Query Processing for Massively Parallel Systems

Paraschos Koutris

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Reading Committee:

Dan Suciu, Chair

Paul Beame

Magdalena Balazinska

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Query Processing for Massively Parallel Systems

Paraschos Koutris

Chair of the Supervisory Committee:
Professor Dan Suciu
Computer Science and Engineering

The need to analyze and understand big data has changed the landscape of data management over the last years. To process the large amounts of data available to users in both industry and science, many modern data management systems leverage the power of massive parallelism. The challenge of scaling computation to thousands of processing units demands that we change our thinking on how we design such systems, and on how we analyze and design parallel algorithms. In this dissertation, I study the fundamental problem of query processing for modern massively parallel architectures.

I propose a theoretical model, the *MPC model* (Massively Parallel Computation), to analyze the performance of parallel algorithms for query processing. In the MPC model, the data is initially evenly distributed among p servers. The computation proceeds in *rounds*: each round consists of some local computation followed by global exchange of data between the servers. The computational complexity of an algorithm is characterized by both the number of rounds necessary, and the maximum amount of data, or *maximum load*, that each processor receives. The challenge is to identify the optimal tradeoff between the number of rounds and maximum load for various computational tasks.

As a first step towards understanding query processing in the MPC model, we study conjunctive queries (multiway joins) for a single round. We show that a particular type of distributed algorithm, the HyperCube algorithm, can optimally compute join queries when

restricted to one communication round and data without skew.

In most real-world applications, data has skew (for example a graph with nodes of large degree) that causes an uneven distribution of the load, and thus reduces the effectiveness of parallelism. We show that the HyperCube algorithm is more resilient to skew than traditional parallel query plans. To deal with any case of skew, we also design data-sensitive techniques that identify the outliers in the data and alleviate the effect of skew by further splitting the computation to more servers.

In the case of multiple rounds, we present nearly optimal algorithms for conjunctive queries for the case of data without skew. A surprising consequence of our results is that they can be applied to analyze iterative computational tasks: we can prove that, in order to compute the connected components of a graph, any algorithm requires more than a constant number of communication rounds. Finally, we show a surprising connection of the MPC model with algorithms in the external memory model of computation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation is a culmination of five years as a graduate student at the University of Washington. I would like to take this opportunity to thank all the people who helped and supported me throughout this academic journey, to whom I will be always grateful.

First of all I would like to express my gratitude to my advisor, Dan Suciu, for being an inspiration and the best mentor I could hope for. I am deeply appreciative of the countless hours we spent brainstorming on the whiteboard, and of how his support and advice helped grow into the researcher I am now.

I would also like to thank the other members of my committee, Magdalena Balazinska and Paul Beame, whose support and feedback have been indispensable during all these years. I particularly valued Magda's advice, which often provided me with a systems perspective on my theoretical work, and helped me mature as a researcher. My appreciation is also extended to Paul, for all the knowledgeable discussions about many things, not always research related.

I would also like to extend my gratitude to several other people at the University of Washington who have assisted me along the way: Bill Howe, Anna Karlin, Luke Zettlemoyer, Lindsay Mitsimoto and all my collaborators and mentors from around the world: Dimitris Fotakis, Stathis Zachos, Aris Pagourtzis, Jef Wijsen, Foto Afrati, Jef Ullman, Jonathan Goldstein.

I would also like to give a huge thanks to Emad Soroush, Prasang Upadhyaya, Nodira Koushainova, YongChul Kwon, Alexandra Meliou, Abhay Jha, Sudeepa Roy, Eric Gribkoff, Shumo Chu, Dan Halperin and Marianne Shaw for the wonderful discussions and their friendship throughout all these years. To all the other members of the database group at the University of Washington I owe them many thanks for the community we created: Jingjing

Wang, Shengliang Xu, Jennifer Ortiz, Laurel Orr, Ryan Maas, Kristi Morton, Dominik Moritz, Jeremy Hyrkas, Victor Ameida, Wolfgang Gatterbauer.

This process would have not been nearly as enjoyable without the support of my colleagues and close friends: Mark Yatskar, Ricardo Martin, Yoav Artzi, Brandon Myers, Nicholas Fitzgerald, Qi Shan and Dimitrios Gklezakos.

These acknowledgements would not have been complete without a special note to my family in Greece, who has supported me in more ways that I can count, even though they are so far away. I will always be grateful for their advice and their encouragement to follow this academic path.

Last, but not least, I would like to thank my partner, Natalie, who, since we started our PhDs at the same time, has been my co-traveller in this journey. She has been my biggest supporter through all the ups and downs, and I will be always grateful to her for making this experience so much fun.

Chapter 1

INTRODUCTION

Over the last decade, there has been a large increase in the volume of data that is being stored, processed and analyzed. In order to extract value from the large amounts of data available to users in both industry and science, the collected data is typically moved through a pipeline of several data processing tasks, which include cleaning, filtering, joining, aggregating [32]. Improving the performance of these tasks is a fundamental problem in big data analysis, and many modern data management systems have resorted to the power of parallelism to speed up the computation. Parallelism enables the distribution of computation for data-intensive tasks into hundreds, and even thousands of machines, and thus significantly reduces the completion time for several crucial data processing tasks.

The motivating question of this dissertation is the following: *How can we analyze the behavior of query processing algorithms in massively parallel environments?* In this dissertation, we explore the parameters that influence system behavior in this scale of parallelism, and present a theoretical framework, called the *Massively Parallel Model, or MPC.* We then apply the MPC model to rigorously analyze the computational complexity of various parallel algorithms for query processing, with a primary focus on *join processing.* Using the MPC model as a theoretical tool, we show how we can design novel algorithms and techniques for query processing, and how we can prove their optimality.

1.1 Motivation

Query processing for big data is typically performed on a *shared-nothing parallel architecture* [76]. In a shared nothing architecture, the processing units share no memory or other resources, and communicate with one another by sending messages via an interconnection

network. Parallel database systems, pioneered in the late 1980s by GAMMA [38], have been using such a parallel architecture: an incomplete list of these systems includes Teradata [6], Netezza [5], Vertica [7] and Greenplum [2].

The use of the shared nothing architecture and the scaling to an even larger number of machines became prevalent following the introduction of the MapReduce framework [37], its open source implementation Hadoop [3], and the rich ecosystem of extensions and languages that has been built on top (PigLatin [44, 66], Hive [74]).

Several other big data management systems were developed after the introduction of the MapReduce paradigm: for example, Scope [31], Dryad [52, 84] and Dremel [60]. More recently big data analysis systems have been built to support efficient iterative computing and machine learning tasks, in addition to standard query processing: Spark [85] and its SQL-extension Shark [81], Hyracks [28] and the software stack of ASTERIX that is built on top [19], Stratosphere [39], and the system that has been developed by the Database group at the University of Washington, Myria [4]. We should also mention two systems based on the Bulk Synchronous Parallel (BSP) model [77]: Pregel [59], developed for parallel graph processing, and Apache Hama [1]. [1]

Reasoning about the computational complexity of algorithms in such massively parallel systems requires that we shift from our thinking of traditional query processing. Typically, a query is evaluated by a sufficiently large number of servers such that the entire data can be kept in the main memory of these servers; hence, the complexity is no longer dominated by the number of disk accesses. Instead, the new complexity bottleneck becomes the amount of communication, and how this communication is distributed among the available computational resources. Indeed, as we increase the number of machines to which the computation is distributed, more communication is required. Even though the interconnection networks are often faster than accessing the local disk, managing the communication cost becomes a major concern for both system and algorithm designers.

[1] GraphLab [58] and Grappa [62], although they run on shared nothing architectures, hide the underlying architecture from the programmer and expose instead a shared-memory logical interface.

In addition, in most systems (such as MapReduce and Pregel), synchronization is guaranteed at every step of the computation. Any data reshuffling requires global synchronization of all servers, which comes at a significant cost; synchronizing requires additional computational resources and communication. Moreover, in synchronous computation we often see the phenomenon of *stragglers*, which are machines that complete a computational task slower than others [2] In the context of MapReduce framework, this phenomenon is commonly referred to as *the curse of the last reducer* [73]. Since we have to wait for every machine to finish at every synchronization step, limiting the number of synchronization steps is an important design consideration in such parallel systems.

An additional reason for the appearance of stragglers is the uneven distribution of the computational or data load among the available resources. Consequently, apart from the communication cost and the amount of synchronization, a fundamental rule when we parallelize at scale is that the computational and data load must be distributed as evenly as possible between all the available machines. Thus, we have to communicate as little as possible, while simultaneously making sure that the data is partitioned evenly among the servers. This means that we have to account for a common phenomenon that occurs in every data partitioning or computation-balancing method, the presence of *skew*. Data skew, in particular, appears when certain parts of the data are more 'hot' than others. As an example, consider the Twitter follower graph, where we store an edge for each 'follows' relation between two users: in this case, the Justin Bieber node results in skew, since a disproportionate amount of edges in the graph will refer to it. Parallel systems can either detect the presence of skew in runtime and rebalance dynamically [57], or use data statistics to deploy skew-resilient algorithms.

In summary, the parameters that are dominating computation in massively parallel systems are the total communication, the number of synchronization steps, and the maximum data load over all machines (instead of the average load). The MPC model that we intro-

[2]Stragglers appear for many reasons, such as hardware heterogeneity or multi-tenancy.

duce captures all these three parameters in a simple but powerful model that allows us to not only analyze the behavior of parallel algorithms, but also show their optimality through lower bounds.

1.2 Contribution

In this dissertation, we introduce the *Massively Parallel Computation model, or MPC,* as a theoretical tool to analyze the performance of parallel algorithms for query processing on relational data.

In the MPC model, the data is initially evenly distributed among p servers/machines. The computation proceeds in *rounds, or steps*: each round consists of some local computation followed by global exchange of data between the servers. The computational complexity of an algorithm is characterized by both the number of rounds r necessary, and the maximum amount of data, or *maximum load*, L, that each machine receives. An ideal parallel algorithm would use only one round and distribute the data evenly without any replication, hence achieving a maximum load M/p, where M is the size of the input data. Since this is rarely possible, the algorithms need to use more rounds, have an increased maximum load, or both.

Using the MPC model as a theoretical framework, we then identify the *optimal tradeoff between the number of rounds and maximum load for query processing tasks*, and in particular for the computation of conjunctive queries (join queries). Join processing is one of the central computational tasks when processing relational data, and a key component of any big data management system. Understanding this tradeoff equips system designers with knowledge about how much synchronization, communication and load the computation of a query requires, and what is possible to achieve under specific system constraints. Our results in this setting are summarized below.

Input Data without Skew. We establish tight upper and lower bounds on the maximum load L for algorithms that compute full [3] join queries in a single round. We show that a

[3]By full join queries we mean join queries that return all possible outputs, *i.e.* there are no projections.

particular type of distributed algorithm, the HyperCube algorithm, which was introduced in [15] as the Shares algorithm, can *optimally* compute join queries when the input is restricted to having no skew.

More formally, consider a conjunctive query q on relations $S_1, \ldots, S_\ell$, of size $M_1, \ldots, M_\ell$ respectively. Let $\mathbf{u} = (u_1, \ldots, u_\ell)$ be a *fractional edge packing* [35] of the hypergraph of the query q: such a packing assigns a fractional value u_j to relation S_j, such that for every variable x, the sum of the values of the relations that include x is at most 1. Then, we show that any (randomized or deterministic) algorithm that computes q in a single round must have load

$$L = \Omega\left(\left(\frac{\prod_{j=1}^{\ell} M_j^{u_j}}{p}\right)^{1/\sum_j u_j}\right)$$

Moreover, we show that the HyperCube algorithm matches this lower bound asymptotically when executed on data with no skew. As an example, for the *triangle query* $C_3(x, y, z) = S_1(x, y), S_2(y, z), S_3(z, x)$, when all relations have size equal to M, we have an MPC algorithm that computes the query in a single round with an optimal load of $O(M/p^{2/3})$. (The optimal edge packing in this case is $(1/2, 1/2, 1/2)$.) Our analysis of the HyperCube algorithm further shows that it is more resilient to skew than traditional parallel join algorithms, such as the parallel hash-join.

For multi-round algorithms, we establish upper and lower bounds in a restricted version of the MPC model that we call the tuple-based MPC model: this model restricts communication so that only relational tuples can be exchanged between servers. Both our upper and lower bounds hold for *matching databases*, where each value appears exactly once in any attribute of a relation (and so the skew is as small as possible). We show that to achieve a load L in r rounds for a given query, we have to construct a query plan of depth r, where each operator is a subquery that can be computed by the HyperCube algorithm in a single round with load L. For example, to compute the query $L_8 = S_1(x_1, x_2), \ldots, S_8(x_8, x_9)$, we can either use a bushy join tree plan of depth 3, where each operation is a simple join between two relations, which will result in a load of $O(M/p)$. Alternatively, since we can compute the

4-way join L_4 in a single round with load $O(M/p^{1/2})$, we can have a plan of depth 2; thus, we can have a 2-round algorithm with load $O(M/p^{1/2})$. We prove that this type of plan is almost optimal for a large class of queries, which we call *tree-like queries*. To the best of our knowledge, these are the first lower bounds on the load of parallel query processing algorithms for multiple rounds.

Input Data with Skew. In many real-world applications, the input data has *skew*; for example, a node in a graph with large degree, or a value that appears frequently in a relation. In this dissertation, we present several algorithms and techniques regarding how we handle skew in the context of the MPC model, mostly focusing on single-round algorithms.

We show first that the HyperCube algorithm, even though it is suboptimal in the presence of skew, is more resilient to skewed load distribution than traditional parallel join algorithms. We then present a general technique of handling skew when we compute join queries; this technique requires though that we know additional information about the outliers in the data and their frequency (we call these *heavy hitters*). We show how to apply this technique to obtain optimal single-round algorithms for *star queries* and the *triangle query* C_3. Our algorithms are optimal in a strong sense: they are not worst-case optimal, but are optimal for the particular data distribution of the input.

For general join queries, we present a single-round algorithm, called the BinHC algorithm, which however does not always match our lower bounds.

Beyond Query Processing. Our techniques for proving lower bounds on the round-load tradeoff for computing join queries imply a powerful result on a different problem, the problem of computing *connected components in a graph*. We prove that any algorithm that computes the connected components on a graph of size M with load that is $o(M)$ cannot use a constant number of rounds. To the best of our knowledge, this is the first result on bounding the number of rounds for this particular graph processing task.

External Memory Algorithms. We show a natural connection of the MPC model with the *External Memory Computational model*. In this model, there exists an internal memory of size M, and a large external memory, and the computational complexity of an algorithm is defined as the number of input/output operations (I/Os) of the internal memory. The main result is that any MPC algorithm can be translated to an algorithm in the external memory model, such that an upper bound on the load translates directly to an upper bound on the number of I/Os.

We show surprisingly that we can apply this connection to obtain an (almost) optimal algorithm for C_3 in the external memory model. This result hence implies that designing parallel algorithms in the MPC model can lead to advancement in the current state-of-the-art in external memory algorithms.

1.3 Organization

We begin this thesis by providing some background and terminology, along with the exposition of some technical tools, in Chapter 2. We then formally define the MPC model in Chapter 3, and present a detailed comparison with previous models for parallel processing.

In Chapter 4, we present our results for computing join queries in a single round for data without skew, and we discuss both the algorithms and lower bounds. We study query processing in the presence of skew in Chapter 5, where we introduce new algorithms to handle the outliers present in the data. In Chapter 6, we present algorithms and lower bounds for multiple rounds, both for data with and without skew.

In Chapter 7, we discuss the surprising connection between the MPC model and the external memory model. We finally conclude in Chapter 8.

Chapter 2

BACKGROUND

In this chapter, we present some background that will be necessary to the reader to follow this dissertation. We present in detail the class of *conjunctive queries*, which will be the queries this dissertation focuses on. We then lay some notation and useful mathematical inequalities that will prove handy throughout this work.

2.1 Conjunctive Queries

The main focus in this work is the class of *Conjunctive Queries (CQ)*. A conjunctive query q will be denoted as

$$q(x_1, \ldots, x_k) = S_1(\bar{x}_1), \ldots, S_\ell(\bar{x}_\ell) \tag{2.1}$$

The notation we are using here is based on the Datalog language (see [8] for more details). The atom $q(x_1, \ldots, x_k)$ is called the *head* of the query. For each atom $S_j(\bar{x}_j)$ in the *body* of the query, S_j is the name of the relation in the database schema. We denote the arity of relation S_j with a_j, and also write $a = \sum_{j=1}^{\ell} a_j$ to express the sum of all the arities for the atoms in the query.

Definition 2.1.1. *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 $k = 0$,* i.e. *the head of the query is $q()$.*

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

Definition 2.1.2. *A CQ q is* self-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 $q(x, y, z) = S(x, y), S(y, z)$ is not self-join-free, since relation S appears twice in the query.

The result $q(D)$ of executing a conjunctive query q over a relational database D is obtained as follows. For each possible assignment α of values to the variables in $\bar{x}_1, \ldots, \bar{x}_\ell$ such that for every $j = 1, \ldots, \ell$ the tuple $\alpha(\bar{x}_j)$ belongs in the instance of relation S_j, the tuple $\alpha(x_1, \ldots, x_k)$ belongs in the output $q(D)$.

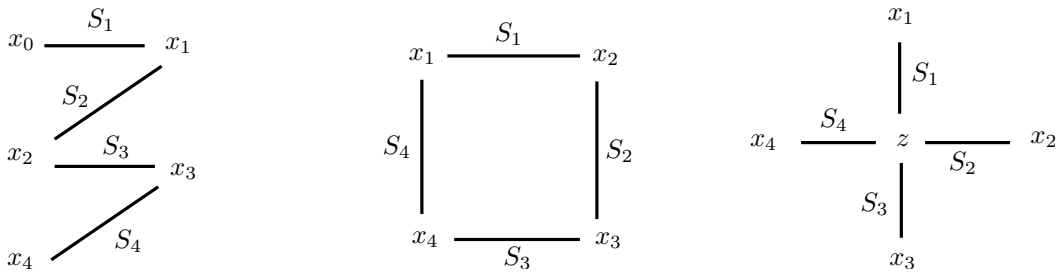
Definition 2.1.3. *The* hypergraph *of a CQ q is defined by introducing one vertex for each variable in the body of the query and one hyperedge for each set of variables that occur in a single atom.*

We say that a conjunctive query is *connected* if the query hypergraph is connected. For example, the query $q(x, y) = R(x), S(y)$ is not connected, whereas $q(x, y) = R(x), S(y), T(x, y)$ is connected. We use $\text{vars}(S_j)$ to denote the set of variables in the atom S_j, and $\text{atoms}(x_i)$ to denote the set of atoms where x_i occurs; k and ℓ denote the number of variables and atoms in q, as in (2.1). The *connected components* of q are the maximal connected subqueries of q.

We define below three important classes of full self-join-free conjunctive queries that will be seen frequently throughout this work.

$$\begin{aligned} L_k(x_0, x_1, \ldots, x_k) &= S_1(x_0, x_1), S_2(x_1, x_2), \ldots, S_k(x_{k-1}, x_k) \\ C_k(x_1, \ldots, x_k) &= S_1(x_1, x_2), S_2(x_2, x_3), \ldots, S_k(x_k, x_1) \\ T_k(z, x_1, \ldots, x_k) &= S_1(z, x_1), S_2(z, x_2), \ldots, S_k(z, x_k) \end{aligned}$$

The first class is the class of *line queries*, the second of *cycle queries*, and the third of *star queries*, and examples of their hypergraphs are in Fig. 2.1.



(a) Hypergraph of L_4 (b) Hypergraph of C_4 (c) Hypergraph of T_4

Figure 2.1: Examples for three different classes of conjunctive queries: (a) line queries L_k, (b) cycle queries C_k and (c) star queries T_k. The hypergraphs are depicted as graphs, since all relations are binary.

2.1.1 The Characteristic of a CQ

We introduce here a new notion, that of the characteristic of a conjunctive query, which will be used later in this work in order to count the number of answers for queries over particular input distributions. The *characteristic* of a conjunctive query q as in (2.1) is defined as $\chi(q) = a - k - \ell + c$, where $a = \sum_j a_j$ is the sum of arities of all atoms, k is the number of variables, ℓ is the number of atoms, and c is the number of connected components of q.

For a query q and a set of atoms $M \subseteq \text{atoms}(q)$, define q/M to be the query that results from contracting the edges in the hypergraph of q. As an example, for the line query L_5 we have that $L_5/\{S_2, S_4\} = S_1(x_0, x_1), S_3(x_1, x_3), S_5(x_3, x_5)$.

Lemma 2.1.4. *The characteristic of a query q satisfies the following properties:*

(a) If $q_1, \ldots, q_c$ are the connected components of q, then $\chi(q) = \sum_{i=1}^{c} \chi(q_i)$.

(b) For any $M \subseteq atoms(q)$, $\chi(q/M) = \chi(q) - \chi(M)$.

(c) $\chi(q) \geqslant 0$.

(d) For any $M \subseteq atoms(q)$, $\chi(q) \geqslant \chi(q/M)$.

Proof. Property (a) is immediate from the definition of χ, since the connected components of q are disjoint with respect to variables and atoms. Since q/M can be produced by contracting

according to each connected component of M in turn, by property (a) and induction it suffices to show that property (b) holds in the case that M is connected. If a connected M has k_M variables, ℓ_M atoms, and total arity a_M, then the query after contraction, q/M, will have the same number of connected components, $k_M - 1$ fewer variables, and the terms for the number of atoms and total arity will be reduced by $a_M - \ell_M$ for a total reduction of $a_M - k_M - \ell_M + 1 = \chi(M)$. Thus, property (b) follows.

By property (a), it suffices to prove (c) when q is connected. If q is a single atom S_j then $\chi(S_j) \geqslant 0$, since the number of variables is at most the arity a_j of the atom. If q has more than one atom, then let S_j be any such atom: then $\chi(q) = \chi(q/S_j) + \chi(S_j) \geqslant \chi(q/S_j)$, because $\chi(S_j) \geqslant 0$. Property (d) follows from (b) using the fact that $\chi(M) \geqslant 0$. □

For a simple illustration of property (b), consider the example above $L_5/\{S_2, S_4\}$, which is equivalent to L_3. We have $\chi(L_5) = 10 - 6 - 5 + 1 = 0$, and $\chi(L_3) = 6 - 4 - 3 + 1 = 0$, and also $\chi(M) = 0$ (because M consists of two disconnected components, $S_2(x_1, x_2)$ and $S_4(x_3, x_4)$, each with characteristic 0). For a more interesting example, consider the query K_4 whose graph is the complete graph with 4 variables:

$$K_4 = S_1(x_1, x_2), S_2(x_1, x_3), S_3(x_2, x_3), S_4(x_1, x_4), S_5(x_2, x_4), S_6(x_3, x_4)$$

and denote $M = \{S_1, S_2, S_3\}$. Then $K_4/M = S_4(x_1, x_4), S_5(x_1, x_4), S_6(x_1, x_4)$ and the characteristics are: $\chi(K_4) = 12-4-6+1 = 3$, $\chi(M) = 6-3-3+1 = 1$, $\chi(K_4/M) = 6-2-3+1 = 2$.

Finally, we define the class of *tree-like* queries, which will be extensively used in Chapter 6 for multi-round algorithms.

Definition 2.1.5. *A conjunctive query q is* tree-like *if q is connected and $\chi(q) = 0$.*

For example, the query L_k is tree-like; in fact, a query over a binary vocabulary is tree-like if and only if its hypergraph is a tree. An important property of tree-like queries is that every connected subquery will be also tree-like.

