατων φορτίων αρκεί να ξέρει τα βάρη που έχει κάθε δουλειά σε κάθε μηχανή. Αυτός ο αλγόριθμος δίνει ένα προσεγγιστικό αποτέλεσμα της τάξης του $O(log(nd))$ όπου n ο αριθμός των μηχανών και d ο αριθμός της διάστασης των δεδομένων.

Algorithm 2 Αλγόριθμος VLB

```
$\beta \leftarrow (1 + \frac{1}{\gamma})^{1/\Lambda}$
**for all** jobs $j$ **do**
    assign job $j$ to machine $s = \underset{i}{argmin} \sum_{k=1}^{d} \left[\beta^{l_i^k(j-1)+a_j^k} - \beta^{l_i^k(j-1)}\right]$
    **for all** k **do**
        let $l_s^k(j) \leftarrow l_s^k(j-1) + a_j^k$
    **end for**
**end for**
```

Εμείς θα χρησιμοποιήσουμε τον αλγόριθμος αυτό σαν μαύρο κουτί για να δημιουργήσουμε έναν άλλο αλγόριθμο που θα επιλύει το δικό μας πρόβλημα για τις δύο διαστάσεις.

4.3 Αλγόριθμος 2-Ισοζύγιο

Σε αυτό το κομμάτι του κεφαλαίου θα παρουσιάσουμε έναν αλγόριθμο που θα επιλύει ντετερμινιστικά το πρόβλημα μας για τη περίπτωση των δύο διαστάσεων.

Αφού ο υπερκύβος αποτελείται από 2 διαστάσεις τότε έχουμε ένα πλέγμα στο οποίο πάνω τοποθετούμε τους υπολογιστές. Αυτό το πλέγμα έχει διάσταση p_X και p_Y τέτοια ώστε $p_X \cdot p_Y = p$. Ας υποθέσουμε χωρίς βλάβη της γενικότητας ότι $\geq p_Y$. Ορίζουμε τα δεδομένα ως Heavy και ως Light ανάλογα με τη συχνότητα εμφάνισης της X μεταβλητής τους.

Συγκεκριμένα αν η X μεταβλητή εμφανίζεται τουλάχιστον p_Y φορές τότε είναι Heavy ενώ στην αντίθετη περίπτωση όπου η συχνότητα εμφάνισης δεν ξεπερνάει τη τιμή p_Y, την θεωρούμε LIGHT. Στο παρακάτω σχήμα φαίνεται τι κάνει ο αλγόριθμος:

Algorithm 3 2-Balance

```
split $X$ to Heavy and Light values.
**for all** $a \in$ Heavy$(X)$ **do**
   apply Greedy Balance
**end for**
**for all** $b \in$ Light$(X)$ **do**
   **for all** $Y \in$ Light$(X)$ **do**
      apply Greedy Balance
   **end for**
   $\forall X \in$ Light$(X)$ get the weights $a_j^k$
   apply $\Lambda$-VLB to the Light $X$ Values
**end for**
$\forall$ $X$ get the weights $a_j^k$
Apply $\Lambda$-VLB to $X$ value.
```

Όπως φαίνεται στο ψευδοκώδικα, κάνουμε χρήση 2 φορές του GreedyBalance και 2 φορές του VLB. Άρα, ο όρος προσέγγισης που πετυχαίνει ο αλγόριθμος αυτός είναι $O(log^2 p)$ όπου p ο αριθμός των υπολογιστών.

Για περιπτώσεις παραπάνω των 2 διαστάσεων δεν έχουν μελετηθεί αλγόριθμοι που να λύνουν το πρόβλημα ντετερμινιστικά. Επίσης, ο προηγούμενος αλγόριθμος έχει υποθέσει ότι λειτουργεί σε σετ δεδομένων όπου κάθε τούπλα εμφανίζεται μια φορά. Ανεβαίνοντας διαστάσεις μπορεί κάποιο υποσύνολα τιμών σε συγκεκριμένες μεταβλητές να εμφανίζονται περισσότερες φορές και αυτό δεν παράγει αποδοτικές λύσεις και περιπλέκει και την ανάλυση.