2.1.2 The Fractional Edge Packing of a CQ

A *fractional edge packing* (also known as a *fractional matching*) of a query q is any feasible solution $\mathbf{u} = (u_1, \ldots, u_\ell)$ of the following linear constraints:

$$\begin{aligned} &\forall i \in [k] : \sum_{j : x_i \in \text{vars}(S_j)} u_j \leqslant 1 \\ &\forall j \in [\ell] : u_j \geqslant 0 \end{aligned} \tag{2.2}$$

The edge packing associates a non-negative weight u_j to each atom S_j such that for every variable x_i, the sum of the weights for the atoms that contain x_i do not exceed 1. If all inequalities are satisfied as equalities by a solution to the LP, we say that the solution is *tight*. The dual notion is a *fractional vertex cover* of q, which is a feasible solution $\mathbf{v} = (v_1, \ldots, v_k)$ to the following linear constraints:

$$\begin{aligned} &\forall j \in [\ell] : \sum_{i : x_i \in \text{vars}(S_j)} v_i \geqslant 1 \\ &\forall i \in [k] : v_i \geqslant 0 \end{aligned}$$

At optimality, $\max_{\mathbf{u}} \sum_j u_j = \min_{\mathbf{v}} \sum_i v_i$; this quantity is denoted τ^* and is called the *fractional vertex covering number of* q.

Example 2.1.6. *An edge packing of the query* $L_3 = S_1(x_1, x_2), S_2(x_2, x_3), S_3(x_3, x_4)$ *is any solution to* $u_1 \leqslant 1$, $u_1 + u_2 \leqslant 1$, $u_2 + u_3 \leqslant 1$ *and* $u_3 \leqslant 1$. *In particular, the solution* $(1, 0, 1)$ *is a tight edge packing; it is also an optimal packing, thus* $\tau^* = 2$.

We also need to refer to the *fractional edge cover*, which is a feasible solution $\mathbf{u} = (u_1, \ldots, u_\ell)$ to the system above where $\leqslant$ is replaced by $\geqslant$ in Eq.(2.2). Every tight fractional edge packing is a tight fractional edge cover, and vice versa. The optimal value of a fractional edge cover is denoted ρ^*. The fractional edge packing and cover have no connection, and there is no relationship between τ^* and ρ^*. For example, for $q = S_1(x, y), S_2(y, z)$, we have

Vertex Covering LP	Edge Packing LP
$\forall j \in [\ell] : \sum_{i:x_i \in \text{vars}(S_j)} v_i \geqslant 1$ (2.3) $\forall i \in [k] : v_i \geqslant 0$	$\forall i \in [k] : \sum_{j:x_i \in \text{vars}(S_j)} u_j \leqslant 1$ (2.4) $\forall j \in [\ell] : u_j \geqslant 0$
minimize $\sum_{i=1}^{k} v_i$	maximize $\sum_{j=1}^{\ell} u_j$

Figure 2.2: The vertex covering LP of the hypergraph of a query q, and its dual edge packing LP.

$\tau^* = 1$ and $\rho^* = 2$, while for $q = S_1(x), S_2(x, y), S_3(y)$ we have $\tau^* = 2$ and $\rho^* = 1$. The two notions coincide, however, when they are tight, meaning that a tight fractional edge cover is also a tight fractional edge packing and vice versa. The fractional edge cover has been used recently in several papers to prove bounds on query size and the running time of a sequential algorithm for the query [22, 63, 64]; for the results in this paper we need the fractional packing.

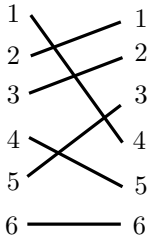
2.1.3 The Database Instance

Throughout this work, we will often focus on specific types of database instances with different properties.

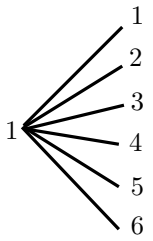
Let R be a relation of arity r. For a tuple t over a subset of the attributes $[r]$ that exists in R, we define $d_t(R) = |\sigma_t(R)|$ as the degree of the tuple t in relation R. In other words, $d_t(R)$ tells us how many times the tuple t appears in the instance of relation R.[1]

A *matching database* restricts the degrees of all relations such that for every tuple t over $U \subseteq [r]$, we have $d_t(R) = 1$; in other words, each value appears at most once in every relation. To see an example of a matching database, consider a binary relation R, and assume that each of the attributes contains the values $1, 2, \ldots, n$. In this scenario, a matching instance of R contains n tuples, and essentially defines a permutation on $[n]$, since each value of the

[1]Notice that our definition of the degree considers only tuples that exist in the relation, and thus the degree can never be zero.



(a) A depiction of a matching instance for relation $R(x, y)$ as a bipartite matching



(b) A depiction of a (maximally) skewed instance for relation $R(x, y)$

Figure 2.3: Examples for two different instances for a binary relation $R(x, y)$: (a) a matching instance over a domain of size 6, (b) a skewed instance.

first attribute maps to a unique value of the second attribute. Notice also that in a matching relation every attribute is a key.

Matching databases are instances *without data skew*. Instances with skew typically have some value, or tuple of values, that appear frequently in the instance. In this work, we do not define an absolute notion of when a relation is skewed, since the measure of skew will be relative to the parallelism available. To give an example of skew for the binary relation R, consider the instance $\{(1, 1), (1, 2), \ldots, (1, n)$, and observe that the degree of the value 1 is n, since 1 appears in all n tuples.

2.2 Entropy

Let us fix a finite probability space. For random variables X and Y, the *entropy* and the *conditional entropy* are defined respectively as follows:

$$H(X) = -\sum_{x} P(X = x) \cdot \log_2 P(X = x) \tag{2.5}$$

$$H(X \mid Y) = \sum_{y} P(Y = y) \cdot H(X \mid Y = y) \tag{2.6}$$

The entropy satisfies the following two basic inequalities:

$$H(X \mid Y) \leqslant H(X)$$
$$H(X,Y) = H(X \mid Y) + H(Y) \tag{2.7}$$

Assuming additionally that X has a support of size n, we have:

$$H(X) \leqslant \log_2 n \tag{2.8}$$

2.3 Yao's Principle

The lower bounds that we show in this work apply not only to deterministic algorithms, but to randomized algorithms as well. To prove lower bounds for randomized algorithms, we use Yao's Principle [83]. In the setting of answering conjunctive queries, Yao's principle can be stated as follows.

Let $\mathcal{P}$ be any probability space from which we choose a database instance I, such that every deterministic algorithm fails to compute $q(I)$ correctly with probability $\geqslant 1-\delta$. Then, for every randomized algorithm, there exists a database instance I' such that the algorithm fails to compute $q(I')$ correctly with probability $\geqslant 1-\delta$, where the probability is over the random choices of the algorithm.

In other words, if we want to prove a lower bound for randomized algorithms, it suffices to construct a probability distribution over instances for which any *deterministic algorithm* fails. As we will see, Yao's principle allows us to prove strong lower bounds for the communication load, even in the case where we allow communication to take arbitrary form.

2.4 Friedgut's Inequality

Friedgut [41] introduces a powerful class of inequalities, which will provide a useful tool for proving lower bounds. Each inequality is described by a hypergraph, but since we work with conjunctive queries, we will describe the inequality using query terminology (and thus the

hypergraph will be the query hypergraph). Fix a query q as in (2.1), and let $n > 0$. For every atom $S_j(\bar{x}_j)$ of arity a_j, we introduce a set of n^{a_j} variables $w_j(\mathbf{a}_j) \geqslant 0$, where $\mathbf{a}_j \in [n]^{a_j}$. If $\mathbf{a} \in [n]^a$, we denote by $\mathbf{a}_j$ the vector of size a_j that results from projecting on the variables of the relation S_j. Let $\mathbf{u} = (u_1, \ldots, u_\ell)$ be a fractional *edge cover* for q. Then:

$$\sum_{\mathbf{a}\in[n]^k} \prod_{j=1}^{\ell} w_j(\mathbf{a}_j) \leqslant \prod_{j=1}^{\ell} \left(\sum_{\mathbf{a}_j\in[n]^{a_j}} w_j(\mathbf{a}_j)^{1/u_j} \right)^{u_j} \tag{2.9}$$

We illustrate Friedgut's inequality on the queries C_3 and L_3:

$$\begin{aligned} C_3(x, y, z) &= S_1(x, y), S_2(y, z), S_3(z, x) \\ L_3(x, y, z, w) &= S_1(x, y), S_2(y, z), S_3(z, w) \end{aligned} \tag{2.10}$$

Consider the cover $(1/2, 1/2, 1/2)$ for C_3, and the cover $(1, 0, 1)$ for L_3. Then, we obtain the following inequalities, where α, β, γ stand for w_1, w_2, w_3 respectively:

$$\begin{aligned} \sum_{x,y,z\in[n]} \alpha_{xy} \cdot \beta_{yz} \cdot \gamma_{zx} &\leqslant \sqrt{\sum_{x,y\in[n]} \alpha_{xy}^2 \sum_{y,z\in[n]} \beta_{yz}^2 \sum_{z,x\in[n]} \gamma_{zx}^2} \\ \sum_{x,y,z,w\in[n]} \alpha_{xy} \cdot \beta_{yz} \cdot \gamma_{zw} &\leqslant \sum_{x,y\in[n]} \alpha_{xy} \cdot \max_{y,z\in[n]} \beta_{yz} \cdot \sum_{z,w\in[n]} \gamma_{zw} \end{aligned}$$

where we used the fact that $\lim_{u\to 0}(\sum \beta_{yz}^{\frac{1}{u}})^u = \max \beta_{yz}$.

Friedgut's inequalities immediately imply a well known result developed in a series of papers [46, 22, 63, 64] that gives an upper bound on the size of a query answer as a function on the cardinality of the relations. For example in the case of C_3, consider an instance S_1, S_2, S_3, and set $\alpha_{xy} = 1$ if $(x, y) \in S_1$, otherwise $\alpha_{xy} = 0$ (and similarly for β_{yz}, γ_{zx}). We obtain then $|C_3| \leqslant \sqrt{|S_1| \cdot |S_2| \cdot |S_3|}$. Note that all these results are expressed in terms of a fractional edge *cover*. When we apply Friedgut's inequality in Chapter 4 to a fractional edge *packing*, we ensure that the packing is tight.

Chapter 3

THE MASSIVELY PARALLEL COMPUTATION MODEL

In this chapter, we introduce the *Massively Parallel Computation* model (MPC), a theoretical model that allows us to analyze algorithms in massively parallel environments. We first give the formal description of the model in Section 3.1, where we also discuss some observations and simplifying assumptions. In Section 3.2, we present a comparison of the MPC model with previous parallel models, and discuss our modeling choices along various axes. Finally in Section 3.3, we study the connections of the MPC model with the area of communication complexity.

3.1 The MPC Model: Computation and Parameters

In the MPC model, introduced in [23, 24], computation is performed by p servers, or processors, connected by a complete network of private channels. Each server can communicate with any other server in an indistinguishable way. The servers run the parallel algorithm in *communication steps*, or *rounds*, where each round consists of two distinct phases[1]:

Communication Phase: The servers exchange data, each by communicating with all other servers (both sending and receiving 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 the data: this describes the way the data is typically par-

[1] In earlier versions of our work on parallel processing [55, 12], we had a third phase called the broadcast phase.

titioned in any distributed storage system, for example in HDFS [71]. We do not make any particular assumptions on whether the data is partitioned according to a specific scheme. Thus, any parallel algorithm must work for an arbitrary data partition, while any lower bound can use a worst-case initial distribution of the data. We should note here that specific partitioning schemes (for example hash-partitioning a relation according to a specific attribute) can help design better parallel algorithms, but this is not something we consider in this dissertation.

After the computation is completed, the *output data* is present in the union of the output of the p servers.

The complexity of a parallel algorithm 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. A smaller number of rounds means that the algorithm can run with less synchronization.

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, we define formally L as:

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

The reader should notice that the MPC model does not restrict or capture the running time of the computation at each server; in other words, the servers can be as computationally powerful as we would like. This modeling choice means that our 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. On the other hand, the algorithms that we present throughout this work are always polynomially bounded and

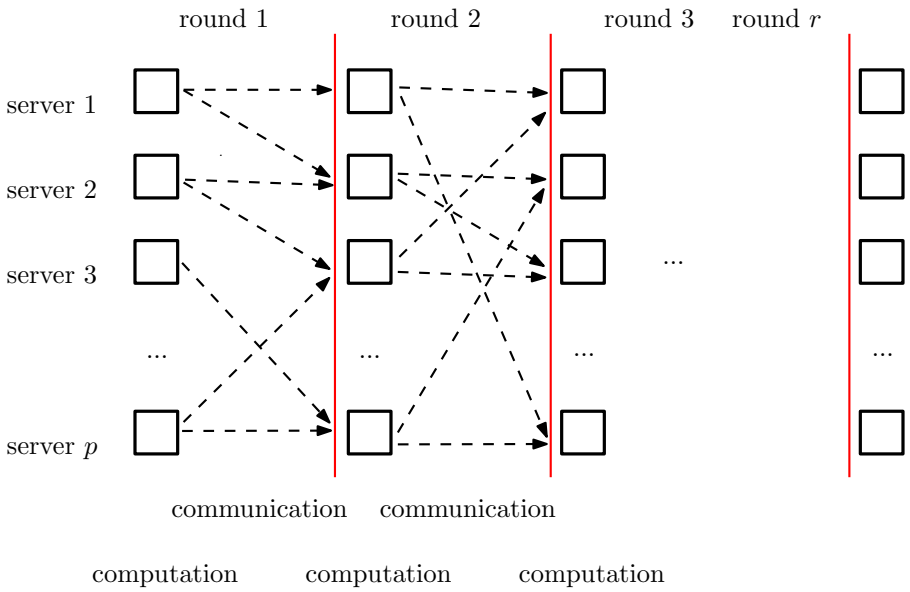


Figure 3.1: Figure of the execution model for MPC. The data is initially distributed among the p servers. In each of the r rounds, the servers perform some local computation and then globally exchange data with each other. The output is the union of the server outputs at the end of round r.

thus can be implemented in practice.

Before we discuss how the parameters we introduced interact with each other, let us give an example of an algorithm in the MPC model and its analysis of the maximum load.

Example 3.1.1 (Set Intersection)**.** *The input consists of two sets R, S, each with m elements, and the goal is to compute the intersection $R \cap S$ of these two sets. To distribute the elements across the p servers, we will use a hash function $h : Dom \to \{1, \ldots, p\}$. The algorithm will run in a single round. During the communication phase, each element $R(a)$ will be sent to server $h(a)$, and each element $S(b)$ to server $h(b)$. During the computation phase, each server s will perform a local intersection of the elements received from relation R and S. Clearly, an element $a \in R \cap S$ will be in the output, since both $R(a), S(a)$ will be received by the server $h(a)$.*

To compute the maximum load L, we will use a probabilistic argument to analyze the behavior of the hash function. Indeed, if we assume that h is a perfectly random hash function,

it can be shown that the load (in number of elements, not bits) will be $O(m/p)$ with high probability when $m \gg p$.

Let us now discuss the interaction between the number of rounds, the input size and the load to provide some intuition on our modeling choices. Normally, the entire data is exchanged during the first communication round, so the load L is at least M/p. Thus, the set intersection algorithm from the above example is asymptotically optimal. On the other hand, if we allowed a load $L = M$, then any problem can solved trivially in one round by simply sending the entire data to server 1, then computing the answer locally. In this case though, we failed to exploit the available parallelism of the p machines. The typical loads in most of our analyses will be of the form $M/p^{1-\varepsilon}$, for some parameter $0 \leqslant \varepsilon < 1$ that depends on the query (we call the parameter ε the *space exponent*, see Subsection 4.4.2).

Observe also that if we allowed the number of rounds to reach $r = p$, any problem can be solved trivially in p rounds by sending at each round M/p bits of data to server 1, until this server accumulates the entire data. In this thesis, we will mostly focus on algorithms that need a constant number of rounds, $r = O(1)$, to perform the necessary computation.

3.1.1 Computing Queries in the MPC model

In this dissertation, the main focus will be the computation of full conjunctive queries, as defined in (2.1). As an example, for the triangle query $R(x, y), S(y, z), T(z, x)$, we want to output all the triangles. It is straightforward that an MPC algorithm that computes all triangles will be able to answer the corresponding boolean query, *i.e.* whether there exist any triangles, but the opposite does not hold, since computing the decision version may require a more efficient algorithm. Computing all answers further allows us to count the answers (for example count the number of triangles), but again the counting version of a query may allow for a better parallel algorithm.

We discuss next two assumptions that will allow us to simplify the treatment of conjunctive queries in the MPC model: the first one concerns the existence of self-joins, and the

second the initial partitioning of the input.

Self-Joins. Instead of studying the computation of any full conjunctive query, we can focus only on algorithms and bounds for self-join-free queries without any loss of generality.

To see this, let q be any conjunctive query, and denote q' the query obtained from q by giving distinct names to repeated occurrences of the same relations (hence q is self-join-free). For example, 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. Then, q' will be the self-join-free triangle query $S_1(x, y), S_2(y, z), S_3(z, x)$.

Clearly, if we have an MPC algorithm $\mathcal{A}'$ for q', we can apply this algorithm directly to obtain an algorithm for q by essentially 'copying' the relations that appear multiple times in the body of the query. The new algorithm will have the same load as $\mathcal{A}'$ when executed on an input that is at most ℓ times larger than the original input (recall that ℓ is the number of relations). In our example, we would copy relation S of size M to create three replicas S_1, S_2, S_3, each of size M, then execute the triangle query on an input 3 times as large, and finally obtain the output answers.

Conversely, suppose that we have an algorithm $\mathcal{A}$ for the query q with self-joins. We construct an algorithm $\mathcal{A}'$ for q' as follows. Suppose that relation S has k occurrences in q and has k distinct names $S_1, \ldots, S_k$ in q'. For each atom $S_i(x, y, z, \ldots)$, algorithm $\mathcal{A}'$ renames every tuple $S_i(a, b, c, \ldots)$ into $S(\langle a, x\rangle, \langle b, y\rangle, \langle c, z\rangle, \ldots)$. That is, each value a in the first column is replaced by the pair $\langle a, x\rangle$, where x is the variable occurring in that column, and similarly for all other columns. This copy operation can be performed locally by all servers, without any additional communication cost. The resulting relation S will be essentially a 'union' of the k relations, but where we remember the variable mapping for each value. We can execute then $\mathcal{A}$ on the constructed input, which will be exactly the same size as the original input. In the end, we have to perform a filtering step to return the correct output for q': if the head variables of q are $x, y, \ldots$ and an output tuple is $(\langle a, u\rangle, \langle b, v\rangle, \ldots)$, then the algorithm needs to check that $x = u, y = v, \ldots$, and only then return the tuple.

Input Partitioning. The MPC model assumes an arbitrary initial distribution of the input relations in the p servers. Our algorithms in the next sections are able to operate with any distribution, since they operate per tuple (and some possibly side information). In order to show lower bounds, however, it will be convenient to use the assumption that initially each relation S_j is stored in a separate server, called an *input server*. During the first round, the input servers send messages to the p servers, but in subsequent rounds they are no longer used in the computation. All lower bounds in this paper assume that the relations S_j are given on separate input servers.

We next show that if we have a lower bound on the maximum load for the model with separate input servers, the bound carries over immediately to the standard MPC model for the class of self-join-free conjunctive queries.

Lemma 3.1.2. *Let q be a self-join-free conjunctive query q with input sizes $\mathbf{M} = (M_1, \ldots, M_\ell)$. Let $\mathcal{A}$ be an algorithm that computes q in r rounds with load $L(\mathbf{M}, p)$ in the standard MPC model. Then, there exists an algorithm $\mathcal{A}'$ that computes q in the input server model in r rounds and load $L(\mathbf{M}, p) + M/p$, where $M = \sum_{j=1}^{\ell} M_j$ is the total size of the input.*

Proof. We will construct an algorithm $\mathcal{A}'$ over the input server model as follows. The input server j will take the input relation S_j, and simply split it into pM_j/M chunks: each chunk now contains M/p data, so we can simulate each of the p servers in the first round of the MPC model. Notice that during the first round we also have to send the initial chunk of the input relation to the corresponding server (which is the data that the server would contain during initialization for $\mathcal{A}$). In subsequent rounds $\geqslant 2$, algorithm $\mathcal{A}'$ behaves exactly the same as algorithm $\mathcal{A}$. We can observe now that algorithm $\mathcal{A}'$ requires r rounds, and achieves a load of $L(\mathbf{M}, p) + M/p$. □

Say that we show a lower bound $L(\mathbf{M}, p)$ for the input server model such that $L(\mathbf{M}, p) \geqslant 2M/p$ (this assumption holds for every lower bound that we show, since intuitively the input must be distributed once among the servers to do any computation). From the above result,

this implies that every algorithm for the standard MPC model using the same number of rounds requires a load of at least $L(\mathbf{M}, p)/2$. Thus, it suffices to prove our lower bounds assuming that each input relation is stored in a separate input server. Observe that this model is even more powerful, because an input server has now access to the entire relation S_j, and can therefore perform some global computation on S_j, for example compute statistics, find outliers, etc., which are common in practice.

3.1.2 The tuple-based MPC model

The MPC model, as defined in the previous sections, allows arbitrary communication among the servers at every round, which makes the theoretical analysis of multi-round algorithms a very hard task. Thus, in order to show lower bounds for the case of multiple rounds, we will need to restrict the form of communication in the MPC model; to do this, we define a restriction of the MPC model that we call the *tuple-based MPC model*. More precisely, the tuple-based MPC model will impose a particular structure on what kind of messages can be exchanged among the servers.

Let I be the input database instance, q be the query we want to compute, and $\mathcal{A}$ an algorithm. For a server $s \in [p]$, we denote by $\mathrm{msg}^1_{j\to s}(\mathcal{A}, I)$ the message sent during round 1 by the input server for S_j to the server s, and by $\mathrm{msg}^k_{s\to s'}(\mathcal{A}, I)$ the message sent to server s' from server s at round $k \geqslant 2$. Let $\mathrm{msg}^1_s(\mathcal{A}, I) = (\mathrm{msg}^1_{1\to s}(\mathcal{A}, I), \ldots, \mathrm{msg}^1_{\ell\to s}(\mathcal{A}, I))$ and $\mathrm{msg}^k_s(\mathcal{A}, I) = (\mathrm{msg}^k_{1\to s}(\mathcal{A}, I), \ldots, \mathrm{msg}^k_{p\to s}(\mathcal{A}, I))$ for any round $k \geqslant 2$.

Further, we define $\mathrm{msg}^{\leqslant k}_s(\mathcal{A}, i)$ to be the vector of messages received by server s during the first k rounds, and $\mathrm{msg}^{\leqslant k}(\mathcal{A}, i) = (\mathrm{msg}^{\leqslant k}_1(\mathcal{A}, i), \ldots, \mathrm{msg}^{\leqslant k}_p(\mathcal{A}, i))$.

Define a *join tuple* to be any tuple in $q'(I)$, where q' is any connected subquery of q. An algorithm $\mathcal{A}$ in the *tuple-based MPC model* has the following two restrictions on communication during rounds $k \geqslant 2$, for every server s

- the message $\mathrm{msg}^k_{s\to s'}(\mathcal{A}, I)$ is a set of join tuples.
- for every join tuple t, the server s decides whether to include t in $\mathrm{msg}^k_{s\to s'}(\mathcal{A}, I)$ based

only on the parameters t, s, s', r, and the messages $\text{msg}^1_{j\to s}(\mathcal{A}, I)$ for all j such that t contains a base tuple in S_j.

The restricted model still allows unrestricted communication during the first round; the information $\text{msg}^1_s(\mathcal{A}, I)$ received by server s in the first round is available throughout the computation. However, during the following rounds, server s can only send messages consisting of join tuples, and, moreover, the destination of these join tuples can depend only on the tuple itself and on $\text{msg}^1_s(\mathcal{A}, I)$.

The restriction of communication to join tuples (except for the first round during which arbitrary, e.g. statistical, information can be sent) is natural and the tuple-based MPC model captures a wide variety of algorithms including those based on MapReduce. Since the servers can perform arbitrary inferences based on the messages that they receive, even a limitation to messages that are join tuples starting in the second round, without a restriction on how they are routed, would still essentially have been equivalent to the fully general MPC model. For example, any server wishing to send a sequence of bits to another server can encode the bits using a sequence of tuples that the two exchanged in previous rounds, or (with slight loss in efficiency) using the understanding that the tuples themselves are not important, but some arbitrary fixed Boolean function of those tuples is the true message being communicated. This explains the need for the condition on routing tuples that the tuple-based MPC model imposes.