Κεφάλαιο 5

Πιθανοτικοί Αλγόριθμοι

Στο προηγούμενο κεφάλαιο μελετήσαμε τη υλοποίηση του αλγορίθμου υπερκύβου με ντετερμινιστικό τρόπο. Είδαμε όμως πως οι αλγόριθμοι όσο αύξαναν οι διαστάσεις περιπλέκουν και δεν μπορεί να αντιμετωπιστεί η περίπτωση ακραίων συχνοτήτων. Οι πιθανοτικοί αλγόριθμοι έρχονται να καλύψουν το κενό αυτό και να λύσουν μερικά απο τα προβλήματα τα οποία εμφανίζονται στο σχεδιασμό ντετερμινιστικών μεθόδων. Το μυστικό τους είναι ότι υλοποιούν απλές ιδέες και μετά μένει να αποδείξεις ότι με πολύ μεγάλη πιθανότητα θα πάρεις το επιθυμητό αποτέλεσμα.

5.1 Υλοποίηση Υπερκύβου με Hashing

Για να ικανοποιήσουμε τους περιορισμούς που είχε θέσει ο αλγόριθμος του Υπερκύβου, θα δημιουργήσουμε k τέλειες hashing συναρτήσεις, μία για κάθε μεταβλητή της τούπλας αλλά και κάθε διάσταση του κυβου, οι οποίες Θα απεικονίζουν κάθε τιμή της μεταβλητής με ίση πιθανότητα σε μία τιμή της αντίστοιχης συντεταγμένης του υπερκύβου όπου έχουμε τοποθετήσει τους υπολογιστές. Έτσι, προφανώς κάθε δύο τούπλες που έχουν μια ίδια πρώτη μεταβλητή, θα αντιστοιχηθούν σε μια ίδια πρώτη συντεταγμένη αφού θα χρησιμοποιήσουν την ίδια συνάρτηση hashing στις ίδιες μεταβλητές. Επομένως, αφήνουμε την κατανομή των τουπλών στην "τύχη" και ευελπιστούμε να ισομοιραστούν καθώς θεωρούμε σπάνιο να "τύχουν" όλες πάνω σε συγκεκριμένους υπολογιστές.

Όπως μπορεί εύκολα να γίνει αντιληπτό, τρέχοντας τον ίδιο αλγόριθμο πολλές φορές στα ίδια δεδομένα, μπορεί να μας δώσει διαφορετικές κατανομές καθώς πάντα υπάρχει ένας μεγάλος βαθμός τυχαιότητας. Όμως, όπως αποδεικνύουμε σε αυτή τη διπλωματική, η πιθανότητα να ξεφύγει από το προσεγγιστικό παράγοντα που θέλουμε να αποδείξουμε είναι εκθετικά μικρή.

5.2 Αποτελέσματα

Το παρακάτω θεώρημα μας δίνει τι συμβαίνει στην περίπτωση όπου έχουμε δεδομένα χωρίς ακραίες συχνότητες.

Theorem 5.1. *Έχοντας μια Σχέση μια Βάσης δεδομένων $R(A_1, ...A_r)$ μεγέθους m και p συνολικά υπολογιστές διατεταγμένους έναν υπερκύβο διαστάσεων $p_1, ..., p_r$ τέτοιο ώστε $p = \prod_i p_i$.Υποθέτουμε ότι κάνουνε hash κάθε τουπλα $(a_1, ..., a_r)$ σε κάθε υπολογιστή με συντεταγμένες $(h_1(a_1), ...h_r(a_r))$, όπου $h_1, ...h_r$ είναι ανεξαρτητες και τέλειες συναρτήσεις hash. Επιπλέον υποθέτουμε ότι για κάθε τούπλα J πάνω στο $U \subseteq [r]$, έχουμε $d_J(R) \leq \frac{m}{a^{|U|}\prod_{i\in U} p_i}$ για κάποια σταθερά $a > 0$ και $d_J(R)$ η συχνότητα εμφάνισης. Τότε η πιθανότητα το μέγιστο φορτίο κάποιου επεξεργαστή να ξεπεράσει το O($\frac{m}{p} log^r p$) είναι εκθετικά μικρή ως προς το p*