3.2 Comparison of MPC to other Parallel Models

In this section, we present several theoretical models that have been developed for parallel computation, and compare them with the MPC model, noting both the modeling similarities and differences. We will discuss only models that capture *shared-nothing* architectures [76]: in such an architecture, every machine has its own memory and there is no shared memory available. This does not include for example the first parallel model, the *Parallel Random-Access Machine (PRAM)* model, where a number of processors share an unbounded memory

and can operate in a synchronous way on a shared input. [2] Further, we discuss models that are *synchronous*, in the sense that the computation and communication proceeds in well-defined rounds. The comparison will be across five different axes:

1. *Number of servers.* The number of servers can be an explicit parameter (as in the MPC model), or it can be chosen by the algorithm according to the input data.
2. *Number of rounds.* This parameter counts the number of rounds, or synchronization barriers, during computation. Reducing the number of steps is important in terms of performance, because of the overhead of synchronization and the presence of *stragglers*, which are machines that finish slower than the rest.
3. *Memory bound.* This parameter models how much memory is available to each server, or how much data (load) each server is allowed to receive during computation. This restriction captures the parallelism in computation: the smaller the amount of data each server receives, the more parallel the task is.
4. *Communication.* This parameter captures how much data is exchanged during the communication between servers. There are many different ways communication is modeled: one can choose to count the total amount of data exchanged, or model it indirectly by looking at the number of servers and the memory bound.
5. *Computation.* Some models require that the computation is polynomial, or that the running time is counted towards measuring the parallel complexity.

The BSP Model. To address some of the issues of the PRAM model, Valiant [77] introduced the **Bulk Synchronous Parallel (BSP)** 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 the *h-relation*, which is defined as a communication pattern where the maximum number of incoming or outgoing messages per machine is h. The cost

[2] Immerman showed in [51] that computing an expression in first-order logic (and thus evaluating conjunctive queries) requires constant time in PRAM, earning the name "embarrassingly parallel".

of a superstep i consists of three components: the cost of synchronization (a constant ℓ), the communication cost h_i (the size of the h_i-relation) and the computation cost w_i (measured as the longest running local computation). The cost of a BSP algorithm is computed as a weighted sum of these terms over all steps: $\sum_i (w_i + gh_i + \ell)$.

The MPC model is similar to the BSP model, but we remove the computation cost from consideration, and also do not allow any form of asynchronous communication; this allows us to prove lower bounds on the cost by using information-theoretic arguments. Moreover, instead of measuring the total communication $\sum_i h_i$ over all supersteps, the MPC model considers the largest value of h_i, which the load L.

Related to the BSP model is the CONGEST model [42] for distributed computing; the difference between the two models is that the CONGEST model considers a communication graph that may not be a full clique, as in the BSP model.

The LogP Model. The **LogP model**, introduced in [36], builds on the BSP model by using more system parameters to model the execution. L denotes the latency of the communication medium, o the overhead of sending and receiving a message, g denotes the gap required between two send/receive operation, and finally P is the number of processing units.

Following the introduction of the MapReduce model [37], several theoretical models were introduced in the literature to capture computation in this setting. MapReduce is a restricted version of the BSP model, where synchronization occurs at every step, and uses a simple programming model inspired by functional programming. In its vanilla version, the user defines two functions: `map` and `reduce`. The map function is applied on a single data item x of the input and returns a key-value pair (k, v). The reduce function is applied to a list of items with the same key, $(k, [v_1, \ldots, v_m])$ and returns a new list of values. During runtime, the system applies in parallel the map function, then performs a *shuffle step* that collects all key-value pairs with the same key to the same location (*reducer*); finally, the reduce function is applied in parallel to each key-group.

The MRC model. In order to capture computation in the MapReduce framework, the authors in [54] define the MapReduce class ($\mathcal{MRC}$) of algorithms. Denoting by M the size of the input in bits, the model restricts both the memory per reducer/machine and the number of machines/reducers to $O(M^{1-\epsilon})$, for some constant $\epsilon > 0$. Observe that this means that the number of machines (or reducers in their case) is not explicitly defined. The machines can perform only polynomial time computations, and further the number of rounds must be limited to $O(\log^c(M))$ for some constant c. In a related work [45], the authors use an explicit parameter B to bound the memory of each reducer, and use the total amount of communication C and number of rounds R as parameters that capture the complexity of a MapReduce algorithm.

Both works do not consider problems related to query processing, but tasks such as sorting or the Minimum Spanning Tree problem. Moreover, they provide no lower bounds on the communication or round complexity.

The Afrati-Ullman model. In [14], Afrati and Ullman develop a model for MapReduce where the main parameter is an upper bound q on the number of input tuples a reducer can receive, called *reducer size*; this is the same as the memory per node. Given an input of size M, a MapReduce algorithm is restricted to deterministically send each input tuple independently to some reducer, which will then produce all the outputs that can be extracted from the received tuples. If $q_i \leqslant q$ is the number of inputs assigned to the i-th reducer, where $i = 1, \ldots, p$, the *replication rate* of the algorithm is $r = \sum_{i=1}^{p} q_i/M$. The replication rate captures the communication cost of the algorithm.

In previous works [15, 10], the authors considered the total communication as the complexity measure instead of the memory per machine, and studied the problems of multiway joins and enumeration of subgraph patterns in a graph. The authors study the tradeoff between r and q for various settings, including Hamming distance (see also [13] for a deeper investigation of Hamming distance algorithms), matrix multiplication, triangle finding, and multiway joins. In particular, the problem of enumerating and/or counting triangles has

	MPC	**BSP**	**MRC**	**Afrati/Ullman**
#servers	p	p	$O(M^{1-\epsilon})$	N/A
#rounds	r	r	$O(\log^c(M))$	1 or 2
memory bound	bits received L	N/A	$O(M^{1-\epsilon})$	reducer size q
communication	$\leqslant rpL$	$\sum_i h_i$	$O(M^{2-2\epsilon})$	replication rate r
computation	N/A	$\sum_i w_i$	polynomial	N/A

Table 3.1: Side-by-side comparison of several parameters for the theoretical parallel models.

been one of the most common problems studied in parallel environments. Suri and Vassilvitskii [73] present two algorithms for triangle counting in a MapReduce environment. We should also mention here the work in [11], where the authors use the same MapReduce model to study multi-round algorithms for join processing.

To complete the discussion on parallel models, we should finally mention the MUD model [40] (which stands for Massive, Unordered and Distributed). The MUD model defines a general distributed computational model that attempts to capture computation in the MapReduce framework; the main result of [40] is that any deterministic streaming algorithm that computes a symmetric function can be simulated by a MUD algorithm with the same total communication cost.

3.3 Communication Complexity

We discuss here the relation of the MPC model with the theoretical study of *communication complexity*; we refer to [56] for a detailed treatment of this area. Communication complexity considers the following scenario: a number of cooperating agents, each with unlimited computational power, need to solve some specific computational problem, but initially know only a part of the input. The goal is to find the optimal amount of communication needed

(in bits) to solve the problem.

There are many versions of this problem, depending on the number of agents, whether randomization is used or how the input is distributed among the agents, but the model closer to MPC is *Number-In-Hand (NIH)* multiparty communication complexity (see [68] for example), where initially each agent receives a part of the input that is not shared with any other agent. Although several computational tasks have been considered in this model, such as set-disjointness or string equality, query processing has not been much studied. It should be noted here that lower bounds for NIH communication complexity are extensively used to obtain results in other areas, such as lower bounds on the space requirements for data streaming algorithms (e.g. [18]).

There are three variations on the communication mode that is being used. In the *blackboard model*, any message sent by an agent is written to a blackboard visible to all other agents. In the *coordinator model*, an additional agent, called the coordinator, receives no input, but all agents can communicate only with the coordinator and not with each other directly. In both of these modes, at least one agent has access to all communication among agents; hence, any computational task with input of size M and output of size O can be computed using no more than $M+O$ bits of communication. The third mode is the *message-passing* model, where there is a private 2-way communication channel connecting any two agents, and every message is privately sent to a specific agent. There has been a recent line of work ([80, 30]) investigating the communication complexity of several tasks in this setting. In [80], the authors investigate various statistical and graph problems (such as connectivity, bipartiteness, degree computation), and show that for most tasks the simple algorithm of communicating all data to a single location is almost optimal. In [30], the authors study the problem of *set intersection*, which can be seen as a case of join computation.

Since the communication in the MPC model is point-to-point private communication, the more related mode is the *message passing* model. However, there is still a key difference: the MPC model measures the communication per agent/machine and per round instead of measuring the total communication. As a result, in many of the computational tasks we

analyze in the next sections, we obtain lower bounds were the total communication is of the form $M^{1+\delta}$, which is much larger than a protocol in the NIH model would require. Thus, by measuring communication in a more fine-grained way, we can obtain stronger lower bounds.

To complete the discussion on communication complexity, we should also mention the work in [47], where the authors examine the 2-party NIH communication complexity for distributed set-joins, which includes multiway joins as a special case.

Chapter 4

COMPUTING JOIN QUERIES IN ONE STEP WITHOUT SKEW

In this chapter, we present the main result of this dissertation. We study the complexity of computing conjunctive queries in the MPC model for a single communication round under the assumption that the input data has no skew. We show that under this assumption, there exists an algorithm, called the HyperCube algorithm, that computes any conjunctive query by achieving the optimal load.

Recall that we represent a full conjunctive query q as:

$$q(x_1, \ldots, x_k) = S_1(\bar{x}_1), \ldots, S_\ell(\bar{x}_\ell)$$

where k is the number of variables and ℓ is the number of atoms. Throughout this chapter, we will assume that the input servers know the cardinalities $m_1, \ldots, m_\ell$ of the relations $S_1, \ldots, S_\ell$. We denote $\mathbf{m} = (m_1, \ldots, m_\ell)$ the vector of cardinalities, and $\mathbf{M} = (M_1, \ldots, M_\ell)$ the vector of the sizes expressed in bits, where $M_j = a_j m_j \log n$, n is the size of the domain of each attribute, and a_j the arity of each atom.

Given the size information, how well can an algorithm do in a single communication round? The central result we show is that a particular type of algorithm, which we call the HyperCube algorithm, if parametrized correctly can achieve optimal load for any conjunctive query q. However, our result does not hold for any input data, but for data without skew. We will give a precise definition of skew later in this chapter, but intuitively no skew means that no value in the input data appears many times.

Recall that a database is a *matching database* if each relation has degree bounded by 1 (*i.e.* the frequency of each value is exactly 1 for each relation). Our lower bounds hold for

an input distribution that consists of such matching databases. One can view a matching database as input with the least amount of skew possible. The upper bound, and in particular the load analysis for the HyperCube algorithm, hold not only for matching databases, but in general for databases with a small amount of skew, which we will formally define in Section 4.1. This means that the HyperCube algorithm has some resilience in skew, and we can quantify exactly how much this resilience is.

4.1 The HyperCube Algorithm

We describe here the HyperCube algorithm, which we will use to compute any conjunctive query in one round. This algorithm was introduced by Afrati and Ullman [9] in a MapReduce setting, and is similar to an algorithm by Suri and Vassilvitskii [73] to count the number of triangles in graphs. The idea though can be traced much earlier in time, to a work by Ganguly [43] on parallel processing of Datalog programs. We call this the HyperCube (HC) algorithm, following [23], but it can also be found in the literature as the Shares algorithm [9].

The algorithm is simple in principle, and the core idea is to perform communication by doing a smart routing of the input tuples. The communication phase of the algorithm is highly distributed, since the destination of each input tuple depends only on the content of the specific tuple, the size $\mathbf{M}$ of the relations and the query q. Thus, the algorithm can be easily implemented in almost any distributed or parallel computing environment.

The HC Algorithm. We initially assign to each variable x_i, where $i = 1, \ldots, k$, a *share* p_i, such that $\prod_{i=1}^{k} p_i = p$. Each server is then represented by a distinct point $\mathbf{y} \in \mathcal{P}$, where $\mathcal{P} = [p_1] \times \cdots \times [p_2]$; in other words, servers are mapped into points of a k-dimensional hypercube.[1]

- **Communication:** We use k independently chosen hash functions $h_i : [n] \to [p_i]$ and

[1]This is where the algorithm takes its name from.

send each tuple t of relation S_j to all servers in the destination subcube of t:

$$\mathcal{D}(t) = \{\mathbf{y} \in \mathcal{P} \mid \forall m = 1, \ldots, a_j : h_{i_m}(t[i_m]) = \mathbf{y}_{i_m}\} \tag{4.1}$$

- **Computation:** Each server locally computes the query q for the subset of the input that it has received.

The correctness of the HC algorithm follows from the observation that, for every potential tuple $(a_1, \ldots, a_k)$, the server $(h_1(a_1), \ldots, h_k(a_k))$ contains all the necessary information to decide whether it belongs in the answer or not. Observe also that the choice of $p_1, \ldots, p_k$ gives a different parametrization of the HC algorithm.

Example 4.1.1. *We illustrate how to compute the triangle query*

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

Consider the following choice of shares: $p_1 = p_2 = p_3 = p^{1/3}$. Each 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 first communication round, the input server storing S_1 sends each tuple $S_1(\alpha_1, \alpha_2)$ to all servers with index $(h_1(\alpha_1), h_2(\alpha_2), y_3)$, for all $y_3 \in [p^{1/3}]$: notice that each tuple is replicated $p^{1/3}$ times. The input servers holding S_2 and S_3 proceed similarly with their tuples. After round 1, any three tuples $S_1(\alpha_1, \alpha_2)$, $S_2(\alpha_2, \alpha_3)$, $S_3(\alpha_3, \alpha_1)$ that contribute to the output tuple $C_3(\alpha_1, \alpha_2, \alpha_3)$ will be seen by the server $\mathbf{y} = (h_1(\alpha_1), h_2(\alpha_2), h_3(\alpha_3))$: any server that detects three matching tuples outputs them.

Analysis of the HC algorithm. In order to analyze the load of the HC algorithm, we first have to study how a relation R is partitioned among the p servers during communication. Our first analysis of the HC algorithm in [23] was only for the special case of matching databases, where the degree of each value is exactly one. In a later work [24], the analysis was extended for relations with larger degrees. We present here the most general result, in

other words we specify the largest possible degree for which the distribution of the tuples will not be influenced by skew. The analysis is based on the following lemma about hashing, which we prove in detail in Section A.2.

Lemma 4.1.2. *Let $R(A_1, \ldots, A_r)$ be a relation of arity r of size m. Let $p_1, \ldots, p_r$ be integers and let $p = \prod_i p_i$. Suppose that we hash each tuple $(a_1, \ldots, a_r)$ to the bin $(h_1(a_1), \ldots, h_r(a_r))$, where $h_1, \ldots, h_r$ are independent and perfectly random hash functions from the domain n to $p_1, \ldots, p_r$ respectively. Then:*

1. *The expected load in every bin is m/p.*
2. *Suppose that for every tuple J over $U \subseteq [r]$ we have $d_J(R) \leqslant \frac{m}{\alpha^{|U|} \prod_{i \in U} p_i}$ for some constant $\alpha > 0$. Then the probability that the maximum load exceeds $\tilde{O}(m/p)$ is exponentially small in p.*[2]

Using the above lemma, we can now prove the following statement on the behavior of the HC algorithm.

Corollary 4.1.3. *Let $\mathbf{p} = (p_1, \ldots, p_k)$ be the shares of the HC algorithm. Suppose that for every relation S_j and every tuple J over $U \subseteq [a_j]$ we have $d_J(S_j) \leqslant \frac{m_j}{\alpha^{|U|} \prod_{i \in U} p_i}$ for some constant $\alpha > 0$. Then with high probability the maximum load per server is*

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

Choosing the Shares. We have not discussed yet how to choose the best shares for the HC algorithm. The above analysis provided us with a tool that allows us to make the best possible choice. Afrati and Ullman in [9] compute the shares by optimizing the total load $\sum_j m_j / \prod_{i:i \in S_j} p_i$ subject to the constraint $\prod_i p_i = 1$, which is a non-linear system that can be solved using Lagrange multipliers. Our approach is to optimize the maximum load *per relation*, $L = \max_j m_j / \prod_{i:i \in S_j} p_i$; the total load per server is $\leqslant \ell L$. This leads to a linear

[2]The notation $\tilde{O}$ hides $\log(p)$ factors.

optimization problem, as follows. First, write the shares as $p_i = p^{e_i}$ where $e_i \in [0,1]$ is called the *share exponent* for x_i, denote $\lambda = \log_p L$ and $\mu_j = \log_p M_j$ (we will assume w.l.o.g. that $M_j \geqslant p$, hence $\mu_j \geqslant 1$ for all j). Then, we optimize the LP:

$$\begin{aligned} \text{minimize} \quad & \lambda \\ \text{subject to} \quad & \sum_{i \in [k]} -e_i \geqslant -1 \\ & \forall j \in [\ell] : \sum_{i \in S_j} e_i + \lambda \geqslant \mu_j \\ & \forall i \in [k] : e_i \geqslant 0, \quad \lambda \geqslant 0 \end{aligned} \tag{4.2}$$

Theorem 4.1.4 (Upper Bound)**.** *For a query q and p servers, with statistics $\mathbf{M}$, let $\mathbf{e} = (e_1, \ldots, e_k)$ be the optimal solution to (4.2) and e^* its objective value.*

Let $p_i = p^{e_i}$ and suppose that for every relation S_j and every tuple J over $U \subseteq [a_j]$ we have $d_J(S_j) \leqslant \frac{m_j}{\alpha^{|U|} \prod_{i \in U} p_i}$ for some constant $\alpha > 0$. Then the HC algorithm with shares p_i achieves $\tilde{O}(L^{upper})$ maximum load with high probability, where $L^{upper} = p^{e^*}$.

A special case of interest is when all cardinalities M_j are equal, therefore $\mu_1 = \ldots = \mu_\ell = \mu$. In that case, the optimal solution to Eq.(4.2) can be obtained from an optimal fractional vertex cover $\mathbf{v}^* = (v_1^*, \ldots, v_k^*)$ by setting $e_i = v_i^*/\tau^*$ (where $\tau^* = \sum_i v_i^*$). To see this, we note that any feasible solution $(\lambda, e_1, \ldots, e_k)$ to Eq.(4.2) defines the vertex cover $v_i = e_i/(\mu - \lambda)$, and in the opposite direction every vertex cover defines the feasible solution $e_i = v_i/(\sum_i v_i)$, $\lambda = \mu - 1/(\sum_i v_i)$; further more, minimizing λ is equivalent to minimizing $\sum_i v_i$. Thus, when all cardinalities are equal to M, at optimality $\lambda^* = \mu - 1/\tau^*$, and $L^{upper} = M/p^{1/\tau^*}$.

We illustrate more examples in Section 4.3.

4.2 The Lower Bound

In this section, we prove a lower bound on the maximum load per server over databases with statistics $\mathbf{M}$.

Fix a query q and a fractional edge packing $\mathbf{u}$ of q. Denote:

$$L(\mathbf{u}, \mathbf{M}, p) = \left(\frac{\prod_{j=1}^{\ell} M_j^{u_j}}{p} \right)^{1/\sum_j u_j} \tag{4.3}$$

Further denote $L^{lower} = \max_{\mathbf{u}} L(\mathbf{u}, \mathbf{M}, p)$, where $\mathbf{u}$ ranges over all edge packings for q. In this section, we will prove that Eq.(4.3) is a lower bound for the load of any algorithm computing the query q, over a database with statistics $\mathbf{M}$. We will prove in Section 4.3 that $L^{lower} = L^{upper}$, showing that the upper bound and lower bound are tight. To gain some intuition behind the formula (4.3), consider the case when all cardinalities are equal, $M_1 = \ldots = M_\ell = M$. Then $L^{lower} = M/p^{1/\sum_j u_j}$, and this quantity is maximized when $\mathbf{u}$ is a maximum fractional edge packing, whose value is τ^*, the fractional vertex covering number for q. Thus, $L^{lower} = M/p^{1/\tau^*}$, which is the same expression as L^{upper}.

To prove the lower bound, we will define a probability space from which the input databases are drawn. Notice that the cardinalities of the ℓ relations are fixed: $m_1, \ldots, m_\ell$. We first choose a domain size $n \geqslant \max_j m_j$, to be specified later, and choose independently and uniformly each relation S_j from all matchings of $[n]^{a_j}$ with exactly m_j tuples. We call this the *matching probability space*. Observe that the probability space contains only databases with relations without skew (in fact all degrees are exactly 1). We write $\mathbf{E}[|q(I)|]$ for the expected number of answers to q under the above probability space.

Theorem 4.2.1 (Lower Bound)**.** *Fix statistics* $\mathbf{m}$*, and consider any deterministic MPC algorithm that runs in one communication round on* p *servers. Let* $\mathbf{u}$ *be any fractional edge packing of* q*. If* s *is any server and* L_s *is its load, then server* s *reports at most*

$$\frac{L_s^{\sum_j u_j}}{(\sum_j u_j/4)^{\sum_j u_j} \prod_{j=1}^{\ell} M_j^{u_j}} \cdot \mathbf{E}[|q(I)|]$$

answers in expectation, where I *is a randomly chosen from the matching probability space with statistics* $\mathbf{m}$ *and domain size* $n = (\max_j m_j)^2$*. Therefore, the* p *servers of the algorithm*

report at most

$$\left(\frac{4L}{(\sum_j u_j)\cdot L(\mathbf{u},\mathbf{M},p)}\right)^{\sum_j u_j}\cdot \mathbf{E}[|q(I)|]$$

answers in expectation, where $L = \max_{s\in[p]} L_s$ *is the maximum load of all servers.*

Furthermore, if if all relations have equal size $m_1 = \ldots = m_\ell = m$ *and arity* $a_j \geqslant 2$*, then one can choose* $n = m$*, and strengthen the number of answers reported by the* p *servers to:*

$$\left(\frac{L}{(\sum_j u_j)\cdot L(\mathbf{u},\mathbf{M},p)}\right)^{\sum_j u_j}\cdot \mathbf{E}[|q(I)|]$$

Therefore, if $\mathbf{u}$ is any fractional edge packing, then $(\sum_j u_j)\cdot L(\mathbf{u},\mathbf{M},p)/4$ is a lower bound for the load of any algorithm computing q. Up to a constant factor, the strongest such lower bound is given by $\mathbf{u}^*$, the optimal solution for Eq.(4.3), since for any $\mathbf{u}$, we have $(\sum_j u_j)\cdot L(\mathbf{u},\mathbf{M},p)/4 \leqslant [(\sum_j u_j)/(\sum_j u_j^*)]\cdot[(\sum_j u_j^*)\cdot L^{lower}/4]$, and $(\sum_j u_j)/(\sum_j u_j^*) \leqslant \tau^* = O(1)$ (since $\sum_j u_j \leqslant \tau^*$ and, at optimality, $\sum_j u_j^* \geqslant 1$).

Before we prove the theorem, we show how to extend it to a lower bound for any randomized algorithm. For this, we start with a lemma that we also need later.

Lemma 4.2.2. *The expected number of answers to* q *is* $\mathbf{E}[|q(I)|] = n^{k-a}\prod_{j=1}^{\ell} m_j$*. In particular, if* $n = m_1 = \cdots = m_\ell$ *then* $\mathbf{E}[|q(I)|] = n^{c-\chi(q)}$*, where* c *is the number of connected components of* q*.*

Proof. For any relation S_j, and any tuple $\mathbf{a}_j \in [n]^{a_j}$, the probability that S_j contains $\mathbf{a}_j$ is $P(\mathbf{a_j} \in S_j) = m_j/n^{a_j}$. Given a tuple $\mathbf{a} \in [n]^k$ of the same arity as the query answer, let $\mathbf{a}_j$ denote its projection on the variables in S_j. Then:

$$\begin{aligned}\mathbf{E}[|q(I)|] &= \sum_{\mathbf{a}\in[n]^k} P(\bigwedge_{j=1}^{\ell}(\mathbf{a}_j \in S_j)) = \sum_{\mathbf{a}\in[n]^k}\prod_{j=1}^{\ell} P(\mathbf{a}_j \in S_j)\\ &= \sum_{\mathbf{a}\in[n]^k}\prod_{j=1}^{\ell} m_j n^{-a_j} = n^{k-a}\prod_{j=1}^{\ell} m_j\end{aligned}$$

□

We now can prove a lower bound for the maximum load of any randomized algorithm, on a fixed database instance.