Σε αυτή την εργασία μελετήσαμε τη περίπτωση των ακραίων συχνοτήτων και καταλήξαμε στο συμπέρασμα ότι για μια βάση δεδομένων η προσέγγιση O($\frac{m}{p} log^r p$) ισχύει και για την 1 διάσταση σε κάθε περίπτωση αλλά και για τις 2 διαστάσεις αρκεί τα δεδομένα εισόδου να είναι σετ, δηλαδή να μην υπάρχουν διπλές τιμές. Βέβαια, ο λόγος προσέγγισης αυτός δεν είναι πάνω στην ιδανικά βέλτιστη τιμή που είναι $\frac{m}{p}$ αλλά στο νέο lower bound το οποίο είναι μεγαλύτερο και προκύπτει από τις μεταβλητές που εμφανίζονται σε ακραίες συχνότητες. Αυτή τη τιμή είναι που προσεγγίσαμε και αποδείξαμε ότι ο αλγόριθμος δεν ξεφεύγει πάνω από O($\frac{m}{p} log^r p$) με μεγάλη πιθανότητα.

Επομένως μετά απο τη δική μας συνεισφορά, προκύπτουν τα αποτελέσματα που φαίνονται στον πίνακα:

Πίνακας 5.1: Προσεγιστικοί Παράγοντες Αλγορίθμοι Υπερκύβου ανάλογα τις συχνότητες

Αλγόριθμος Υπερκύβου	
Συχνότητες και διαστάση	**Προσεγγιστικός παράγοντας**
Χωρίς ακραίες συχνότητες διάστασης d	$O(log^d p)$
Με ακραίες συχνότητες διάστασης 1	$O(logp)$
Με ακραίες συχνότητες διάστασης 2	$O(log^2 p)$

Κεφάλαιο 6

Επίλογος

6.1 Μελλοντικοί Στόχοι

Έχοντας κάνει πρόοδο πάνω σε αλγορίθμους για την επίλυση του προβλήματος σε μικρές διαστάσεις, επιθυμούμε να τους γενικεύσουμε για να δουλέουν για κάθε διάσταση και για κάθε είδος δεδομένων εισόδου.

Συγκεκριμένα, στη περίπτωση του ντετερμινιστικού αλγόριθμου θέλουμε να αποδείξουμε το bound και για την περίπτωση που δεν έχουμε σετ. Έτσι, θα κάνουμε ένα βήμα για την επίλυση των τριών διαστάσεων και θα πάρουμε μια πολύ καλή εικόνα για το τι ισχύει για διαστάσεις μεγαλύτερες του 2.

Για το πιθανοτικό αλγόριθμο. θέλουμε να δούμε τι συμβαίνει όταν ο μεγαλύτερος παράγοντας του lower bound είναι το d_{XY} για τη περίπτωση των 2 διαστάσεων και αποδεικνύοντας ότι πάλι ισχύει ο προσεγγιστικός λογαριθμικός παράγοντας συμπληρώνουμε πλήρως την απόδειξη για της δύο διαστάσεις. Ο λόγος που αυτός δεν το χουμε ήδη καταφέρει είναι ότι υπάρχει μεγάλος βαθμός αλληλεξάρτησης των κατανομών των τιμών και δεν μπορούμε να εφαρμόσουμε την ίδια μέθοδο που εφαρμόζαμε μέχρι στιγμής. Άμα το καταφέρουμε αυτό, θα έχουμε κάνει ένα σπουδαίο βήμα για τη γενίκευση του αλγορίθμου σε κάθε περίπτωση δεδομένων εισόδου και για όλες τις διαστάσεις.