Theorem 4.2.3. *Consider a connected query q with fractional vertex covering number τ^*. Fix some database statistics $\mathbf{M}$. Let A be any one round, randomized MPC algorithm A for q, with maximum load $L \leqslant \delta \cdot L^{lower}$, for some constant $\delta < 1/(4 \cdot 9^{\tau^*})$. Then there exists an instance I such that the randomized algorithm A fails to compute $q(I)$ correctly with probability $> 1 - 9(4\delta)^{1/\tau^*} = \Omega(1)$*

Proof. We use Yao's principle, which we defined in Section 2.3. To apply the principle, we need to choose the right probability space over database instances I. The space of random matchings is not useful for this purpose, because for a connected query with a large characteristic $\chi(q)$, $\mathbf{E}[|q(I)|] = O(1/n)$ and therefore $P(q(I) \neq \varnothing) = O(1/n)$, which means that a naive deterministic algorithm that always returns the empty answer with fail with a very small probably, $O(1/n)$. Instead, denoting $\mu = \mathbf{E}[|q(I)|]$, we define C_α the event $|q(I)| > \alpha\mu$, where $\alpha > 1$ is some constant. We will apply Yao's principle to the probability space of random matchings conditioned on C_α.

We prove that, for $\alpha = 1/3$, any deterministic algorithm A fails to compute $q(I)$ correctly with probability $\geqslant 1 - 9(4\delta)^{1/\tau^*}$, over random matchings conditioned on $C_{1/3}$. Let $\mathbf{u}^*$ be an edge packing that maximizes $L(\mathbf{u}, \mathbf{M}, p)$, and denote $f = \left(\frac{4L}{(\sum_j u_j^*) \cdot L^{lower}}\right)^{\sum_j u_j^*}$. Lemma 4.2.2 implies that that, for any one-round deterministic algorithm with load $\leqslant L$, $\mathbf{E}[|A(I)|] \leqslant f\mathbf{E}[|q(I)|]$. We prove the following in Section A.1:

Lemma 4.2.4. *If A is a deterministic algorithm for q (more precisely: $\forall I, A(I) \subseteq q(I)$), such that, over random matchings, $\mathbf{E}[|A(I)|] \leqslant f\mathbf{E}[|q(I)|]$ for some constant $f < 1/9$, then, denoting $\texttt{fail}$ the event $A(I) \neq q(I)$, we have*

$$P(\texttt{fail}|C_{1/3}) \geqslant 1 - 9f$$

The proof of the theorem follows from Yao's principle and the fact that $f \leqslant \left(\frac{4\delta}{\sum_j u_j^*}\right)^{1/\sum_j u_j^*} \leqslant (4\delta)^{1/\tau^*}$ because $\sum_j u_j^* \leqslant \tau^*$ and, at optimality, $\sum_j u_j^* \geqslant 1$.

□

In the rest of this section, we give the proof of Theorem 4.2.1.

Let us fix some server $s \in [p]$, and let $\text{msg}(I)$ denote the function specifying the message the server receives on input I. Recall that, in the input-sever model, each input relation S_j is stored at a separate input server, and therefore the message received by s consists of ℓ separate message $\text{msg}_j = \text{msg}_j(S_j)$, for each $j = 1, \ldots, \ell$. One should think of msg_j is a bit string. Once the server s receives msg_j it "knows" that the input relation S_j is in the set $\{S_j \mid \text{msg}_j(S_j) = \text{msg}_j\}$. This justifies the following definition: given a message msg_j, *the set of tuples known by the server* is:

$$K_{\text{msg}_j}(S_j) = \{t \in [n]^{a_j} \mid \text{ for all instances } S_j \subseteq [n]^{a_j}, \text{msg}_j(S_j) = \text{msg}_j \Rightarrow t \in S_j\}$$

where a_j is the arity of S_j.

Clearly, an algorithm A may output a tuple $\mathbf{a} \in [n]^k$ as answer to the query q iff, for every j, $\mathbf{a}_j \in K_{\text{msg}_j}(S_j)$ for all $j = 1, \ldots, \ell$, where $\mathbf{a}_j$ denotes the projection of $\mathbf{a}$ on the variables in the atom S_j.

We will first prove an upper bound for each $|K_{\text{msg}_j}(S_j)|$ in Section 4.2.1. Then in Section 4.2.2 we use this bound, along with Friedgut's inequality, to establish an upper bound for $|K_{\text{msg}}(q)|$ and hence prove Theorem 4.2.1.

4.2.1 Bounding the Knowledge of Each Relation

Let us fix a server s, and an input relation S_j. Recall that $M_j = m_j \log n$ denotes the number of bits necessary to encode S_j. An algorithm A may use few bits, $\mathcal{M}_j$, by exploiting the fact that S_j is a uniformly chosen a_j-dimensional matching. There are precisely $\binom{n}{m_j}^{a_j}(m_j!)^{a_j-1}$ different a_j-dimensional matchings of arity a_j and size m_j and thus the number of bits N

necessary to represent the relation is given by the entropy:

$$\mathcal{M}_j = H(S_j) = a_j \log \binom{n}{m_j} + (a_j - 1)\log(m_j!) \tag{4.4}$$

We will prove later that $\mathcal{M}_j = \Omega(M_j)$. The following lemma provides a bound on the expected knowledge $K_{m_j}(S_j)$ the server may obtain from S_j:

Lemma 4.2.5. *Suppose that the size of S_j is $m_j \leqslant n/2$ (or $m_j = n$), and that the message $msg_j(S_j)$ has at most $f_j \cdot \mathcal{M}_j$ bits. Then $\mathbf{E}[|K_{msg_j}(S_j)|] \leqslant 2f_j \cdot m_j$ (or $\leqslant f_j \cdot m_j$), where the expectation is taken over random choices of the matching S_j.*

It says that, if the message msg_j has only a fraction f_j of the bits needed to encode S_j, then a server receiving this message knows, in expectation, only a fraction $2f_j$ of the m_j tuples in S_j. Notice that the bound holds only in expectation: a specialized encoding may choose to use very few bits to represent a particular matching $S_j \subseteq [n]^{a_j}$: when a server receives that message, then it knows all tuples in S_j, however then there will be fewer bit combinations left to encode the other matchings S_j.

Proof. The entropy $H(S_j)$ in Eq.(4.4) has two parts, corresponding to the two parts needed to encode S_j: for each attribute of S_j we need to encode a subset $\subseteq [n]$ of size m_j, and for each attribute except one we need to encode a permutation over $[m_j]$. Fix a value msg_j of the message received by the server from the input S_j, and let $k = |K_{\text{msg}_j}(S_j)|$. Since msg_j fixes precisely k tuples of S_j, the conditional entropy $H(S_j|\text{msg}_j(S_j) = \text{msg}_j)$ is:

$$\log|\{S_j \mid \text{msg}_j(S_j) = \text{msg}_j\}| \leqslant a_j \log \binom{n-k}{m_j - k} + (a_j - 1)\log((m_j - k))!)$$

We will next show that

$$\log|\{S_j \mid \text{msg}_j(S_j) = \text{msg}_j\}| \leqslant \left(1 - \frac{k}{2m_j}\right)\mathcal{M}_j \tag{4.5}$$

In other words, we claim that the entropy has decreased by at least a fraction $k/(2m_j)$. We show this by proving that each of the two parts of the entropy decreased by that amount:

Proposition 4.2.6. $\log\left((m-k)!\right) \leqslant \left(1-\frac{k}{m}\right)\log(m!)$

Proof. Since $\log(x)$ is an increasing function, it holds that $\sum_{i=1}^{m-k}(\log(i)/(m-k)) \leqslant \sum_{i=1}^{m}(\log(i)/m)$, which is equivalent to:

$$\frac{\log((m-k)!)}{\log(m!)} \leqslant \frac{m-k}{m}$$

This proves the claim.

□

Proposition 4.2.7. *For any $k \leqslant m \leqslant n/2$, or $k \leqslant m = n$:*

$$\log\binom{n-k}{m-k} \leqslant \left(1-\frac{k}{2m}\right)\log\binom{n}{m}$$

Proof. If $m = n$ then the claim holds trivially because both sides are 0, so we assume $m \leqslant n/2$. We have:

$$\frac{\binom{n-k}{m-k}}{\binom{n}{m}} = \frac{m\cdot(m-1)\cdots(m-k+1)}{n\cdot(n-1)\cdots(n-k+1)} \leqslant \left(\frac{m}{n}\right)^k$$

and therefore:

$$\log\binom{n-k}{m-k} \leqslant \log\binom{n}{m} - k\log(n/m) = \left(1-\frac{k\log(n/m)}{\log\binom{n}{m}}\right)\log\binom{n}{m}$$

To conclude the proof, it suffices to show that $\log\binom{n}{m} \leqslant 2m\log(n/m)$. For this, we use the bound $\log\binom{n}{m} \leqslant nH(m/n)$, where $H(x) = -x\log(x) - (1-x)\log(1-x)$ is the *binary entropy*. Denote $f(x) = -x\log(x)$, therefore $H(x) = f(x) + f(1-x)$. Then we have $f(x) \leqslant f(1-x)$ for $x \leqslant 1/2$, because the function $g(x) = f(x) - f(1-x)$ is concave (by direct calculation, $g''(x) = -1/x + 1/(1-x) \leqslant 0$ for $x \in [0, 1/2]$), and $g(0) = g(1/2) = 0$,

meaning that $g(x) \geqslant 0$ on the interval $x \in [0, 1/2]$. Therefore, $H(x) \leqslant 2f(x)$, and our claim follows from:

$$\log \binom{n}{m} \leqslant nH(m/n) \leqslant 2nf(m/n) = 2m \log(n/m)$$

This concludes the proof of Proposition 4.2.7. □

Now we will use Eq.(4.5) to complete the proof of Lemma 4.2.5. We apply the chain rule for entropy, $H(S_j, \text{msg}_j(S_j)) = H(\text{msg}_j(S_j)) + H(S_j|\text{msg}_j(S_j))$, then use the fact that $H(S_j, \text{msg}_j(S_j)) = H(S_j)$ (since S_j completely determines $\text{msg}_j(S_j)$) and apply the definition of $H(S_j|\text{msg}_j(S_j))$:

$$\begin{aligned}
H(S_j) &= H(\text{msg}_j(S_j)) + \sum_{\text{msg}_j} P(\text{msg}_j(S_j) = \text{msg}_j) \cdot H(S_j|\text{msg}_j(S_j) = \text{msg}_j) \\
&\leqslant f_j \cdot H(S_j) + \sum_{\text{msg}_j} P(\text{msg}_j(S_j) = \text{msg}_j) \cdot H(S_j|\text{msg}_j(S_j) = \text{msg}_j) && \text{assumption} \\
&\leqslant f_j \cdot H(S_j) + \sum_{\text{msg}_j} P(\text{msg}_j(S_j) = \text{msg}_j) \cdot (1 - \frac{|K_{\text{msg}_j}(S_j)|}{2m_j}) H(S_j) && \text{Eq.(4.5)} \\
&= f_j \cdot H(S_j) + (1 - \sum_{\text{msg}_j} P(\text{msg}_j(S_j) = \text{msg}_j) \ \frac{|K_{\text{msg}_j}(S_j)|}{2m_j}) H(S_j) \\
&= f_j \cdot H(S_j) + (1 - \frac{\mathbf{E}[|K_{\text{msg}_j(S_j)}(S_j)|]}{2m_j}) H(S_j) && (4.6)
\end{aligned}$$

where the first inequality follows from the assumed upper bound on $|\text{msg}_j(S_j)|$, the second inequality follows by (4.5), and the last two lines follow by definition. Dividing both sides of (4.6) by $H(S_j)$ since $H(S_j)$ is not zero and rearranging we obtain the required statement. □

4.2.2 Bounding the Knowledge of the Query

We use now Lemma 4.2.5 to derive an upper bound on the number of answers to $q(I)$ that a server s can report. Recall that Lemma 4.2.5 assumed that the message $\text{msg}_j(S_j)$ is at most

a fraction f_j of the entropy of S_j. We do not know the values of f_j, instead we know that the entire $\text{msg}(I)$ received by the server s (the concatenation of all ℓ messages $\text{msg}_j(S_j)$) has at most L bits. For each relation S_j, define

$$f_j = \frac{\max_{S_j \subseteq [n]^{a_j}} |\text{msg}_j(S_j)|}{\mathcal{M}_j}.$$

Thus, f_j is the largest fraction of bits of S_j that the server receives, over all choices of the matching S_j. We immediately derive an upper bound on the f_j's. We have $\sum_{j=1}^{\ell} \max_{S_j} |\text{msg}_j(S_j)| \leqslant L$, because each relation S_j can be chosen independently, which implies $\sum_{j=1}^{\ell} f_j \mathcal{M}_j \leqslant L$.

For $\mathbf{a}_j \in [n]^{a_j}$, let $w_j(\mathbf{a}_j)$ denote the probability that the server knows the tuple $\mathbf{a}_j$. In other words $w_j(\mathbf{a}_j) = P(\mathbf{a}_j \in K_{\text{msg}_j(S_j)}(S_j))$, where the probability is over the random choices of S_j.

Lemma 4.2.8. *For any relation S_j:*

(a) $\forall \mathbf{a}_j \in [n]^{a_j} : w_j(\mathbf{a}_j) \leqslant m_j/n^{a_j}$, *and*

(b) $\sum_{\mathbf{a}_j \in [n]^{a_j}} w_j(\mathbf{a}_j) \leqslant 2 f_j \cdot m_j$.

Proof. To show (a), notice that $w_j(\mathbf{a}_j) \leqslant P(\mathbf{a}_j \in S_j) = m_j/n^{a_j}$, while (b) follows from the fact $\sum_{\mathbf{a}_j \in [n]^{a_j}} w_j(\mathbf{a}_j) = \mathbf{E}[|K_{\text{msg}_j(S_j)}(S_j)|] \leqslant 2 f_j \cdot m_j$ (Lemma 4.2.5). □

Since the server receives a separate message for each relation S_j, from a distinct input server, the events $\mathbf{a}_1 \in K_{\text{msg}_1}(S_1), \ldots, \mathbf{a}_\ell \in K_{\text{msg}_\ell}(S_\ell)$ are independent, hence:

$$\mathbf{E}[|K_{\text{msg}(I)}(q)|] = \sum_{\mathbf{a} \in [n]^k} P(\mathbf{a} \in K_{\text{msg}(I)}(q)) = \sum_{\mathbf{a} \in [n]^k} \prod_{j=1}^{\ell} w_j(\mathbf{a}_j)$$

We now use Friedgut's inequality. Recall that in order to apply the inequality, we need to find a fractional edge cover. Let us pick any fractional edge packing $\mathbf{u} = (u_1, \ldots, u_\ell)$. Given

q, defined as in (2.1), consider the *extended query*, which has a new unary atom for each variable x_i:

$$q'(x_1, \ldots, x_k) = S_1(\bar{x}_1), \ldots, S_\ell(\bar{x}_\ell), T_1(x_1), \ldots, T_k(x_k)$$

For each new symbol T_i, define $u'_i = 1 - \sum_{j:x_i \in \text{vars}(S_j)} u_j$. Since $\mathbf{u}$ is a packing, $u'_i \geqslant 0$. Let us define $\mathbf{u}' = (u'_1, \ldots, u'_k)$.

Lemma 4.2.9. *(a) The assignment* $(\mathbf{u}, \mathbf{u}')$ *is both a tight fractional edge packing and a tight fractional edge cover for* q'*. (b)* $\sum_{j=1}^{\ell} a_j u_j + \sum_{i=1}^{k} u'_i = k$

Proof. (a) is straightforward, since for every variable x_i we have $u'_i + \sum_{j:x_i \in \text{vars}(S_j)} u_j = 1$. Summing up:

$$k = \sum_{i=1}^{k} \left(u'_i + \sum_{j:x_i \in \text{vars}(S_j)} u_j \right) = \sum_{i=1}^{k} u'_i + \sum_{j=1}^{\ell} a_j u_j$$

which proves (b). □

We will apply Friedgut's inequality to the extended query q'. Set the variables $w(-)$ used in Friedgut's inequality as follows:

$$\begin{aligned} w_j(\mathbf{a}_j) &= \mathbb{P}(\mathbf{a}_j \in K_{\text{msg}_j(S_j)}(S_j)) \text{ for } S_j, \text{ tuple } \mathbf{a}_j \in [n]^{a_j} \\ w'_i(\alpha) &= 1 \qquad\qquad\qquad\qquad \text{for } T_i, \text{ value } \alpha \in [n] \end{aligned}$$

Recall that, for a tuple $\mathbf{a} \in [n]^k$ we use $\mathbf{a}_j \in [n]^{a_j}$ for its projection on the variables in S_j; with some abuse, we write $\mathbf{a}_i \in [n]$ for the projection on the variable x_i. Assume first that $u_j > 0$, for $j = 1, \ldots, \ell$. Then:

$$\mathbf{E}[|K_{\text{msg}}(q)|] = \sum_{\mathbf{a} \in [n]^k} \prod_{j=1}^{\ell} w_j(\mathbf{a}_j)$$

$$
\begin{aligned}
&= \sum_{\mathbf{a}\in[n]^k} \prod_{j=1}^{\ell} w_j(\mathbf{a}_j) \prod_{i=1}^{k} w'_i(\mathbf{a}_i) \\
&\leqslant \prod_{j=1}^{\ell} \left(\sum_{\mathbf{a}\in[n]^{a_j}} w_j(\mathbf{a})^{1/u_j} \right)^{u_j} \prod_{i=1}^{k} \left(\sum_{\alpha\in[n]} w'_i(\alpha)^{1/u'_i} \right)^{u'_i} \\
&= \prod_{j=1}^{\ell} \left(\sum_{\mathbf{a}\in[n]^{a_j}} w_j(\mathbf{a})^{1/u_j} \right)^{u_j} \prod_{i=1}^{k} n^{u'_i}
\end{aligned}
$$

Note that, since $w'_i(\alpha) = 1$ we have $w'_i(\alpha)^{1/u'_i} = 1$ even if $u'_i = 0$. Write $w_j(\mathbf{a})^{1/u_j} = w_j(\mathbf{a})^{1/u_j - 1} w_j(\mathbf{a})$, and use Lemma 4.2.8 to obtain:

$$
\begin{aligned}
\sum_{\mathbf{a}\in[n]^{a_j}} w_j(\mathbf{a})^{1/u_j} &\leqslant (m_j/n^{a_j})^{1/u_j - 1} \sum_{\mathbf{a}\in[n]^{a_j}} w_j(\mathbf{a}) \\
&\leqslant (m_j n^{-a_j})^{1/u_j - 1} 2 f_j \cdot m_j \\
&= 2 f_j \cdot m_j^{1/u_j} \cdot n^{(a_j - a_j/u_j)}
\end{aligned}
$$

Plugging this in the bound, we have shown that:

$$
\begin{aligned}
\mathbf{E}[|K_{\mathrm{msg}}(q)|] &\leqslant \prod_{j=1}^{\ell} (2 f_j \cdot m_j^{1/u_j} \cdot n^{(a_j - a_j/u_j)})^{u_j} \cdot \prod_{i=1}^{k} n^{u'_i} \\
&= \prod_{j=1}^{\ell} (2 f_j)^{u_j} \cdot \prod_{j=1}^{\ell} m_j \cdot n^{(\sum_{j=1}^{\ell} a_j u_j - a)} \cdot n^{\sum_{i=1}^{k} u'_i} \\
&= \prod_{j=1}^{\ell} (2 f_j)^{u_j} \cdot \prod_{j=1}^{\ell} m_j \cdot n^{-a + (\sum_{j=1}^{\ell} a_j u_j + \sum_{i=1}^{k} u'_i)} \\
&= \prod_{j=1}^{\ell} (2 f_j)^{u_j} \cdot \prod_{j=1}^{\ell} m_j \cdot n^{k-a} \\
&= \prod_{j=1}^{\ell} (2 f_j)^{u_j} \cdot \mathbf{E}[|q(I)|] \qquad (4.7)
\end{aligned}
$$

If some $u_j = 0$, then we can derive the same lower bound as follows: We can replace each u_j with $u_j + \delta$ for any $\delta > 0$ still yielding an edge cover. Then we have $\sum_j a_j u_j + \sum_i u'_i = k + a\delta$,

and hence an extra factor $n^{a\delta}$ multiplying the term $n^{\ell+k-a}$ in (4.7); however, we obtain the same upper bound since, in the limit as δ approaches 0, this extra factor approaches 1.

Let $f_q = \prod_{j=1}^{\ell}(2f_j)^{u_j}$; the final step is to upper bound the quantity f_q using the fact that $\sum_{j=1}^{\ell} f_j \mathcal{M}_j \leqslant L$. Recall that $u = \sum_j u_j$, then:

$$\begin{aligned}
f_q = \prod_{j=1}^{\ell}(2f_j)^{u_j} &= \prod_{j=1}^{\ell}\left(\frac{f_j\mathcal{M}_j}{u_j}\right)^{u_j}\prod_{j=1}^{\ell}\left(\frac{2u_j}{\mathcal{M}_j}\right)^{u_j} \\
&\leqslant \left(\frac{\sum_{j=1}^{\ell} f_j\mathcal{M}_j}{\sum_j u_j}\right)^{\sum_j u_j}\prod_{j=1}^{\ell}\left(\frac{2u_j}{\mathcal{M}_j}\right)^{u_j} \\
&\leqslant \left(\frac{L}{\sum_j u_j}\right)^{\sum_j u_j}\prod_{j=1}^{\ell}\left(\frac{2u_j}{\mathcal{M}_j}\right)^{u_j} \\
&= \prod_{j=1}^{\ell}\left(\frac{2L}{u\cdot\mathcal{M}_j}\right)^{u_j}\prod_{j=1}^{\ell}(u_j)^{u_j} \\
&\leqslant \prod_{j=1}^{\ell}\left(\frac{2L}{u\cdot\mathcal{M}_j}\right)^{u_j}
\end{aligned}$$

Here, the first inequality comes from the weighted version of the Arithmetic Mean-Geometric Mean inequality. The last inequality holds since $u_j \leqslant 1$ for any j.

Finally, we need a lower bound on the number of bits $\mathcal{M}_j$ needed to represent relation S_j. Indeed:

Proposition 4.2.10. *The number of bits $\mathcal{M}_j$ needed to represent S_j are:*

(a) If $n \geqslant m_j^2$, then $\mathcal{M}_j \leqslant M_j/2$

(b) If $n = m_j$ and $a_j \geqslant 2$, then $\mathcal{M}_j \leqslant M_j/4$

Proof. For the first item, we have:

$$\mathcal{M}_j \geqslant a_j \log\binom{n}{m_j} \geqslant a_j m_j \log(n/m_j) \geqslant (1/2)a_j m_j \log(n) = M_j/2$$

For the second item, we have:

$$\mathcal{M}_j \geqslant (a_j - 1)\log(m_j!) \geqslant \frac{a_j - 1}{2} m_j \log(m_j) \geqslant \frac{(a_j - 1)}{2a_j} M_j \geqslant M_j/4$$

where the last inequality comes from the assumption that $a_j \geqslant 2$. □

Applying the above bound on $\mathcal{M}_j$, we complete the proof of Theorem 4.2.1. Recall that our L denotes the load of an arbitrary server, which was denoted L_i in the statement of the theorem.