Τέλος, ενδιαφερόμαστε να αναπτύξουμε αλγόριθμους επίλυσης, που να μην ικανοποιούν τους περιορισμούς του υπερκύβου, αλλά να είναι το ίδιο διαμοιρασμένοι, παράλληλοι και αποδοτικοί.

Bibliography

[1] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In Manolescu et al. [2], pages 99–110. ISBN 978-1-60558-945-9. doi: 10.1145/1739041.1739056. URL `http://doi.acm.org/10.1145/1739041.1739056`.

[2] Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors. *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, 2010. ACM. ISBN 978-1-60558-945-9.

[3] Adam Meyerson, Alan Roytman, and Brian Tagiku. Online multidimensional load balancing. In Raghavendra et al. [4], pages 287–302. ISBN 978-3-642-40327-9. doi: 10.1007/978-3-642-40328-6_21. URL `https://doi.org/10.1007/978-3-642-40328-6_21`.

[4] Prasad Raghavendra, Sofya Raskhodnikova, Klaus Jansen, and José D. P. Rolim, editors. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, Berkeley, CA, USA, August 21-23, 2013. Proceedings*, volume 8096 of *Lecture Notes in Computer Science*, 2013. Springer. ISBN 978-3-642-40327-9. doi: 10.1007/978-3-642-40328-6. URL `https://doi.org/10.1007/978-3-642-40328-6`.

[5] Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *j-J-ACM*, 28(2):289–304, April 1981. ISSN 0004-5411 (print), 1557-735X (electronic).

[6] Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In Luby et al. [7], pages 159–170. ISBN 3-540-65142-X.

doi: 10.1007/3-540-49543-6_13. URL https://doi.org/10.1007/3-540-49543-6_13.

[7] Michael Luby, José D. P. Rolim, and Maria J. Serna, editors. *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain, October 8-10, 1998, Proceedings*, volume 1518 of *Lecture Notes in Computer Science*, 1998. Springer. ISBN 3-540-65142-X. doi: 10.1007/3-540-49543-6. URL https://doi.org/10.1007/3-540-49543-6.

[8] Petra Berenbrink, Friedhelm Meyer auf der Heide, and Klaus Schroder. Allocating weighted jobs in parallel. *Theory Comput. Syst.*, 32(3):281–300, 1999.

[9] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001. doi: 10.1109/71.963420. URL https://doi.org/10.1109/71.963420.

[10] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. *CoRR*, abs/1401.1872, 2014. URL http://arxiv.org/abs/1401.1872.

[11] Paraschos Koutris and Nivetha Singara Vadivelu. Deterministic load balancing for parallel joins. In Afrati et al. [12], page 10. ISBN 978-1-4503-4311-4. doi: 10.1145/2926534.2926536. URL http://doi.acm.org/10.1145/2926534.2926536.

[12] Foto N. Afrati, Jacek Sroka, and Jan Hidders, editors. *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2016, San Francisco, CA, USA, July 1, 2016*, 2016. ACM. ISBN 978-1-4503-4311-4. doi: 10.1145/2926534. URL http://doi.acm.org/10.1145/2926534.

[13] Paraschos Koutris and Dan Suciu. A guide to formal analysis of join processing in massively parallel systems. *SIGMOD Record*, 45(4):18–27, 2016. doi: 10.1145/3092931.3092934. URL http://doi.acm.org/10.1145/3092931.3092934.

[14] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication cost in parallel query processing. *CoRR*, abs/1602.06236, 2016. URL http://arxiv.org/abs/1602.06236.

[15] Russell Impagliazzo and Valentine Kabanets. Constructive proofs of concentration bounds. In Serna et al. [16], pages 617–631. ISBN 978-3-642-15368-6.

doi: 10.1007/978-3-642-15369-3_46. URL `https://doi.org/10.1007/978-3-642-15369-3_46`.