4.3 Proof of Equivalence

Let $pk(q)$ be the *extreme points* of the convex polytope defined by the fractional edge packing constraints in (2.2). Recall that the vertices of the polytope are feasible solutions $\mathbf{u}_1, \mathbf{u}_2, \ldots$, with the property that every other feasible solution $\mathbf{u}$ to the LP is a convex combination of these vertices. Each vertex can be obtained by choosing m out of the $k+\ell$ inequalities in (2.2), transforming them into equalities, then solving for $\mathbf{u}$. Thus, it holds that $|pk(q)| \leqslant \binom{k+\ell}{m}$. We prove here:

Theorem 4.3.1. *For any vector of statistics* $\mathbf{M}$ *and number of processors* p *, we have:*

$$L^{lower} = L^{upper} = \max_{\mathbf{u} \in pk(q)} L(\mathbf{u}, \mathbf{M}, p)$$

Proof. Recall that $L^{upper} = p^{e^*}$, where e^* is the optimal solution to the *primal* LP problem (4.2). Consider its *dual* LP:

$$\begin{aligned} \text{maximize} \quad & \sum_{j \in [\ell]} \mu_j f_j - f \\ \text{subject to} \quad & \sum_{j \in [\ell]} f_j \leqslant 1 \\ \forall i \in [k] : & \sum_{j : i \in S_j} f_j - f \leqslant 0 \end{aligned}$$

$$\forall j \in [\ell] : f_j \geqslant 0, \quad f \geqslant 0 \tag{4.8}$$

By the primal-dual theorem, its optimal solution is also e^*. Writing $u_j = f_j/f$ and $u = 1/f$, we transform it into the following non-linear optimization problem:

$$\begin{aligned} \text{maximize} \quad & \frac{1}{u} \cdot \left(\sum_{j \in [\ell]} \mu_j u_j - 1 \right) \\ \text{subject to} \quad & \sum_{j \in [\ell]} u_j \leqslant u \\ & \forall i \in [k] : \sum_{j : i \in S_j} u_j \leqslant 1 \\ & \forall j \in [\ell] : u_j \geqslant 0 \end{aligned} \tag{4.9}$$

Consider optimizing the above non-linear problem. Its optimal solution must have $u = \sum_j u_j$, otherwise we simply replace u with $\sum_j u_j$ and obtain a feasible solution with at least as good objective function (indeed, $\mu_j \geqslant 1$ for any j, and hence $\sum_j \mu_j u_j \geqslant \sum_j u_j \geqslant 1$, since any optimal $\mathbf{u}$ will have sum at least 1). Therefore, the optimal is given by a fractional edge packing $\mathbf{u}$. Furthermore, for any packing $\mathbf{u}$, the objective function $\sum_j \frac{1}{u} \cdot (\mu_j u_j - 1)$ is $\log_p L(\mathbf{u}, \mathbf{M}, p)$. To prove the theorem, we show that (a) $e^* = u^*$ and (b) the optimum is obtained when $\mathbf{u} \in pk(q)$. This follows from:

Lemma 4.3.2. *Consider the function $F : \mathbf{R}^{k+1} \to \mathbf{R}^{k+1}$, where $F(x_0, x_1, \ldots, x_k) = (1/x_0, x_1/x_0, \ldots, x_k/x_0)$. Then:*

- *F is its own inverse, $F = F^{-1}$.*
- *F maps any feasible solution to the system (4.8) to a feasible solution to (4.9), and conversely.*
- *F maps a convex set to a convex set.*

Proof. If $y_0 = 1/x_0$ and $y_j = x_j/x_0$, then obviously $x_0 = 1/y_0$ and $x_j = y_j/y$. The second item can be checked directly. For the third item, it suffices to prove that F maps a convex

combination $\lambda\mathbf{x} + \lambda'\mathbf{x}'$ where $\lambda + \lambda' = 1$ into a convex combination $\mu F(\mathbf{x}) + \mu' F(\mathbf{x}')$, where $\mu + \mu' = 1$. Assuming $\mathbf{x} = (x_0, x_1, \ldots, x_k)$ and $\mathbf{x}' = (x'_0, x'_1, \ldots, x'_k)$, this follows by setting $\mu = x_0/(\lambda x_0 + \lambda x'_0)$ and $\mu' = x'_0/(\lambda x_0 + \lambda x'_0)$. □

This completes the proof of Theorem 4.3.1. □

4.4 Discussion

We present here examples and applications of the theorems proved in this section.

4.4.1 The Speedup of the HyperCube

Denote $\mathbf{u}^*$ the fractional edge packing that maximizes $L(\mathbf{u}, \mathbf{M}, p)$ (4.3). When the number of servers increases, the load decreases at a rate of $1/p^{1/\sum_j u_j^*}$, which we call the *speedup* of the HC algorithm. We call the quantity $1/\sum_j u_j^*$ the *speedup exponent*. Ideally, we want to compute a query with linear speedup, which is in this case is equivalent to having $\sum_j u_j^* = 1$, but as we have seen this holds only for very few queries.

We have seen that, when all cardinalities are equal, then the speedup exponent is $1/\tau^*$, but when the cardinalities are unequal then the speedup exponent may be better.

Example 4.4.1. *Consider the triangle query*

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

and assume the relation sizes are M_1, M_2, M_3. Then, $pk(C_3)$ has five vertices, and each gives a different value for $L(\mathbf{u}, \mathbf{M}, p) = (M_1^{u_1} M_2^{u_2} M_3^{u_3}/p)^{1/(u_1+u_2+u_3)}$:

$\mathbf{u}$	$L(\mathbf{u}, \mathbf{M}, p)$
$(1/2, 1/2, 1/2)$	$(M_1 M_2 M_3)^{1/3}/p^{2/3}$
$(1, 0, 0)$	M_1/p
$(0, 1, 0)$	M_2/p
$(0, 0, 1)$	M_3/p
$(0, 0, 0)$	0

(The last row is justified by the fact that $L(\mathbf{u}, \mathbf{M}, p) \leqslant \max(M_1, M_2, M_3)/p^{1/(u_1+u_2+u_3)} \to 0$ *when* $u_1 + u_2 + u_3 \to 0$*.) The load of the HC algorithm is given by the largest of these quantities, in other words, the optimal solution to the LP* (4.2) *that gives the load of the HC algorithm can be given in closed form, as the maximum over these five expressions. To compute the speedup, suppose* $M_1 < M_2 = M_3 = M$*. Then there are two cases. When* $p \leqslant M/M_1$*, the optimal packing is* $(0, 1, 0)$ *(or* $(0, 0, 1)$*) and the load is* M/p*. HyperCube achieves linear speedup by computing a standard join of* $S_2 \bowtie S_3$ *and broadcasting the smaller relation* S_1*; it does this by allocating shares* $p_1 = p_2 = 1$*,* $p_3 = p$*. When* $p > M/M_1$ *then the optimal packing is* $(1/2, 1/2, 1/2)$ *the load is* $(M_1 M_2 M_3)^{1/3}/p^{2/3}$*, and the speedup decreases to* $1/p^{2/3}$*.*

The following lemma sheds some light into how the HyperCube algorithm exploits unequal cardinalities.

Lemma 4.4.2. *Let* q *be a query, over a database with statistics* $\mathbf{M}$*,* $\mathbf{u}^* = argmax_{\mathbf{u}} L(\mathbf{u}, \mathbf{M}, p)$*, and* $L = L(\mathbf{u}^*, \mathbf{M}, p)$*. Then:*

1. *If for some* j*,* $M_j < L$*, then* $u_j^* = 0$*.*
2. *Let* $M = \max_k M_k$*. If for some* j*,* $M_j < M/p$*, then* $u_j^* = 0$*.*
3. *When* p *increases, the speedup exponent remains constant or decreases, eventually reaching* $1/\tau^*$*.*

Proof. We prove the three items of the lemma.

(1) If we modify a fractional edge packing $\mathbf{u}$ by setting $u_j = 0$, we still obtain a fractional edge packing. We claim that the function $f(u_j) = L(\mathbf{u}, \mathbf{M}, p)$ is strictly decreasing in u_j on $(0, \infty)$: the claim implies the lemma because $f(0) > f(u_j)$ for any $u_j > 0$. The claim follows by noticing that $f(u_j) = p^{(u_j \log_p M_j + b)/(u_j + c)}$ where a, b, c are positive constants, hence f is monotone on $u_j \in (0, \infty)$, and $f(u_j) = L > M_j = f(\infty)$, implying that it is monotonically decreasing.

(2) This follows immediately from the previous item by noticing that $M/p \leqslant L$; to see the latter, let k be such that $M_k = M$, and let $\mathbf{u}$ be the packing $u_k = 1$, $u_j = 0$ for $j \neq k$. Then $M/p = L(\mathbf{u}, \mathbf{M}, p) \leqslant L(\mathbf{u}^*, \mathbf{M}, p) = L$.

(3) Consider two edge packings $\mathbf{u}, \mathbf{u}'$, denote $u = \sum_j u_j$, $u' = \sum_j u'_j$, and assume $u < u'$. Let $f(p) = L(\mathbf{u}, \mathbf{M}, p)$ and $g(p) = L(\mathbf{u}', \mathbf{M}, p)$. We have $f(p) = c/p^{1/u}$ and $g(p) = c'/p^{1/u'}$, where c, c' are constants independent of p. Then $f(p) < g(p)$ if and only if $p > (c/c')^{1/(1/u - 1/u')}$, since $1/u - 1/u' > 0$. Thus, as p increases from 1 to ∞, initially we have $f(p) < g(p)$, then $f(p) > g(p)$, and the crossover point is $(c/c')^{1/(1/u-1/u')}$. Therefore, the value $\sum_j u_j^*$ can never decrease, proving the claim. To see that the speedup exponent reaches $1/\tau^*$, denote $\mathbf{u}^*$ the optimal vertex packing (maximizing $\sum_j u_j$) and let $\mathbf{u}$ be any edge packing s.t. $u = \sum_j u_j < \tau^*$. Then, when $p^{1/u - 1/\tau^*} > (\prod_j M_j^{u_j^*})^{1/\tau^*} / (\prod_j M_j^{u_j})^{1/u}$, we have $L(\mathbf{u}^*, \mathbf{M}, p) > L(\mathbf{u}, \mathbf{M}, p)$. □

The first two items in the lemma say that, if M is the size of the largest relation, then the only relations S_j that matter to the HC algorithm are those for which $M_j \geqslant M/p$; any smaller relation will be broadcast by the HC algorithm. The last item says that the HC algorithm can take advantage of unequal cardinalities and achieve speedup better than $1/p^{1/\tau^*}$, *e.g.* by allocating fewer shares to the smaller relations, or even broadcasting them. As p increases, the speedup decreases until it reaches $1/p^{1/\tau^*}$.

4.4.2 Space Exponent

Let $|I| = \sum_j M_j$ denote the size of the input database. Sometimes it is convenient to study algorithms whose maximum load per server is given as $L = O(|I|/p^{1-\varepsilon})$, where $0 \leqslant \varepsilon < 1$ is a constant parameter ε called the *space exponent* of the algorithm. The lower bound given by Theorem 4.2.1 can be interpreted as a lower bound on the space exponent. To see this, consider the special case, when all relations have equal size $M_1 = \ldots = M_\ell = M$; then the load can also be written as $L = O(M/p^{1-\varepsilon})$, and, denoting $\mathbf{u}^*$ the optimal fractional edge packing, we have $\sum_j u_j^* = \tau^*$ and $L(\mathbf{u}^*, \mathbf{M}, p) = M/p^{1/\tau^*}$. Theorem 4.2.1 implies that any algorithm with a fixed space exponent ε will report at most as many answers:

$$O\left(\left(\frac{L}{L(\mathbf{u}^*, \mathbf{M}, p)}\right)^{\tau^*}\right) \cdot \mathbf{E}[|q(I)|] = O(p^{\tau^*[\varepsilon-(1-1/\tau^*)]}) \cdot \mathbf{E}[|q(I)|]$$

Therefore, if the algorithm has a space exponent $\varepsilon < 1 - 1/\tau^*$, then, as p increases, it will return a smaller fraction of the expected number of answers. This supports the intuition that achieving parallelism becomes harder when p increases: an algorithm with a small space exponent may be able to compute the query correctly when p is small, but will eventually fail, when p becomes large enough.

4.4.3 Replication Rate

Given an algorithm that computes a conjunctive query q, let L_s be the load of server s, where $s = 1, \ldots, p$. The *replication rate* r of the algorithm, defined in [14], is $r = \sum_{s=1}^{p} L_i/|I|$. In other words, the replication rate computes how many times on average each input bit is communicated. The authors in [14] discuss the tradeoff between r and the maximum load in the case where the number of servers is not given, but can be chosen optimally. We show next how we can apply our lower bounds to obtain a lower bound for the tradeoff between the replication rate and the maximum load.

Corollary 4.4.3. *Let q be a conjunctive query with statistics $\mathbf{M}$. Any algorithm that com-*

Conjunctive Query	Share Exponents	Value $\tau^*(q)$	Lower Bound for Space Exponent
$C_k(x_1,\ldots,x_k) = \bigwedge_{j=1}^{k} S_j(x_j, x_{(j \bmod k)+1})$	$\frac{1}{k},\ldots,\frac{1}{k}$	$k/2$	$1-2/k$
$T_k(z,x_1,\ldots,x_k) = \bigwedge_{j=1}^{k} S_j(z,x_j)$	$1,0,\ldots,0$	1	0
$L_k(x_0,x_1,\ldots,x_k) = \bigwedge_{j=1}^{k} S_j(x_{j-1},x_j)$	$0,\frac{1}{\lceil k/2\rceil},0,\frac{1}{\lceil k/2\rceil},\ldots$	$\lceil k/2\rceil$	$1-1/\lceil k/2\rceil$
$B_{k,m}(x_1,\ldots,x_k) = \bigwedge_{I\subseteq[k],\lvert I\rvert=m} S_I(\bar{x}_I)$	$\frac{1}{k},\ldots,\frac{1}{k}$	k/m	$1-m/k$

Table 4.1: Query examples: C_k = cycle query, L_k = linear query, T_k = star query, and $B_{k,m}$ = query with $\binom{k}{m}$ relations, where each relation contains a distinct set of m out of the k head variables. The share exponents presented are for the case where the relation sizes are equal.

putes q with maximum load L, where $L \leqslant M_j$ for every S_j[3] must have replication rate

$$r \geqslant \frac{cL}{\sum_j M_j} \max_{\mathbf{u}} \prod_{j=1}^{\ell} \left(\frac{M_j}{L}\right)^{u_j}$$

where $\mathbf{u}$ ranges over all fractional edge packings of q and $c = \max_{\mathbf{u}}(\sum_j u_j/4)^{\sum_j u_j}$.

Proof. Let f_s be the fraction of answers returned by server s, in expectation, where I is a randomly chosen matching database with statistics $\mathbf{M}$. Let $\mathbf{u}$ be an edge packing for q and $c(\mathbf{u}) = (\sum_j u_j/4)^{\sum_j u_j}$; by Theorem 4.2.1, $f_s \leqslant \frac{L_s^{\sum_j u_j}}{c(\mathbf{u})\prod_j M_j^{u_j}}$. Since we assume all answers are returned,

$$1 \leqslant \sum_{s=1}^{p} f_s = \sum_{s=1}^{p} \frac{L_s^{\sum_j u_j}}{c(\mathbf{u})\prod_j M_j^{u_j}} \leqslant \frac{L^{\sum_j u_j - 1}\sum_{s=1}^{p} L_s}{c(\mathbf{u})\prod_j M_j^{u_j}} = \frac{L^{\sum_j u_j - 1} r|I|}{c(\mathbf{u})\prod_j M_j^{u_j}}$$

where we used the fact that $\sum_j u_j \geqslant 1$ for the optimal $\mathbf{u}$. The claim follows by noting that $|I| = \sum_j M_j$. □

[3] if $L > M_j$, we can send the whole relation to any processor without cost

Our lower bounds match the tradeoff for the queries that are discussed in [14], and generalize the tradeoff analysis for all conjunctive queries.

In the specific case where the relation sizes are all equal to M, the above corollary tells us that the replication rate must be $r = \Omega((M/L)^{\tau^*-1})$. Hence, the ideal case of $r = o(1)$ is achieved only when the maximum vertex cover number τ^* is equal to 1 (which happens if and only if a variable occurs in every atom of the query). Another example of the replication rate analysis is presented below:

Example 4.4.4. *Consider again the triangle query C_3 and assume that all sizes are equal to M. In this case, the edge packing that maximizes the lower bound is $(1/2, 1/2, 1/2)$, and $\tau^* = 3/2$. Thus, we obtain an $\Omega(\sqrt{M/L})$ bound for the replication rate for the triangle query.*

Chapter 5

COMPUTING JOIN QUERIES IN ONE STEP WITH SKEW

In this chapter, we focus on the effect of data skew in parallel computation. A value in the database is skewed, and is called a heavy hitter, when its frequency exceeds some predefined threshold. Skew for parallel joins has been studied intensively since the early days of parallel databases, see [79]. The standard parallel join algorithm that handles skew is *the skew join* [66], which consists of first detecting the heavy hitters, then treating them differently from the others values, *e.g.* by partitioning tuples with heavy hitters on the other attributes; a detailed description is in [82]. However, none of these algorithms has been proven to be optimal in any formal sense, and to the best of our knowledge there are no lower bounds for the communication required to compute a join in the presence of skew.

In the previous chapter, we showed tight upper and lower bounds for the load restricted to the case where each value in the input database appears relatively few times. Here we answer the following question: *how can we compute conjunctive queries in one round under the presence of skew?* We give different answers to this question, depending on what kind of information is available to the algorithm that computes the queries. Most of the results in this chapter originally appear in [24, 25].

To go into more detail, let us start by presenting an example where the HC algorithm that uses the optimal shares from (4.2) fails to work when the data has skew.

Example 5.0.5. *Let $q(x, y, z) = S_1(x, z), S_2(y, z)$ be a simple join query, where both relations have cardinality m (and size in bits M). The optimal shares are $p_1 = p_2 = 1$, and $p_3 = p$. This allocation of shares corresponds 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 value of variable z in both S_1 and S_2 is at most m/p), the maximum*

load is $O(M/p)$ with high probability.

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 $\{(1,1),(2,1),\ldots,(m,1)\}$ and so is S_2. In other words, the frequency of value 1 is m in both S_1 and S_2. In this case, the HC algorithm would give a maximum load of $O(M)$.

As we can see from the above example, the problem occurs when the input data contains values with high frequency of occurrence, which we call *outliers*, or *heavy hitters*. In the rest of this chapter, we will consider two different scenarios when handling data skew:

1. In the first scenario, in Section 5.1, we assume that the servers have no information on the input data apart from the size of the relations. We study what parametrization of the HC algorithm gives the best maximum load over all possible distributions of data.

2. In the second scenario, presented in Section 5.2, we assume that the algorithm knows about the outliers in our data. In this case, we can do significantly better than the no information case. We present tight algorithms for the class of star queries and the triangle query, and a general algorithm for all conjunctive queries, which is not optimal.

5.1 The HyperCube Algorithm with Skew

We answer the following question: what are the optimal shares for the HC algorithm such that the maximum load is minimized over all possible distributions of input data? In other words, we limit our treatment to the HyperCube algorithm, but we consider data that can heavily skewed, as in Example 5.0.5. Notice that the HC algorithm is oblivious to the values that are skewed, so it cannot be modified in order to handle these cases separately. Our analysis is based on the following lemma about hashing, which we prove in detail in Section A.2.

Lemma 5.1.1. *Let $R(A_1,\ldots,A_r)$ be a relation of arity r with m tuples. Let $p_1,\ldots,p_r$ be integers and $p = \prod_i p_i$. Suppose that we hash each tuple $(a_1,\ldots,a_r)$ to the bin $(h_1(a_1),\ldots,h_r(a_r))$, where $h_1,\ldots,h_r$ are independent and perfectly random hash functions.*

Then, the probability that the maximum load exceeds $O(m/(\min_i p_i))$ is exponentially small in m.

Corollary 5.1.2. *Let $\mathbf{p} = (p_1, \ldots, p_k)$ be the shares of the HC algorithm. For any relations, with high probability the maximum load per server is*

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

The above bound is tight: we can always construct an instance for given shares such that the maximum load is at least as above. Indeed, for a relation S_j with $i = \arg\min_{i\in S_j} p_i$, we can construct an instance with a single value for any attribute other than x_i, and M_j values for x_i. In this case, the hashing will be across only one dimension with p_i servers, and so the maximum load has to be at least M_j/p_i for the relation S_j.

As in the previous section, if L denotes the maximum load per server, we must have that $M_j/\min_{i\in S_j} p_i \leqslant L$. Denoting $\lambda = \log_p L$ and $\mu_j = \log_p M_j$, the load is optimized by the following LP:

$$\begin{aligned}
\text{minimize} \quad & \lambda \\
\text{subject to} \quad & \sum_{i\in[k]} -e_i \geqslant -1 \\
\forall j \in [\ell] : & h_j + \lambda \geqslant \mu_j \\
\forall j \in [\ell], i \in S_j : & e_i - h_j \geqslant 0 \\
\forall i \in [k] : & e_i \geqslant 0, \quad \forall j \in [\ell] : h_j \geqslant 0 \quad \lambda \geqslant 0
\end{aligned} \tag{5.1}$$

Following the same process as in the previous section, we can obtain the dual of the above LP, and after transformations obtain the following non-linear program with the same optimal objective function:

$$\text{maximize} \quad \frac{\sum_{j\in[\ell]} \mu_j u_j - 1}{\sum_{j\in[\ell]} u_j}$$

$$\begin{aligned} \text{subject to} \quad \forall i \in [k] : \sum_{j:i \in S_j} w_{ij} &\leqslant 1 \\ \forall j \in [\ell] : u_j &\leqslant \sum_{i \in S_j} w_{ij} \\ \forall j \in [\ell] : u_j &\geqslant 0 \\ \forall i \in [k], j \in [\ell] : w_{ij} &\geqslant 0 \end{aligned} \tag{5.2}$$

Example 5.1.3. *We continue the example of the join query $q(x, y, z) = S_1(x, z), S_2(y, z)$, where both relations have m tuples. The optimal solution of* (5.1) *will give us shares $p_x = p_y = p_z = 1/3$, and maximum load of $O(M/p^{1/3})$. Observe that this load guarantee is for any possible data distribution, so even with a highly skewed input distribution, like the one in Example 5.0.5, the HC algorithm will still achieve a load of $O(M/p^{1/3})$ instead of $O(M)$.*

It is possible that some other algorithm than the HC algorithm can do better without any skew information, but we leave this as an open question. Instead, we will next look at how we can do much better when we are provided with information about the highly skewed values in our data.

5.2 Skew with Information

We discuss here the case where there is additional information known about skew in the input database. We will present a general lower bound for arbitrary conjunctive queries, and show an algorithm that matches the bound for *star queries*

$$q = S_1(z, x_1), S_2(z, x_2), \ldots, S_\ell(z, x_\ell)$$

which are a generalization of the join query. We will also present a 1-round optimal algorithm for the triangle query. Finally, we will describe a general algorithmic technique, which we call the BinHC algorithm, presented first in [24], which can be used to compute arbitrary conjunctive queries. However, there is a remaining gap between the upper and lower bounds

in the general case.

We first introduce some necessary notation. For each relation S_j with $|S_j| = m_j$, and each assignment $h \in [n]$ for variable z, we define its *frequency* as $m_j(h) = |\sigma_{z=h}(S_j)|$. We will be interested in assignments that have high frequency, which we call *heavy hitters*. In order to design algorithms that take skew into account, we will assume that every input server knows the assignments with frequency $\geqslant m_j/p$ for every relation S_j, along with their frequency. Because each relation can contain at most p heavy hitters, the total number over all relations will be $O(p)$. Since we are considering cases where the number of servers is much smaller than the data, an $O(p)$ amount of information can be easily stored in the input server with a minimal increase of the load.