[16] Maria J. Serna, Ronen Shaltiel, Klaus Jansen, and José D. P. Rolim, editors. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 13th International Workshop, APPROX 2010, and 14th International Workshop, RANDOM 2010, Barcelona, Spain, September 1-3, 2010. Proceedings*, volume 6302 of *Lecture Notes in Computer Science*, 2010. Springer. ISBN 978-3-642-15368-6. doi: 10.1007/978-3-642-15369-3. URL `https://doi.org/10.1007/978-3-642-15369-3`.

[17] Elias Koutsoupias, Marios Mavronicolas, and Paul G. Spirakis. Approximate equilibria and ball fusion. In Kaklamanis and Kirousis [18], pages 223–235.

[18] Christos Kaklamanis and Lefteris M. Kirousis, editors. *SIROCCO 9, Proceedings of the 9th International Colloquium on Structural Information and Communication Complexity, Andros, Greece, June 10-12, 2002*, volume 13 of *Proceedings in Informatics*, 2002. Carleton Scientific.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Brewer and Chen [20], pages 137–150. URL `http://www.usenix.org/events/osdi04/tech/dean.html`.

[20] Eric A. Brewer and Peter Chen, editors. *6th Symposium on Operating System Design and Implementation (OSDI) 2004), San Francisco, California, USA, December 6-8, 2004*, 2004. USENIX Association. URL `https://www.usenix.org/publications/proceedings/?f[0]=im_group_audience%3A167`.

[21] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *CoRR*, abs/1306.5972, 2013. URL `http://arxiv.org/abs/1306.5972`.

[22] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33 (8):103–111, 1990. doi: 10.1145/79173.79181. URL `http://doi.acm.org/10.1145/79173.79181`.

[23] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In Srinivasan et al. [24], pages 607–614. ISBN 978-1-4503-0632-4. doi: 10.1145/1963405.1963491. URL `http://doi.acm.org/10.1145/1963405.1963491`.

[24] Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors. *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, 2011. ACM. ISBN 978-1-4503-0632-4.

[25] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. Parallel bottom-up processing of datalog queries. *J. Log. Program.*, 14(1&2):101–126, 1992. doi: 10.1016/0743-1066(92)90048-8. URL `https://doi.org/10.1016/0743-1066(92)90048-8`.

[26] Yossi Azar, Ilan Reuven Cohen, Seny Kamara, and F. Bruce Shepherd. Tight bounds for online vector bin packing. In Boneh et al. [27], pages 961–970. ISBN 978-1-4503-2029-0. doi: 10.1145/2488608.2488730. URL `http://doi.acm.org/10.1145/2488608.2488730`.

[27] Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors. *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, 2013. ACM. ISBN 978-1-4503-2029-0. URL `http://dl.acm.org/citation.cfm?id=2488608`.

[28] Chandra Chekuri and Sanjeev Khanna. On multidimensional packing problems. *SIAM J. Comput.*, 33(4):837–851, 2004. doi: 10.1137/S0097539799356265. URL `https://doi.org/10.1137/S0097539799356265`.

[29] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In Dyreson et al. [30], pages 881–884. ISBN 978-1-4503-2376-5. doi: 10.1145/2588555.2594530. URL `http://doi.acm.org/10.1145/2588555.2594530`.

[30] Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors. *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014. ACM. ISBN 978-1-4503-2376-5. URL `http://dl.acm.org/citation.cfm?id=2588555`.

[31] Sungjin Im, Nathaniel Kell, Janardhan Kulkarni, and Debmalya Panigrahi. Tight bounds for online vector scheduling. In Guruswami [32], pages 525–544. ISBN 978-1-4673-8191-8. doi: 10.1109/FOCS.2015.39. URL `https://doi.org/10.1109/FOCS.2015.39`.

[32] Venkatesan Guruswami, editor. *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, 2015. IEEE Computer Society. ISBN 978-1-4673-8191-8. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7352273`.

[33] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In Ross et al. [34], pages 13–24. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465288. URL `http://doi.acm.org/10.1145/2463676.2465288`.

[34] Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, 2013. ACM. ISBN 978-1-4503-2037-5. URL `http://dl.acm.org/citation.cfm?id=2463676`.

[35] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. ISBN 0-521-47465-5.