To prove the lower bound, we will make a stronger assumption about the information available to the input servers. Given a conjunctive query q, fix a set of variables $\mathbf{x}$ and let $d = |\mathbf{x}|$ Also, let $\mathbf{x}_j = \mathbf{x} \cap \text{vars}(S_j)$ for every relation S_j, and $d_j = |\mathbf{x}_j|$. A *statistics of type* $\mathbf{x}$*, or* $\mathbf{x}$*-statistics* is a vector $m = (m_1, \ldots, m_\ell)$, where m_j is a function $m_j : [n]^{\mathbf{x}_j} \to \mathbb{N}$. We associate with m the function $m : [n]^{\mathbf{x}} \to (\mathbb{N})^\ell$, where $m(\mathbf{h}) = (m_1(\mathbf{h}_1), \ldots, m_\ell(\mathbf{h}_\ell))$, and $\mathbf{h}_j$ denotes the restriction of the tuple $\mathbf{h}$ to the variables in $\mathbf{x}_j$. We say that an instance of S_j satisfies the statistics if for any tuple $\mathbf{h}_j \in [n]^{\mathbf{x}_j}$, its frequency is precisely $m_j(\mathbf{h}_j)$. When $\mathbf{x} = \varnothing$, then m simply consists of ℓ numbers, each representing the cardinality of a relation; thus, a $\mathbf{x}$-statistics generalizes the cardinality statistics. Recall that we use upper case $\mathbf{M} = (M_1, \ldots, M_\ell)$ to denote the same statistics expressed in bits, i.e. $M_j(\mathbf{h}) = a_j m_j(\mathbf{h}) \log(n)$.

5.2.1 Warmup: an Optimal Algorithm for Star Queries

For the case of the star query, we will assume that the input servers know the z-statistics; in other words, for every assignment $h \in [n]$ of variable z, we know that its frequency in relation $S_j(z, x_j)$ is precisely $m_j(h)$. Observe that in this case the cardinality of S_j is $|S_j| = \sum_{h \in [n]} m_j(h)$.

The algorithm uses the same principle popular in virtually all parallel join implementations to date: identify the heavy hitters and treat them differently when distributing the

data. However, the analysis and optimality proof is new, to the best of our knowledge.

Let H denote the set of heavy hitters in all relations. Note that $|H| \leqslant \ell p$. The algorithm will deal with the tuples that have no heavy hitter values (*light tuples*) by running the vanilla HC algorithm, which runs with shares $p_z = p$ and $p_{x_j} = 1$ for every $j = 1, \ldots, \ell$. For this case, the load analysis of Section A.2 will give us a maximum load of $\tilde{O}(\max_j M_j/p)$ with high probability, where $\tilde{O}$ hides a polylogarithmic factor that depends on p. For heavy hitters, we will have to adapt its function as follows.

To compute q, the algorithm must compute for each $h \in H$ the subquery

$$q[h/z] = S_1(h, x_1), \ldots S_k(h, x_k)$$

which is equivalent to computing the cartesian product $q_z = S'_1(x_1), \ldots, S'_k(x_k)$, where $S'_1(x_1) = S_1(h, x_1)$ and $S'_2(x_2) = S_2(h, x_2)$, and each relation S'_j has cardinality $m_j(h)$ (and size in bits $M_j(h)$). We call q_z the *residual query*. The algorithm will allocate p_h servers to compute $q[h/z]$ for each $h \in H$, such that $\sum_{h \in H} p_h = \Theta(p)$. Since the unary relations have no skew, they will be of low degree and thus the maximum load L_h for each h is given by

$$L_h = O\left(\max_{\mathbf{u} \in pk(q_z)} L(\mathbf{u}, \mathbf{M}(h), p_h)\right)$$

For the star query, we have $pk(q_z) = \{0, 1\}^\ell \backslash (0, 0, \ldots, 0)$. At this point, since p_h is not specified, it is not clear which edge packing in $pk(q_z)$ maximizes the above quantity for each h. To overcome this problem, we further refine the assignment of servers to heavy hitters: we allocate $p_{h,\mathbf{u}}$ servers to each h and each $\mathbf{u} \in pk(q_z)$, such that $p_h = \sum_{\mathbf{u}} p_{h,\mathbf{u}}$. Now, for a given $\mathbf{u} \in pk(q_z)$, we can evenly distribute the load among the heavy hitters by allocating servers proportionally to the "heaviness" of executing the residual query. In other words we want $p_{h,\mathbf{u}} \sim \prod_j M_j(h)^{u_j}$ for every $h \in H$. Hence, we will choose:

$$p_{h,\mathbf{u}} = \left\lceil p \cdot \frac{\prod_j M_j(h)^{u_j}}{\sum_{h' \in H} \prod_j M_j(h')^{u_j}} \right\rceil$$

Since $\lceil x \rceil \leqslant x + 1$, and $|H| \leqslant \ell p$, we can compute that the total number of servers we need is at most $(\ell + 1) \cdot |pk(q_z)| \cdot p$, which is $\Theta(p)$. Additionally, the maximum load L_h for every $h \in H$ will be

$$L_h = O\left(\max_{\mathbf{u} \in pk(q_z)} \left(\frac{\sum_{h \in H} \prod_j M_j(h)^{u_j}}{p}\right)^{1/(\sum_j u_j)}\right)$$

Plugging in the values of $pk(q_z)$, we obtain the following upper bound on the algorithm for the heavy hitter case:

$$O\left(\max_{I \subseteq [\ell]} \left(\frac{\sum_{h \in H} \prod_{j \in I} M_j(h)}{p}\right)^{1/|I|}\right) \tag{5.3}$$

Observe that the terms depend on the frequencies of the heavy hitters, and can be much larger than the bound $\tilde{O}(\max_j M_j/p)$ we obtain from the light hitter case. In the extreme, a single heavy hitter h with $m_j(h) = m_j$ for $j = 1, \ldots, \ell$ will demand maximum load equal to $O((\prod_j M_j/p)^{1/\ell})$.

5.2.2 An Optimal Algorithm for Triangle Queries

We show here how to compute the triangle query,

$$q = R(x, y), S(y, z), T(z, x)$$

when all relation sizes are equal to m (and have size in bits M). As with the star query, the algorithm will deal with the tuples that have no heavy hitter values, *i.e.* the frequency is less than $m/p^{1/3}$, by running the vanilla HC algorithm. For this case, we apply the standard analysis to obtain a maximum load of $\tilde{O}(M/p^{2/3})$.

Next, we show how to handle the heavy hitters. We distinguish two different cases.

Case 1. In this case, we handle the tuples that have values with frequency $\geqslant m/p$ in at least two variables. Observe that we did not set the heaviness threshold to $m/p^{1/3}$, for reasons that we will explain in the next case.

Without loss of generality, suppose that both x, y are heavy in at least one of the two relations they belong to. The observation is that there at most p such heavy values for each variable, and hence we can send all tuples of $R(x, y)$ with both x, y heavy (at most p^2) to all servers. Then, we essentially have to compute the query $S'(y, z), T'(z, x)$, where x and y can take only p values. We can do this by computing the join on z; since the frequency of z will be at most p each relation, the maximum load from the join computation will be at most $O(M/p)$.

Case 2. In this case, we handle the remaining output: this includes the tuples where one variable has frequency $\geqslant m/p^{1/3}$, and the other variables are light, *i.e.* have frequency $\leqslant m/p$. Without loss of generality, assume that we want to compute the query q for the values of x that are heavy in either R or T. Observe that there are at most $2p^{1/3}$ of such heavy hitters. If H_x denotes the set of heavy hitter values for variable x, the residual query $q[h/x]$ for each $h \in H$ is:

$$q[h/x] = R(h, y), S(y, z), T(z, h)$$

which is equivalent to computing the query $q_x = R'(y), S(y, z), T'(z)$ with cardinalities $m_R(h), m, m_T(h)$ respectively. As before, we allocate p_h servers to compute $q[h/x]$ for each $h \in H$. If there is no skew, the maximum load L_h is given by the following formula:

$$L_h = O\left(\max\left(\frac{M}{p_h}, \sqrt{\frac{M_R(h)M_T(h)}{p_h}}\right)\right)$$

Notice now that the only cause of skew for q_x may be that y or z are heavy in $S(y, z)$. However, we assumed that the frequencies for both y, z are $\leqslant m/p$, so there will be no skew (this is why we set the heaviness threshold for Case 1 to m/p instead of $m/p^{1/3}$).

We can now set $p_h = p_{h,1} + p_{h,2}$ (for each of the quantities in the max expression), and choose the allocated servers similarly to how we chose for the star queries:

$$p_{h,1} = \left\lceil p \cdot \frac{M_S(h)}{M} \right\rceil \qquad p_{h,2} = \left\lceil p \cdot \frac{M_R(h)M_T(h)}{\sum_{h \in H_x} M_R(h)M_T(h)} \right\rceil$$

We now get a load of:

$$L = O\left(\max\left(\frac{M}{p}, \sqrt{\frac{\sum_h M_R(h)M_T(h)}{p}}\right)\right)$$

Summing up all the cases, we obtain that the load of the 1-round algorithm for computing triangles is:

$$L = \tilde{O}\left(\max\left(\frac{M}{p^{2/3}}, \sqrt{\frac{\sum_h M_R(h)M_T(h)}{p}}, \sqrt{\frac{\sum_h M_R(h)M_S(h)}{p}}, \sqrt{\frac{\sum_h M_S(h)M_T(h)}{p}}\right)\right)$$

5.2.3 The binning algorithm

We now generalize some of the ideas for the simple join to an arbitrary conjunctive query q. Extending the notion for simple joins, for each relation S_j with $|S_j| = m_j$ we say that a partial assignment $\mathbf{h}_j$ to a subset $\mathbf{x}_j \subset vars(S_j)$ is a *heavy hitter* if and only if the number of tuples, $m_j(\mathbf{h}_j)$, from S_j that contain $\mathbf{h}_j$ satisfies $m_j(\mathbf{h}_j) > m_j/p$. As before, there are $O(p)$ such heavy hitters. We will assume that each input server knows the entire set of heavy hitters for all relations.

For simplicity we assume that p is a power of 2. We will not produce quite as smooth a bound as we did for the simple join, but we will show that the bound we produce is within a $\log^{O(1)} p$ factor of the optimal. To do this, for each relation S_j and subset of variables $\mathbf{x}_j$, we define $\log_2 p$ bins for the frequencies, or degrees of each of the heavy hitters. The b-th bin, for $b = 1, \ldots, \log_2 p$ will contain all heavy hitters $\mathbf{h}_j$ with $m_j/2^{b-1} \geqslant m_j(\mathbf{h}_j) > m_j/2^b$. The last bin, a bin of light hitters with $b = \log_2 p + 1$, will contain all assignments $\mathbf{h}_j$ to $\mathbf{x}_j$ that are not heavy hitters. Notice that, when $\mathbf{x}_j = \varnothing$, then the only non-empty bin is the

first bin, the only heavy hitter is the empty tuple $\mathbf{h}_j = ()$, and $m_j(\mathbf{h}_j) = m_j$.

For a bin b on $\mathbf{x}_j$ define $\beta_b = \log_p(2^{b-1})$; observe that for each heavy hitter bin, there are at most $2p^{\beta_b}$ heavy hitters in this bin, and for the last bin we have $\beta_b = 1$. Instead of identifying each bin using its index b, we identify each bin by β_b, called its *bin exponent*, along with the index of the relation S_j for which it is defined, and the set $\mathbf{x}_j \subset vars(S_j)$. Note that $0 = \beta_1 < \beta_2 < \cdots < \beta_{\log_2 p+1} = 1$.

Definition 5.2.1 (Bin Combination)**.** *Let $\mathbf{x} \subset V = vars(q)$, and define $\mathbf{x}_j = \mathbf{x} \cap vars(S_j)$. A pair $\mathcal{B} = (H, (\beta_j)_j)$ is called a* bin combination *if (1) $\beta_j = 0$ for every j where $\mathbf{x}_j = \varnothing$, and (2) there is some consistent assignment $\mathbf{h}$ to $\mathbf{x}$ such that for each j with $\mathbf{x}_j \neq \varnothing$ the induced assignment $\mathbf{h}_j$ to $\mathbf{x}_j$ has bin exponent β_j in relation S_j. We write $C(\mathcal{B})$ for the set of all such assignments $\mathbf{h}$.*

Our algorithm allocates p virtual processors to each bin combination and handles associated inputs separately. There are $O(\log p)$ bin choices for each relation and therefore at most $\log^{O(1)} p$ bin combinations in total, so at most $p \log^{O(1)} p$ virtual processors in total. Let N_{bc} be the number of possible bin combinations. As in the star join algorithm (Subsection 5.2.1), within each bin combination we partition the p servers among the heavy hitters, using $p_{\mathbf{h}} = p^{1-\alpha}$ servers for heavy hitter $\mathbf{h}$ (independent of $\mathbf{h}$, since we have ensured complete uniformity within a bin combination). However, we can only process $p^\alpha \leqslant p$ heavy hitters in every bin combination: in general, we may have $C(\mathcal{B}) > p$, e.g. when $\mathbf{x}$ contains a variable x_1 in S_1 and a variable x_2 in S_2, then there may be up to $p \times p$ heavy hitters in this bin combination.

For a bin combination $\mathcal{B}$ and assignment $\mathbf{h} \in C(\mathcal{B})$, write $m_j(\mathbf{h})$ for $m_j(\mathbf{h}_j)$. For each $\mathcal{B}$ we will define below a set $C'(\mathcal{B}) \subseteq C(\mathcal{B})$ with $|C'(\mathcal{B})| \leqslant p$ and sets $S_j^{(\mathcal{B})} \subseteq S_j$ of tuples for $j \in [\ell]$ that extend $\mathbf{h}_j$ for some $\mathbf{h} \in C'(\mathcal{B})$.

The algorithm for $\mathcal{B}$ will compute all query answers that are joins of $(S_j^{(\mathcal{B})})_j$, by executing the HC algorithm for a particular choice of exponents given. The share exponents for the HC algorithm will be provided by a modification of the vertex covering primal LP (4.2), which

describes an algorithm that suffices for all light hitters. Recall that in this LP, $\mu_j = \log_p M_j$ and λ is $\log_p L$ for the load L. That LP corresponds to the bin combination $\mathcal{B}_\varnothing$ which has $\mathbf{x} = \varnothing$ and all $\beta_j = 0$. In this case $C'(\mathcal{B}_\varnothing) = C(\mathcal{B}_\varnothing)$ has 1 element, the empty partial assignment. More generally, write $\alpha = \log_p |C'(\mathcal{B})|$, the LP associated with our algorithm for bin combination $\mathcal{B}$ is:

$$\begin{aligned} &\text{mininimize} \qquad \lambda \\ &\text{subject to} \\ &\forall j \in [\ell]: \quad \lambda + \sum_{x_i \in vars(S_j) - \mathbf{x}_j} e_i \geqslant \mu_j - \beta_j \\ &\sum_{i \in V - \mathbf{x}} e_i \leqslant 1 - \alpha \\ &\forall i \in V - \mathbf{x}: \quad e_i \geqslant 0, \quad \lambda \geqslant 0 \end{aligned} \tag{5.4}$$

Thus far, this is only fully specified for $\mathcal{B} = \mathcal{B}_\varnothing$ since we have not defined $C'(\mathcal{B})$ for other choices of $\mathcal{B}$. We define $C'(\mathcal{B})$ inductively based on optimal solutions $(\lambda^{(\mathcal{B}')}, (e_i^{(\mathcal{B}')})_{i \in V - \mathbf{x}'})$ to the above LP applied to bin combinations $\mathcal{B}'$ with $\mathbf{x}' \subset \mathbf{x}$ as follows. (Such solutions may not be unique but we fix one arbitrarily for each bin combination.)

For $\mathbf{h}' \in C'(\mathcal{B}')$, we say that a heavy hitter $\mathbf{h}_j$ of S_j that is an extension of $\mathbf{h}'_j$ to $\mathbf{x}_j$ is *overweight* for $\mathcal{B}'$ if there are more than $N_{\text{bc}} \cdot m_j / p^{\beta_j + \sum_{i \in \mathbf{x}_j - \mathbf{x}'_j} e_i^{(\mathcal{B}')}}$ elements of S_j consistent with $\mathbf{h}_j$. $C'(\mathcal{B})$ consists of all assignments $\mathbf{h} \in C(\mathcal{B})$ such that there is some $j \in [\ell]$, some bin combination $\mathcal{B}'$ on set $\mathbf{x}' \subset \mathbf{x}$ such that $\mathbf{x} - \mathbf{x}' \subseteq vars(S_j)$, and some $\mathbf{h}' \in C'(\mathcal{B}')$ such that $\mathbf{h}$ is an extension of $\mathbf{h}'$ and $\mathbf{h}_j$ is an overweight heavy hitter of S_j for $\mathcal{B}'$. The following lemma, which is proved in the appendix, shows that $\alpha \leqslant 1$ and thus that the LP for $\mathcal{B}$ is feasible.

Lemma 5.2.2. *For all bin combinations $\mathcal{B}$, $|C'(\mathcal{B})| \leqslant p$.*

Let $A_\mathcal{B} \subseteq [\ell]$ be the set of all j such that $\mathbf{x}_j \neq \varnothing$. For each $j \in [\ell] - A_\mathcal{B}$, let $S_j^{(\mathcal{B})}$ consist of all tuples in S_j that do not contain any heavy hitter $\mathbf{h}''_j$ of S_j that is overweight for $\mathcal{B}$. For each $j \in A_\mathcal{B}$, and $\mathbf{h} \in C'(\mathcal{B})$ let $S_j^{(\mathcal{B})}(\mathbf{h})$ consist of all tuples in S_j that contain $\mathbf{h}_j$ on $\mathbf{x}_j$

(with bin exponent β_j) but do not contain any heavy hitter $\mathbf{h}''_j$ of S_j that is overweight for $\mathcal{B}$ and a proper extension of $\mathbf{h}_j$. $S_j^{(\mathcal{B})}$ will be the union of all $S_j^{(\mathcal{B})}(\mathbf{h})$ for all $\mathbf{h} \in C'(\mathcal{B})$.

For each of the p^α heavy hitter assignments $\mathbf{h} \in C'(\mathcal{B})$ the algorithm uses $p^{1-\alpha}$ virtual processors to compute the join of the subinstances $S_j^{(\mathcal{B})}(\mathbf{h})$ for $j \in [\ell]$. Those processors will be allocated using the HC algorithm that assigns $p^{e_i^{(\mathcal{B})}}$ shares to each variable $x_i \in V - \mathbf{x}$. It remains to show that the load of the algorithm for $\mathcal{B}$ is within a $\log^{O(1)} p$ factor of $p^{\lambda^{(\mathcal{B})}}$, where $\lambda^{(\mathcal{B})}$ is given by the LP for $\mathcal{B}$.

Lemma 5.2.3. *Let* $\mathbf{h}$ *be an assignment to* $\mathbf{x}$ *that is consistent with bin combination* $\mathcal{B}$*. If we hash each residual relation* $S_j^{(\mathcal{B})}(\mathbf{h})$ *on* $vars(S_j) - \mathbf{x}_j$ *using* $p^{e_i^{(\mathcal{B})}}$ *values for each* $x_i \in vars(S_j) - \mathbf{x}_j$*, each processor receives*

$$O\left((N_{\text{bc}} \cdot \ln p)^{r'} \cdot m_j / p^{\min(\beta_j + \sum_{i \in vars(S_j) - \mathbf{x}_j} e_i^{(\mathcal{B})}, 1)}\right)$$

values with high probability, where $r' = \max_j(r_j - |\mathbf{x}_j|)$.

Proof. For $j \in [\ell] - A_{\mathcal{B}}$, $S_j^{(\mathcal{B})}$ only contains tuples of S_j that are not overweight for $\mathcal{B}$, which means that for every $\mathbf{x}''_j \subseteq vars(S_j)$ and every heavy hitter assignment $\mathbf{h}''$ to the variables of $\mathbf{x}''_j$, there are at most

$$N_{\text{bc}} \cdot m_j / p^{\beta_j + \sum_{i \in \mathbf{x}''_j} e_i^{(\mathcal{B})}} = N_{\text{bc}} \cdot m_j / p^{\beta_j + \sum_{i \in \mathbf{x}''_j - \mathbf{x}_j} e_i^{(\mathcal{B})}}$$

elements of S_j consistent with $\mathbf{h}''$. Every other assignment $\mathbf{h}''$ to the variables of $\mathbf{x}''_j$ is a light hitter and therefore is contained in at most m_j/p consistent tuples of S_j. For $j \in A_{\mathcal{B}}$, we obtain the same bound, where the only difference is that we need to restrict things to extensions of $\mathbf{h}_j$. This bound gives the smoothness condition on $S_j^{(\mathcal{B})}(h)$ necessary to apply Lemma 4.1.2 to each relation $S_j^{(\mathcal{B})}(\mathbf{h})$ and yields the claimed result. □

As a corollary, we obtain:

Corollary 5.2.4. *Let $L_{min} = \max_j(m_j/p)$. The maximum load of our algorithm using p (virtual) processors for $\mathcal{B}$ is $O((N_{\text{bc}} \cdot \ln p)^{r_{\max}} \cdot \max(L_{min}, p^{\lambda}))$ with high probability, where $\lambda = \lambda^{(\mathcal{B})}$ is the optimum of the LP for $\mathcal{B}$ and $r_{\max}$ is the maximum arity of any S_j.*

Proof. There are $p^{1-\alpha}$ processors allocated to each $\mathbf{h}$ and p^{α} such assignments $\mathbf{h}$ so that the maximum load per $\mathbf{h}$ is also the maximum overall load. Given the presence of the L_{min} term, it suffices to show that the maximum load per $\mathbf{h}$ due to relation $S_j^{(\mathcal{B})}(\mathbf{h})$ is at most $(\ln p)^{k-|\mathbf{x}|} \cdot \max(m_j/p, p^{\lambda})$ for each $j \in [\ell]$. Observe that by construction, p^{λ} is the smallest value such that $p^{\lambda} \cdot p^{\beta_j + \sum_{x_i \in vars(S_j) - \mathbf{x}_j} e_i^{(\mathcal{B})}} \geqslant m_j$ for all j and $\sum_{i \in \mathbf{x}} e_i^{(\mathcal{B})} \leqslant 1 - \alpha$. Lemma 5.2.3 then implies that the load due to relation $S_j^{(\mathcal{B})}(\mathbf{h})$ is at most a polylogarithmic factor times

$$\max(m_j/p, m_j/p^{\beta_j + \sum_{x_i \in vars(S_j) - H_j} e_i^{(\mathcal{B})}})$$

which is at most $\max(m_j/p, p^{\lambda})$. □

Lemma 5.2.5. *Every tuple in the join of $(S_j)_{j \in [\ell]}$ is contained in a join of subrelations $(S_j^{(\mathcal{B})})_{j \in [\ell]}$ for some bin combination $\mathcal{B}$.*

Proof. Observe first that every join tuple is consistent with the empty bin combination $\mathcal{B}_{\varnothing}$. Therefore the join of $(S_j^{(\mathcal{B}_{\varnothing})})_{j \in [\ell]}$ contains all join tuples that do not contain an overweight heavy hitter $\mathbf{h}_j$ for any relation S_j with respect to $\mathcal{B}_{\varnothing}$ (and therefore contains all join tuples that are not consistent with any heavy hitter). Now fix a join tuple $\mathbf{t}$ that is overweight for $\mathcal{B}_{\varnothing}$. By definition, there is an associated relation S_{j_1} and $\mathbf{x}^1 \subset vars(S_{j_1})$ such that $\mathbf{h}^1 = (\mathbf{t}_{\mathbf{x}^1})$ is an overweight heavy hitter of S_{j_1} for $\mathcal{B}_{\varnothing}$. Let $\mathcal{B}_1$ be the bin combination associated with $\mathbf{h}^1$. By definition $\mathbf{h}^1 \in C'(\mathcal{B}_1)$. Now either $\mathbf{t}$ is contained in the join of $(S_j^{(\mathcal{B}_1)})_{j \in [\ell]}$ and we are done or there is some relation S_{j_2} and $\mathbf{x}^2$ such that $\mathbf{x}^2 - \mathbf{x}^1 \subset vars(S_{j_2})$ such that $\mathbf{h}^2 = (\mathbf{t}_{\mathbf{x}^2})$ has the property that $\mathbf{h}^2_{j_2}$ is an overweight heavy hitter of S_{j_2} for $\mathcal{B}_1$. Again, in the latter case, if $\mathcal{B}_2$ is the bin combination associated with $\mathbf{h}^2$ then $\mathbf{h}^2 \in C'(\mathcal{B}_2)$ by definition and we can repeat the previous argument for $\mathcal{B}_2$ instead of $\mathcal{B}_1$. Since the number of variables grows at each iteration, we can repeat this at most k times before finding a first $\mathcal{B}_r$ such that $\mathbf{t}$ is

not associated with any overweight heavy hitter for $\mathcal{B}_r$. In this case $\mathbf{t}$ will be computed in the join of $(S_j^{(\mathcal{B}_r)})_{j\in[\ell]}$. □

We can now prove the main theorem.

Theorem 5.2.6. *The algorithm that computes the joins of $(S_j^{(\mathcal{B})})_{j\in[\ell]}$ for every bin combination $\mathcal{B}$ using the HC algorithm as described above has maximum load $L \leqslant \log^{O(1)} p \cdot \max_{\mathcal{B}} p^{\lambda^{(\mathcal{B})}}$.*

Proof. There are only $\log^{O(1)} p$ choices of $\mathcal{B}$ and for each choice of $\mathcal{B}$, by Corollary 5.2.4, the load with p virtual processors is $\log^{O(1)} p \cdot \max(L_{min}, p^{\lambda^{(\mathcal{B})}})$. To derive our claim it suffices to show that we can remove the L_{min} term. Observe that in the original LP which corresponds to an empty bin combination $\mathcal{B}$, we have $\lambda + \sum_{i\in S_j} e_i \geqslant \mu_j$ for each $j \in [\ell]$ and $\sum_i e_i \leqslant 1$. This implies that $\lambda \geqslant \mu_j - 1$ and hence $p^\lambda \geqslant m_j/p$ for each j, so $p^\lambda \geqslant L_{min}$. □

5.2.4 Lower Bound

The lower bound we present here holds for any conjunctive query, and generalizes the lower bound in Theorem 4.2.1, which was over databases with cardinality statistics $\mathbf{M} = (M_1, \ldots, M_\ell)$, to databases with a fixed degree sequence. If the degree sequence is skewed, then the new bounds can be stronger, proving that skew in the input data makes query evaluation harder.

Let us fix statistics $\mathbf{M}$ of type $\mathbf{x}$. We define $q_{\mathbf{x}}$ as the *residual query*, obtained by removing all variables $\mathbf{x}$, and decreasing the arities of S_j as necessary (the new arity of relation S_j is $a_j - d_j$). Clearly, every fractional edge packing of q is also a fractional edge packing of $q_{\mathbf{x}}$, but the converse does not hold in general. If $\mathbf{u}$ is a fractional edge packing of $q_{\mathbf{x}}$, we say that $\mathbf{u}$ *saturates* a variable $x_i \in \mathbf{x}$, if $\sum_{j:x_i\in vars(S_j)} u_j \geqslant 1$; we say that $\mathbf{u}$ saturates $\mathbf{x}$ if it saturates all variables in $\mathbf{x}$. For a given $\mathbf{x}$ and $\mathbf{u}$ that saturates $\mathbf{x}$, define

$$L_{\mathbf{x}}(\mathbf{u}, \mathbf{M}, p) = \left(\frac{\sum_{\mathbf{h}\in[n]^{\mathbf{x}}} \prod_j M_j(\mathbf{h}_j)^{u_j})}{p} \right)^{1/\sum_j u_j} \tag{5.5}$$

Theorem 5.2.7. *Fix statistics* $\mathbf{M}$ *of type* $\mathbf{x}$ *such that* $a_j > d_j$ *for every relation* S_j*. Consider any deterministic MPC algorithm that runs in one communication round on* p *servers and has maximum load* L *in bits. Then, for any fractional edge packing* $\mathbf{u}$ *of* q *that saturates* $\mathbf{x}$*, we must have*

$$L \geqslant \min_j \frac{(a_j - d_j)}{4a_j} \cdot L_{\mathbf{x}}(\mathbf{u}, \mathbf{M}, p).$$

Note that, when $\mathbf{x} = \varnothing$ then $L_{\mathbf{x}}(\mathbf{u}, \mathbf{M}, p) = L(\mathbf{u}, \mathbf{M}, p)$, as defined in (4.3). However, our theorem does not imply Theorem 4.2.1, since it does not give a lower bound on the expected size of the algorithm output as a fraction of the expected output size.

Proof. For $\mathbf{h} \in [n]^{\mathbf{x}}$ and $\mathbf{a}_j \in S_j$, we write $\mathbf{a}_j \parallel \mathbf{h}$ to denote that the tuple $\mathbf{a}_j$ from S_j matches with $\mathbf{h}$ at their common variables $\mathbf{x}_j$, and denote $(S_j)_{\mathbf{h}}$ the subset of tuples $\mathbf{a}_j$ that match $\mathbf{h}$: $(S_j)_{\mathbf{h}} = \{\mathbf{a}_j \mid \mathbf{a}_j \in S_j, \mathbf{a}_j \parallel \mathbf{h}\}$. Let $I_{\mathbf{h}}$ denote the restriction of I to $\mathbf{h}$, in other words $I_{\mathbf{h}} = ((S_1)_{\mathbf{h}}, \ldots, (S_\ell)_{\mathbf{h}})$.

We pick the domain n such that $n = (\max_j\{m_j\})^2$ and construct a probability space for instances I defined by the $\mathbf{x}$-statistics $\mathbf{M}$ as follows. For a fixed tuple $\mathbf{h} \in [n]^{\mathbf{x}}$, the restriction $I_{\mathbf{h}}$ is a uniformly chosen instance over all matching databases with cardinality vector $\mathbf{M}(\mathbf{h})$, which is precisely the probability space that we used in the proof of Theorem 4.3.1. In particular, for every $\mathbf{a}_j \in [n]^{\mathbf{x}_j}$, the probability that S_j contains $\mathbf{a}_j$ is $P(\mathbf{a}_j \in S_j) = m_j(\mathbf{h}_j)/n^{a_j - d_j}$. Lemma 4.2.2 immediately gives:

$$\mathbf{E}[|q(I_{\mathbf{h}})|] = n^{k-d} \prod_{j=1}^{\ell} \frac{m_j(\mathbf{h}_j)}{n^{a_j - d_j}}$$

Let us fix some server and let $\mathrm{msg}(I)$ be the message the server receives on input I. As in the previous section, let $K_{\mathrm{msg}_j}(S_j)$ denote the set of tuples from relation S_j *known by the server*. Let $w_j(\mathbf{a}_j) = P(\mathbf{a}_j \in K_{\mathrm{msg}_j(S_j)}(S_j))$, where the probability is over the random choices of S_j. This is upper bounded by $P(\mathbf{a}_j \in S_j)$:

$$w_j(\mathbf{a}_j) \leqslant m_j(\mathbf{h}_j)/n^{a_j - d_j}, \quad \text{if } \mathbf{a}_j \parallel \mathbf{h} \tag{5.6}$$

We derive a second upper bound by exploiting the fact that the server receives a limited number of bits, in analogy with Lemma 4.2.5:

Lemma 5.2.8. *Let S_j a relation with $a_j > d_j$. Suppose that the size of S_j is $m_j \leqslant n/2$ (or $m_j = n$), and that the message $msg_j(S_j)$ has at most L bits. Then, we have $\mathbf{E}[|K_{msg_j}(S_j)|] \leqslant \frac{4L}{(a_j - d_j)\log(n)}$.*

Observe that in the case where $a_j = d_j$ for some relation S_j, the $\mathbf{x}$-statistics fix all the tuples of the instance for this particular relation, and hence $\mathbf{E}[|K_{\mathrm{msg}_j}(S_j)|] = m_j$.

Proof. We can express the entropy $H(S_j)$ as follows:

$$\begin{aligned} H(S_j) &= H(\mathrm{msg}_j(S_j)) + \sum_{\mathrm{msg}_j} P(\mathrm{msg}_j(S_j) = \mathrm{msg}_j) \cdot H(S_j \mid \mathrm{msg}_j(S_j) = \mathrm{msg}_j) \\ &\leqslant L + \sum_{\mathrm{msg}_j} P(\mathrm{msg}_j(S_j) = \mathrm{msg}_j) \cdot H(S_j \mid \mathrm{msg}_j(S_j) = \mathrm{msg}_j) \end{aligned} \tag{5.7}$$

For every $\mathbf{h} \in [n]^{\mathbf{x}}$, let $K_{\mathrm{msg}_j}((S_j)_{\mathbf{h}})$ denote the known tuples that belong in the restriction of S_j to $\mathbf{h}$. Following the proof of Lemma 4.2.5, and denoting by $\mathcal{M}_j(\mathbf{h}_j)$ the number of bits necessary to represent $(S_j)_{\mathbf{h}}$, we have:

$$\begin{aligned} H(S_j \mid \mathrm{msg}_j(S_j) = \mathrm{msg}_j) &\leqslant \sum_{\mathbf{h} \in [n]^{\mathbf{x}}} \left(1 - \frac{|K_{\mathrm{msg}_j}((S_j)_{\mathbf{h}})|}{2m_j(\mathbf{h}_j)}\right) \mathcal{M}_j(\mathbf{h}_j) \\ &= H(S_j) - \sum_{\mathbf{h} \in [n]^{\mathbf{x}}} \frac{|K_{\mathrm{msg}_j}((S_j)_{\mathbf{h}})|}{2m_j(\mathbf{h}_j)} \mathcal{M}_j(\mathbf{h}_j) \\ &\leqslant H(S_j) - \sum_{\mathbf{h} \in [n]^{\mathbf{x}}} \frac{|K_{\mathrm{msg}_j}((S_j)_{\mathbf{h}})|}{2m_j(\mathbf{h}_j)} m_j(\mathbf{h}_j) \frac{a_j - d_j}{2} \log(n) \\ &= H(S_j) - (1/4) \cdot |K_{\mathrm{msg}_j}(S_j)|(a_j - d_j)\log(n) \end{aligned}$$

where the last inequality comes from Proposition 4.2.10. Plugging this in (5.7), and solving

for $\mathbf{E}[|K_m(S_j)|]$:

$$\mathbf{E}[|K_{\text{msg}_j}(S_j)|] \leqslant \frac{4L}{(a_j - d_j)\log(n)}$$

This concludes our proof. □

Let $q_{\mathbf{x}}$ be the residual query, and recall that $\mathbf{u}$ is a fractional edge packing that saturates $\mathbf{x}$. Define the *extended query* $q_{\mathbf{x}}{}'$ to consist of $q_{\mathbf{x}}$, where we add a new atom $S_i'(x_i)$ for every variable $x_i \in vars(q_{\mathbf{x}})$. Define $u_i' = 1 - \sum_{j:i\in S_j} u_j$. In other words, u_i' is defined to be the slack at the variable x_i of the packing $\mathbf{u}$. The new edge packing $(\mathbf{u}, \mathbf{u}')$ for the extended query $q_{\mathbf{x}}'$ has no more slack, hence it is both a tight fractional edge packing and a tight fractional edge cover for $q_{\mathbf{x}}$. By adding all equalities of the tight packing we obtain:

$$\sum_{j=1}^{\ell}(a_j - d_j)u_j + \sum_{i=1}^{k-d} u_i' = k - d$$

We next compute how many output tuples from $q(I_{\mathbf{h}})$ will be known in expectation by the server. Note that $q(I_{\mathbf{h}}) = q_{\mathbf{x}}(I_{\mathbf{h}})$, and thus:

$$\begin{aligned}
\mathbf{E}[|K_{\text{msg}}(q(I_{\mathbf{h}}))|] &= \mathbf{E}[|K_{\text{msg}}(q_{\mathbf{x}}(I_{\mathbf{h}}))|] \\
&= \sum_{\mathbf{a}\|\mathbf{h}} \prod_{j=1}^{\ell} w_j(\mathbf{a}_j) \\
&= \sum_{\mathbf{a}\|\mathbf{h}} \prod_{j=1}^{\ell} w_j(\mathbf{a}_j) \prod_{i=1}^{k-d} w_i'(\mathbf{a}_i) \\
&\leqslant \prod_{i=1}^{k-d} n^{u_i'} \cdot \prod_{j=1}^{\ell} \left(\sum_{\mathbf{a}_j\|\mathbf{h}} w_j(\mathbf{a}_j)^{1/u_j} \right)^{u_j}
\end{aligned}$$

By writing $w_j(\mathbf{a}_j)^{1/u_j} = w_j(\mathbf{a}_j)^{1/u_j - 1} w_j(\mathbf{a}_j)$ for $\mathbf{a}_j \parallel \mathbf{h}$, we can bound the sum in the above

quantity as follows:

$$\sum_{\mathbf{a}_j \| \mathbf{h}} w_j(\mathbf{a}_j)^{1/u_j} \leqslant \left(\frac{m_j(\mathbf{h}_j)}{n^{a_j - d_j}}\right)^{1/u_j - 1} \sum_{\mathbf{a}_j \| \mathbf{h}} w_j(\mathbf{a}_j) \quad = (m_j(\mathbf{h}_j) n^{d_j - a_j})^{1/u_j - 1} L_j(\mathbf{h})$$

where $L_j(\mathbf{h}) = \sum_{\mathbf{a}_j \| \mathbf{h}} w_j(\mathbf{a}_j)$. Notice that for every relation S_j, we have $\sum_{\mathbf{h}_j \in [n]^{\mathbf{x}_j}} L_j(\mathbf{h}_j) = \sum_{\mathbf{a}_j \in [n]^{a_j}} w_j(\mathbf{a}_j)$. We can now write:

$$\begin{aligned}
\mathbf{E}[|K_{\text{msg}}(q(I_{\mathbf{h}}))|] &\leqslant n^{\sum_{i=1}^{k-d} u'_i} \prod_{j=1}^{\ell} \left(L_j(\mathbf{h}) m_j(\mathbf{h}_j)^{1/u_j - 1} n^{(d_j - a_j)(1/u_j - 1)}\right)^{u_j} \\
&= \prod_{j=1}^{\ell} L_j(\mathbf{h})^{u_j} \cdot \prod_{j=1}^{\ell} m_j(\mathbf{h}_j)^{-u_j} \cdot \mathbf{E}[|q(I_{\mathbf{h}})|] \qquad (5.8)
\end{aligned}$$

Summing over all p servers, we obtain that the expected number of answers that can be output for $q(I_{\mathbf{h}})$ is at most $p \cdot \mathbf{E}[|K_{\text{msg}}(q(I_{\mathbf{h}}))|]$. If some $\mathbf{h} \in [n]^{\mathbf{x}}$ this number is not at least $\mathbf{E}[|q(I_{\mathbf{h}})|]$, the algorithm will fail to compute $q(I)$. Consequently, for every $\mathbf{h}$ we must have that $\prod_{j=1}^{\ell} L_j(\mathbf{h}_j)^{u_j} \geqslant (1/p) \cdot \prod_{j=1}^{\ell} m_j(\mathbf{h}_j)^{u_j}$. Summing the inequalities for every $\mathbf{h} \in [n]^{\mathbf{x}}$:

$$\begin{aligned}
\frac{1}{p} \cdot \sum_{\mathbf{h} \in [n]^{\mathbf{x}}} \prod_{j=1}^{\ell} m_j(\mathbf{h}_j)^{u_j} &\leqslant \sum_{\mathbf{h} \in [n]^{\mathbf{x}}} \prod_{j=1}^{\ell} L_j(\mathbf{h}_j)^{u_j} \\
&\leqslant \prod_{j=1}^{\ell} \left(\sum_{\mathbf{h}_j \in [n]^{\mathbf{x}_j}} L_j(\mathbf{h}_j)\right)^{u_j} && \text{by Friedgut's inequality} \\
&\leqslant \prod_{j=1}^{\ell} \left(\frac{4L}{(a_j - d_j)\log(n)}\right)^{u_j} && \text{by Lemma 5.2.8}
\end{aligned}$$

Solving for L and using the fact that $M_j = a_j m_j \log(n)$, we obtain that for any edge packing $\mathbf{u}$ that saturates $\mathbf{x}$,

$$L \geqslant \left(\min_j \frac{a_j - d_j}{4a_j}\right) \cdot \left(\frac{\sum_{\mathbf{h} \in [n]^{\mathbf{x}}} \prod_j M_j(\mathbf{h}_j)^{u_j}}{p}\right)^{1/\sum_j u_j}$$

which concludes the proof. □

To see how Theorem 5.2.7 applies to the star query, we assume that the input servers know z-statistics $\mathbf{M}$; in other words, for every assignment $h \in [n]$ of variable z, we know that its frequency in relation S_j is $m_j(h)$. Then, for any edge packing $\mathbf{u}$ that saturates z, we obtain a lower bound of

$$L \geqslant (1/8) \cdot \left(\frac{\sum_{h \in [n]} \prod_j M_j(h)^{u_j}}{p} \right)^{1/\sum_j u_j}$$

Observe that the set of edge packings that saturate z and maximize the above quantity is $\{0,1\}^\ell \backslash (0,\ldots,0)$. Hence, we obtain a lower bound

$$L \geqslant (1/8) \cdot \max_{I \subseteq [\ell]} \left(\frac{\sum_{h \in [n]} \prod_{j \in I} M_j(h)}{p} \right)^{1/|I|}$$

A similar argument proves the optimality for the triangle query C_3.

Chapter 6

COMPUTING JOIN QUERIES IN MULTIPLE ROUNDS

In this chapter, we analyze the computation of conjunctive queries in the MPC model for algorithms with multiple rounds. We show that there exists a tradeoff between the number of rounds and the load when computing queries, at least for the case where the input data has no skew: using more rounds, and hence more synchronization, it is possible to achieve a better load per round.

We mainly focus on the following question: given a query q over input data without skew, what is the optimal load to compute q in r rounds, where $r \geqslant 2$? In Section 6.1, we provide both upper and lower bounds for the case of matching databases. In order to prove lower bounds for the case of multiple rounds, we restrict to the tuple-based MPC model, where it is easier to analyze how communication is performed. A surprising consequence of our lower bounds is that computing connected components in a graph of size M cannot be done in a constant number of rounds with load less than M.

In Section 6.2, we briefly explore the computation of conjunctive queries in multiple rounds for any input data. In particular, we present an algorithm for computing the triangle query in 2 rounds with worst-case load that is better than the load that can be achieved in a single round.

6.1 Input Data without Skew

To present and analyze the behavior of multi-round algorithms, we restrict both the structure of the input and the type of computation in the MPC model. In particular, our multi-round algorithms will process only queries where the relations are *of equal size* and the data has *no skew*. Additionally, our lower bounds are proven for a restricted version of the MPC model,

called the the *tuple-based MPC model*, which limits the way communication is performed.

6.1.1 An Algorithm for Multiple Rounds

In Chapter 4, we showed that in the case where all relations have size equal to M and are matching databases (*i.e.* the degree of any value is exactly one), we can compute a conjunctive query q in one round with maximum load

$$L = O(M/p^{1/\tau^*(q)})$$

where $\tau^*(q)$ denotes the fractional vertex covering number of q. Hence, for any $\varepsilon \geqslant 0$, a conjunctive query q with $\tau^*(q) \leqslant 1/(1-\varepsilon)$ can be computed in one round in the MPC model with load $L = O(M/p^{1-\varepsilon})$; recall from Section 4.4 that we call the parameter ε the *space exponent*.

We define now the class of queries Γ^r_ε using induction on r. For $r = 1$, we define

$$\Gamma^1_\varepsilon = \{q \mid \tau^*(q) \leqslant 1/(1-\varepsilon)\}$$

For $r > 1$, we define Γ^r_ε to be the set of all conjunctive queries q constructed as follows. Let $q_1, \ldots, q_m \in \Gamma^{r-1}_\varepsilon$ be m queries, and let $q_0 \in \Gamma^1_\varepsilon$ be a query over a different vocabulary $V_1, \ldots, V_m$, such that $|\text{vars}(q_j)| = \text{arity}(V_j)$ for all $j \in [m]$. Then, the query $q = q_0[q_1/V_1, \ldots, q_m/V_m]$, obtained by substituting each view V_j in q_0 with its definition q_j, is in Γ^r_ε. In other words, Γ^r_ε consists of queries that have a *query plan* of depth r, where each operator is a query computable in one step with maximum load $O(M/p^{1-\varepsilon})$. The following proposition is now straightforward.

Proposition 6.1.1. *Every conjunctive query $q \in \Gamma^r_\varepsilon$ with input a matching database where each relation has size M can be computed by an MPC algorithm in r rounds with maximum load $L = O(M/p^{1-\varepsilon})$.*

We next present two examples that provide some intuition on the structure of the queries

in the class Γ^r_ε.

Example 6.1.2. *Consider the query L_k in Table 4.1 with $k = 16$; we can construct a query plan of depth $r = 2$ and load $L = O(M/p^{1/2})$ (with space exponent $\varepsilon = 1/2$). The first step computes in parallel four queries, $v_1 = S_1, S_2, S_3, S_4, \ldots, v_4 = S_{13}, S_{14}, S_{15}, S_{16}$. Each query is isomorphic to L_4, therefore $\tau^*(q_1) = \cdots = \tau^*(q_4) = 2$ and thus each can be computed in one step with load $L = O(M/p^{1/\tau^*(q_1)}) = O(M/p^{1/2})$. The second step computes the query $q_0 = V_1, V_2, V_3, V_4$, which is also isomorphic to L_4.*

We can generalize the above approach for any query L_k. For any $\varepsilon \geqslant 0$, let k_ε be the largest integer such that $L_{k_\varepsilon} \in \Gamma^\varepsilon_1$. In other words, $\tau^*(L_{k_\varepsilon}) \leqslant 1/(1-\varepsilon)$ and so we choose $k_\varepsilon = 2\lfloor 1/(1-\varepsilon) \rfloor$. Then, for any $k \geqslant k_\varepsilon$, L_k can be computed using L_{k_ε} as a building block at each round: the plan will have a depth of $\lceil \log_{k_\varepsilon}(k) \rceil$ and will achieve a load of $L = O(M/p^{1-\varepsilon})$.

Example 6.1.3. *Consider the query $SP_k = \bigwedge_{i=1}^k R_i(z, x_i), S_i(x_i, y_i)$. Since $\tau^*(SP_k) = k$, the one round algorithm can achieve a load of $O(M/p^{1/k})$.*

However, we can construct a query plan of depth 2 for SP_k with load $O(M/p)$, by computing the joins $q_i = R_i(z, x_i), S_i(x_i, y_i)$ in the first round and in the second round joining all q_i on the common variable z.

We next present an upper bound on the number of rounds needed to compute any query if we want to achieve a given load $L = O(M/p^{1-\varepsilon})$; in other words, we ask what is the minimum number of rounds for which we can achieve a space exponent ε.

Let $\mathrm{rad}(q) = \min_u \max_v d(u, v)$ denote the *radius* of a query q, where $d(u, v)$ denotes the distance between two nodes in the hypergraph of q. For example, $\mathrm{rad}(L_k) = \lceil k/2 \rceil$ and $\mathrm{rad}(C_k) = \lfloor k/2 \rfloor$.

Lemma 6.1.4. *Fix $\varepsilon \geqslant 0$, let $k_\varepsilon = 2\lfloor 1/(1-\varepsilon) \rfloor$, and let q be any connected query. Define*

$$r(q) = \begin{cases} \lceil \log_{k_\varepsilon}(rad(q)) \rceil + 1 & \text{if } q \text{ is tree-like,} \\ \lfloor \log_{k_\varepsilon}(rad(q)) \rfloor + 2 & \text{otherwise.} \end{cases}$$

q query	ε space exponent for 1 round	r rounds to achieve load $O(M/p)$	$r = f(\varepsilon)$ rounds/space tradeoff
C_k	$1 - 2/k$	$\lceil \log k \rceil$	$\sim \frac{\log k}{\log(2/(1-\varepsilon))}$
L_k	$1 - \frac{1}{\lceil k/2 \rceil}$	$\lceil \log k \rceil$	$\sim \frac{\log k}{\log(2/(1-\varepsilon))}$
T_k	0	1	NA
SP_k	$1 - 1/k$	2	NA

Table 6.1: The tradeoff between space and communication rounds for several queries.

Then, q can be computed in $r(q)$ rounds on any matching database input with relations of size M with maximum load $L = O(M/p^{1-\varepsilon})$.

Proof. By definition of $\mathrm{rad}(q)$, there exists some node $v \in \mathrm{vars}(q)$, such that the maximum distance of v to any other node in the hypergraph of q is at most $\mathrm{rad}(q)$. If q is tree-like then we can decompose q into a set of at most $|\mathrm{atoms}(q)|^{\mathrm{rad}(q)}$ (possibly overlapping) paths $\mathcal{P}$ of length $\leqslant \mathrm{rad}(q)$, each having v as one endpoint. Since it is essentially isomorphic to L_ℓ, a path of length $\ell \leqslant \mathrm{rad}(q)$ can be computed in at most $\lceil \log_{k_\varepsilon}(\mathrm{rad}(q)) \rceil$ rounds using the query plan from Proposition 6.1.1 together with repeated use of the one-round HyperCube algorithm for paths of length k_ε. Moreover, all the paths in $\mathcal{P}$ can be computed in parallel, because $|\mathcal{P}|$ is a constant depending only on q. Since every path will contain variable v, we can compute the join of all the paths in one final round with load $O(M/p)$.

The only difference for general connected queries is that q may also contain atoms that join vertices at distance $\mathrm{rad}(q)$ from v that are not on any of the paths of length $\mathrm{rad}(q)$ from v: these can be covered using paths of length $\mathrm{rad}(q) + 1$ from v. To get the final formula, we apply the equality $\lceil \log_a(b+1) \rceil = \lfloor \log_a(b) \rfloor + 1$, which holds for positive integers a, b. □

As an application of the above lemma, Table 6.1 shows the number of rounds required by different types of queries.

6.1.2 Lower Bounds for Multiple Rounds

We present here a general lower bound for connected conjunctive queries in the tuple-based MPC model.

We first introduce a combinatorial object associated with every query q, called the (ε, r)-plan, which is central to the construction of the multi-round lower bound. We next define this notion, and also discuss how we can construct such plans for various classes of queries.

Given a query q and a set $M \subseteq \text{atoms}(q)$, recall that q/M is the query that results from contracting the edges M in the hypergraph of q. Also, we define $\overline{M} = \text{atoms}(q) \backslash M$.

Definition 6.1.5. *Let q be a connected conjunctive query. A set $M \subseteq atoms(q)$ is* ε-good for q *if it satisfies the following two properties:*

1. *Every connected subquery of q that is in Γ^1_ε contains at most one atom in M.*

2. *$\chi(\overline{M}) = 0$ (and thus $\chi(q/\overline{M}) = \chi(q)$ by Lemma 2.1.4).*

For $\varepsilon \in [0, 1)$ and integer $r \geqslant 0$, an (ε, r)-plan $\mathcal{M}$ is a sequence $M_1, \ldots, M_r$, with $M_0 = atoms(q) \supset M_1 \supset \cdots M_r$ such that (a) for $j = 0, \ldots, r-1$, M_{j+1} is ε-good for $q/\overline{M}_j$, and (b) $q/\overline{M}_r \notin \Gamma^1_\varepsilon$.

We provide some intuition about the above definition with the next two lemmas, which shows how we can obtain such a plan for the query L_k and C_k respectively.

Lemma 6.1.6. *The query L_k admits an $(\varepsilon, \lceil \log_{k_\varepsilon}(k) \rceil - 2)$-plan for any integer $k > k_\varepsilon = 2\lfloor 1/(1-\varepsilon) \rfloor$.*

Proof. We will prove using induction that for every integer $r \geqslant 0$, if $k \geqslant k_\varepsilon^{r+1} + 1$ then L_k admits an (ε, r)-plan. This proves the lemma, because then for a given k the smallest integer r we can choose for the plan is $r = \lfloor \log_{k_\varepsilon}(k-1) \rfloor - 1 = \lceil \log_{k_\varepsilon}(k) \rceil - 2$. For the base case $r = 0$, we have that $k \geqslant k_\varepsilon + 1$, and observe that $L_k/\overline{M}_0 = L_k \notin \Gamma^1_\varepsilon$.

For the induction step, let $k_0 \geqslant k_\varepsilon^{r+1} + 1$; then from the inductive hypothesis for every $k \geqslant k_\varepsilon^r + 1$ the query L_k has an $(\varepsilon, r-1)$-plan. Define M to be the set of atoms where we

include every k_ε-th atom L_{k_0}, starting with S_1; in other words, $M = \{S_1, S_{k_\varepsilon+1}, S_{2k_\varepsilon+1}, \ldots\}$. Observe now that $L_{k_0}/\overline{M} = S_1(x_0, x_1), S_{k_\varepsilon+1}(x_1, x_{k_\varepsilon+1}), S_{2k_\varepsilon+1}(x_{k_\varepsilon+1}, x_{2k_\varepsilon+1}), \ldots$, which is isomorphic to $L_{\lceil k_0/k_\varepsilon \rceil}$.

We will show first that M is ε-good for L_{k_0}. Indeed, $\chi(L_{k_0}/\overline{M}) = \chi(L_{\lceil k_0/k_\varepsilon \rceil}) = \chi(L_{k_0})$ and thus property (2) is satisfied. Additionally, recall that Γ^1_ε consists of queries for which $\tau^*(q) \leqslant 1/(1-\varepsilon)$; thus the connected subqueries of L_{k_0} that are in Γ^1_ε are precisely queries of the form $S_j(x_{j-1}, x_j), S_{j+1}(x_j, x_{j+1}), \ldots, S_{j+k-1}(x_{j+k-2}, x_{j+k-1})$, where $k \leqslant k_\varepsilon$. By choosing M to contain every k_ε-atom, no such subquery in Γ^1_ε will contain more than one atom from M and thus property (1) is satisfied as well.

Finally, we have that $\lceil k_0/k_\varepsilon \rceil \geqslant \lceil k^r_\varepsilon + 1/k_\varepsilon \rceil = k^r_\varepsilon + 1$ and thus from the inductive hypothesis the query $L_{k_0}/\overline{M}$ admits an $(\varepsilon, r-1)$-plan. By definition, this implies a sequence $M_1, \ldots, M_{r-1}$; the extended sequence $M, M_1, \ldots, M_{r-1}$ will now be an (ε, r)-plan for L_{k_0}. □

Lemma 6.1.7. *The query C_k admits an $(\varepsilon, \lfloor \log_{k_\varepsilon}(k/(m_\varepsilon+1)) \rfloor)$-plan, for every integer $k > m_\varepsilon = \lfloor 2/(1-\varepsilon) \rfloor$.*

Proof. The proof is similar to the proof for the query L_k, since we can observe that any set M of atoms that are (at least) k_ε apart along any cycle C_k is a ε-good set for C_k and further $C_k/\overline{M}$ is isomorphic to $C_{\lfloor k/k_\varepsilon \rfloor}$. The only difference is that the base case for $r = 0$ is that $k \geqslant m_\varepsilon + 1$. Thus, the inductive step is that for every integer $r \geqslant 0$, if $k \geqslant k^r_\varepsilon(m_\varepsilon + 1)$ then C_k admits an (ε, r)-plan. □

The above examples of queries show how we can construct (ε, r)-plans. We next present the main theorem of this section, which tells us how we can use such plans to obtain lower bounds on the number of communication rounds needed to compute a conjunctive query.

Theorem 6.1.8 (Lower Bound for Multiple Rounds)**.** *Let q be a conjunctive query that admits an (ε, r)-plan. For every randomized algorithm in the tuple-based MPC model that computes q in $r+1$ rounds and with load $L \leqslant cM/p^{1-\varepsilon}$ for a sufficiently small constant c, there exists a (matching) instance I with relations of size M where the algorithm fails to compute q with probability $\Omega(1)$.*

The constant c in the above theorem depends on the query q and the parameter ε. To state the precise expression the constant c, we need some additional definitions.

Definition 6.1.9. *Let q be a conjunctive query and $\mathcal{M}$ be an (ε, r)-plan for q. We define $\tau^*(\mathcal{M})$ to be the minimum of $\tau^*(q/\overline{M}_r)$ and $\tau^*(q')$, where q' ranges over all connected subqueries of $q/\overline{M}_{j-1}$, $j \in [r]$, such that $q' \notin \Gamma^1_\varepsilon$.*

Proposition 6.1.10. *Let q be a conjunctive query and $\mathcal{M}$ be an (ε, r)-plan for q. Then, $\tau^*(\mathcal{M}) > 1/(1-\varepsilon)$.*

Proof. For every $q' \notin \Gamma^1_\varepsilon$, we have by definition that $\tau^*(q') > 1/(1-\varepsilon)$. Additionally, by the definition of an (ε, r)-plan, we have that $\tau^*(q/\overline{M}_r) > 1/(1-\varepsilon)$. □

Further, for a given query q let us define the following sets:

$$\begin{aligned}
\mathcal{C}(q) &= \{q' \mid q' \text{ is a connected subquery of } q\} \\
\mathcal{C}_\varepsilon(q) &= \{q' \mid q' \notin \Gamma^1_\varepsilon,\ q' \text{ is a connected subquery of } q\} \\
\mathcal{S}_\varepsilon(q) &= \{q' \mid q' \notin \Gamma^1_\varepsilon,\ q' \text{ is a minimal connected subquery of } q\}.
\end{aligned}$$

and let

$$\beta(q, \mathcal{M}) = \left(\frac{1}{\tau^*(q/\overline{M}_r)}\right)^{\tau^*(\mathcal{M})} + \sum_{k=1}^{r} \sum_{q' \in \mathcal{S}_\varepsilon(q/\overline{M}_{k-1})} \left(\frac{1}{\tau^*(q')}\right)^{\tau^*(\mathcal{M})}$$

We can now present the precise statement.

Theorem 6.1.11. *If q has an (ε, r)-plan $\mathcal{M}$ then any deterministic tuple-based MPC algorithm running in $r+1$ rounds with maximum load L reports at most*

$$\beta(q, \mathcal{M}) \cdot \left(\frac{(r+1)L}{M}\right)^{\tau^*(\mathcal{M})} p \cdot \mathbf{E}[|q(I)|]$$

correct answers in expectation over a uniformly at random chosen matching database I where each relation has size M.

The above theorem implies Theorem 6.1.8 by following the same proof as in Theorem 4.2.3. Indeed, for $L \leqslant cM/p^{1-\varepsilon}$ we obtain that the output tuples will be at most $f \cdot \mathbf{E}[|q(I)|]$, where $f = \beta(q, \mathcal{M}) \cdot ((r+1)c)^{\tau^*(\mathcal{M})}$. If we choose the constant c such that $f < 1/9$, we can apply Lemma 4.2.4 to show that for any randomized algorithm we can find an instance I where it will fail to produce the output with probability $\Omega(1)$.

In the rest of this section, we present the proof of Theorem 6.1.11. Let $\mathcal{A}$ be an algorithm that computes q in $r+1$ rounds. The intuition is as follows. Consider an ε-good set M; then any matching database i consists of two parts, $i = (i_M, i_{\overline{M}})$,[1] where i_M are the relations for atoms in M, and $i_{\overline{M}}$ are all the other relations. We show that, for a fixed instance $i_{\overline{M}}$, the algorithm can be used to compute $q/\overline{M}(i_M)$ in $r+1$ rounds; however, the first round is almost useless, because the algorithm can discover only a tiny number of join tuples with two or more atoms $S_j \in M$ (since every subquery q' of q that has two atoms in M is not in Γ^1_ε). This shows that the algorithm can compute most of the answers in $q/\overline{M}(i_M)$ in only r rounds, and we repeat the argument until a one-round algorithm remains.

To formalize this intuition, we need some notation. For two relations A, B we write $A \ltimes B$, called the *semijoin*, to denote the set of tuples in A for which there is a tuple in B that has equal values on their common variables. We also write $A \rhd B$, called the *antijoin*, to denote the set of tuples in A for which no tuple in B has equal values on their common variables.

Let $\mathcal{A}$ be a deterministic algorithm with $r+1$ rounds, $k \in [r+1]$ a round number, s a server, and q' a subquery of q. We define:

$$K^{\mathcal{A},k,s}_{\text{msg}}(q') = \{\mathbf{a}' \in [n]^{\text{vars}(q')} \mid \forall \text{ matching database } i, \text{msg}^{\leqslant k}_s(\mathcal{A}, i) = \text{msg} \Rightarrow \mathbf{a}' \in q'(i)\}$$
$$K^{\mathcal{A},k}_{\text{msg}}(q') = \bigcup_{s=1}^{p} K^{\mathcal{A},k,s}_{\text{msg}_s}(q')$$

Using the above notation, $K^{\mathcal{A},k,s}_{\text{msg}^{\leqslant k}_s(\mathcal{A},i)}(q')$ and $K^{\mathcal{A},k}_{\text{msg}^{\leqslant k}(\mathcal{A},i)}(q')$ denote the set of join tuples from q' known at round k by server s, and by all servers, respectively, on input i. Further,

[1] We will use i to denote a fixed matching instance, as opposed to I that denotes a random instance.

$\mathcal{A}(i) = K^{\mathcal{A},r+1}_{\mathrm{msg}^{\leqslant r+1}(\mathcal{A},i)}(q)$ is w.l.o.g. the final answer of the algorithm $\mathcal{A}$ on input i. Finally, let us define

$$J^{\mathcal{A},q}(i) = \bigcup_{q' \in \mathcal{C}(q)} K^{\mathcal{A},1}_{\mathrm{msg}^{\leqslant 1}(\mathcal{A},i)}(q')$$
$$J^{\mathcal{A},q}_{\varepsilon}(i) = \bigcup_{q' \in \mathcal{C}_{\varepsilon}(q)} K^{\mathcal{A},1}_{\mathrm{msg}^{\leqslant 1}(\mathcal{A},i)}(q')$$

$J^{\mathcal{A},q}_{\varepsilon}(i)$ is precisely the set of join tuples known after the first round, but restricted to those that correspond to subqueries that are not computable in one round; thus, the number of tuples in $J^{\mathcal{A},q}_{\varepsilon}(i)$ will be small.

We can now state the two lemmas we need as building blocks to prove Theorem 6.1.11.

Lemma 6.1.12. *Let q be a query, and M be any ε-good set for q. If $\mathcal{A}$ is an algorithm with $r+1$ rounds for q, then for any matching database $i_{\overline{M}}$ over the atoms of $\overline{M}$, there exists an algorithm $\mathcal{A}'$ with r rounds for $q/\overline{M}$ using the same number of processors and the same total number of bits of communication received per processor such that, for every matching database i_M defined over the atoms of M:*

$$|\mathcal{A}(i_M, i_{\overline{M}})| \leqslant |q(i_M, i_{\overline{M}}) \ltimes J^{\mathcal{A},q}_{\varepsilon}(i_M, i_{\overline{M}})| + |\mathcal{A}'(i_M)|.$$

In other words, the algorithm returns no more answers than the (very few) tuples in $J^{\mathcal{A},q}_{\varepsilon}$, plus what another algorithm $\mathcal{A}'$ that we define next computes for $q/\overline{M}$ using *one fewer* round.

Proof. We call $q/\overline{M}$ the *contracted* query. While the original query q takes as input the complete database $i = (i_M, i_{\overline{M}})$, the input to the contracted query is only i_M. Observe also that for different matching databases $i_{\overline{M}}$, the lemma produces different algorithms $\mathcal{A}'$. We fix now a matching database $i_{\overline{M}}$.

The construction of $\mathcal{A}'$ is based on the following two constructions, which we call *contraction* and *retraction*.

Contraction. We first show how to use the algorithm $\mathcal{A}$ to derive an algorithm $\mathcal{A}^c$ for $q/\overline{M}$ that uses the same number of rounds as $\mathcal{A}$.

For each connected component q_c of $\overline{M}$, we choose a representative variable $z_c \in \text{vars}(q_c)$. The query answer $q_c(i_{\overline{M}})$ is a matching instance, since q_c is tree-like (because $\chi(\overline{M}) = 0$). Denote $\mathbf{m}\sigma = \{\sigma_x \mid x \in \text{vars}(q)\}$, where, for every variable $x \in \text{vars}(q)$, $\sigma_x : [n] \to [n]$ is the following permutation. If $x \notin \text{vars}(\overline{M})$, then σ_x is defined as the identity, *i.e.* $\sigma_x(a) = a$ for every $a \in [n]$. Otherwise, if q_c is the unique connected component such that $x \in \text{vars}(q_c)$ and $\mathbf{a} \in q_c(i_{\overline{M}})$ is the unique tuple such that $\mathbf{a}_x = a$, we define $\sigma_x(a) = \mathbf{a}_{z_c}$. In other words, we think of $\mathbf{m}\sigma$ as permuting the domain of each variable $x \in \text{vars}(q)$. Observe that $\mathbf{m}\sigma$ is known to all servers, since $i_{\overline{M}}$ is a fixed instance.

It holds that $\mathbf{m}\sigma(q(i)) = q(\mathbf{m}\sigma(i))$, and $\mathbf{m}\sigma(i_{\overline{M}}) = \mathbf{id}_{\overline{M}}$, where $\mathbf{id}_{\overline{M}}$ is the identity matching database (where each relation in $\overline{M}$ is $\{(1,1,\ldots),(2,2,\ldots),\ldots\}$). Therefore,[2]

$$q/\overline{M}(i_M) = \mathbf{m}\sigma^{-1}(\Pi_{\text{vars}(q/\overline{M})}(q(\mathbf{m}\sigma(i_M), \mathbf{id}_{\overline{M}})))$$

Using the above equation, we can now define the algorithm $\mathcal{A}^c$ that computes the query $q/\overline{M}(i_M)$. First, each input server for $S_j \in M$ replaces S_j with $\mathbf{m}\sigma(S_j)$. Second, we run $\mathcal{A}$ unchanged, substituting all relations $S_j \in \overline{M}$ with the identity. Finally, we apply $\mathbf{m}\sigma^{-1}$ to the answers and return the output. Hence, we have:

$$\mathcal{A}^c(i_M) = \mathbf{m}\sigma^{-1}(\Pi_{\text{vars}(q/\overline{M})}(\mathcal{A}(\mathbf{m}\sigma(i_M), \mathbf{id}_{\overline{M}}))) \tag{6.1}$$

Retraction. Next, we transform $\mathcal{A}^c$ into a new algorithm $\mathcal{A}^r$, called the *retraction* of A^c, that takes as input i_M as follows.

- In round 1, each input server for S_j sends (in addition to the messages sent by $\mathcal{A}^c$)

[2]We assume $\text{vars}(q/\overline{M}) \subseteq \text{vars}(q)$; for that, when we contract a set of nodes of the hypergraph, we replace them with one of the nodes in the set.

every tuple in $\mathbf{a}_j \in S_j$ to all servers s that eventually receive $\mathbf{a}_j$. In other words, the input server sends t to every s for which there exists $k \in [r+1]$ such that $\mathbf{a}_j \in K^{\mathcal{A}^c,k,s}_{\mathrm{msg}_s^{\leqslant k}(\mathcal{A}^c,i_M)}(S_j)$. This is possible because of the restrictions in the tuple-based MPC model: all destinations of $\mathbf{a}_j$ depend only on S_j, and hence can be computed by the input server. Note that this does not increase the total number of bits received by any processor, though it does mean that more communication will be performed during the first round.

- In round 2, $\mathcal{A}^r$ sends *no tuples*.
- In rounds $k \geqslant 3$, $\mathcal{A}^r$ sends a join tuple t from server s to server s' if server s *knows* t at round k, and also algorithm $\mathcal{A}^c$ sends t from s to s' at round k.

Observe first that the algorithm $\mathcal{A}^r$ is correct, in the sense that the output $\mathcal{A}^r(i_M)$ will be a subset of $q/\overline{M}(i_M)$. We now need to quantify how many tuples $\mathcal{A}^r$ misses compared to the contracted algorithm $\mathcal{A}^c$. Let $\mathcal{Q}_M = \{q' \mid q' \text{ subquery of } q/\overline{M}, |q'| \geqslant 2\}$, and define:

$$J_+^{\mathcal{A}^c}(i_M) = \bigcup_{q' \in \mathcal{Q}_M} K^{\mathcal{A}^c,1}_{\mathrm{msg}^1(\mathcal{A}^c,i)}(q').$$

The set $J_+^{\mathcal{A}^c}(i_M)$ is exactly the set of non-atomic tuples known by $\mathcal{A}^c$ right after round 1: these are also the tuples that the new algorithm $\mathcal{A}^r$ will choose not to send during round 2.

Lemma 6.1.13. $(\mathcal{A}^c(i_M) \rhd J_+^{\mathcal{A}^c}(i_M)) \subseteq \mathcal{A}^r(i_M)$

Proof. We will prove the statement by induction on the number of rounds: for any subquery q' of $q/\overline{M}$, if server s knows $t \in (q'(i_M) \rhd J_+^{\mathcal{A}^c}(i_M))$ at round k for algorithm $\mathcal{A}^c$, then server s knows t at round k for algorithm $\mathcal{A}^r$ as well.

For the induction base, in round 1 we have by construction that $K^{\mathcal{A}^c,1,s}_{\mathrm{msg}_s^1(\mathcal{A}^c,i_M)}(S_j) \subseteq K^{\mathcal{A}^r,1,s}_{\mathrm{msg}_s^1(\mathcal{A}^r,i_M)}(S_j)$ for every $S_j \in M$, and thus any tuple t (join or atomic) that is known by server s for algorithm $\mathcal{A}^c$ will be also known for algorithm $\mathcal{A}^r$.

Consider now some round $k+1$ and a tuple $t \in (q'(i_M) \rhd J_+^{\mathcal{A}^c}(i_M))$ known by server s for algorithm $\mathcal{A}^c$. If q' is a single relation, the statement is correct since by construction all

atomic tuples are known at round 1 for algorithm $\mathcal{A}^r$. Otherwise $q' \in \mathcal{Q}_M$. Let $t_1, \ldots, t_m$ be the subtuples at server s from which tuple t is constructed, where $t_j \in q_j(i_M)$ for every $j = 1, \ldots, m$. Observe that $t_j \in (q_j(i_M) \rhd J_+^{\mathcal{A}^c}(i_M))$. Thus, if t_i was known at round k by some server s' for algorithm $\mathcal{A}^c$, by the induction hypothesis it would be known by server s' for algorithm $\mathcal{A}^r$ as well, and thus it would have been communicated to server s at round $k+1$. □

From the above lemma it follows that:

$$\mathcal{A}^c(i_M) \subseteq \mathcal{A}^r(i_M) \cup (q/\overline{M}(i_M) \ltimes J_+^{\mathcal{A}^c}(i_M)) \tag{6.2}$$

Additionally, by the definition of ε-goodness, if a subquery q' of q has two atoms in M, then $q' \notin \Gamma_\varepsilon^1$. Hence, we also have:

$$J_+^{\mathcal{A}^c}(i_M) \subseteq \mathbf{m}\sigma^{-1}(\Pi_{\mathrm{vars}(q/\overline{M})}(J_\varepsilon^{\mathcal{A},q}(\mathbf{m}\sigma(i)))) \tag{6.3}$$

Since $\mathcal{A}^r$ send no information during the second round, we can compress it to an algorithm $\mathcal{A}'$ that uses only r rounds. Finally, since M is ε-good, we have $\chi(q/\overline{M}) = \chi(q)$ and thus $|\mathcal{A}^c(i_M)| = |\mathcal{A}(i_M, i_{\overline{M}})|$. Combining everything together:

$$\begin{aligned}
|\mathcal{A}(i_M, i_{\overline{M}})| &= |\mathcal{A}^c(i_M)| \\
&\leqslant |\mathcal{A}^r(i_M)| + |(q/\overline{M}(i_M) \ltimes J_+^{\mathcal{A}^c}(i_M))| \\
&\leqslant |\mathcal{A}'(i_M)| + |(q/\overline{M}(i_M) \ltimes \mathbf{m}\sigma^{-1}(\Pi_{\mathrm{vars}(q/\overline{M})}(J_\varepsilon^{\mathcal{A},q}(\mathbf{m}\sigma(i)))))| \\
&\leqslant |\mathcal{A}'(i_M)| + |\Pi_{\mathrm{vars}(q/\overline{M})}(q(i) \ltimes J_\varepsilon^{\mathcal{A},q}(i))| \\
&\leqslant \mathcal{A}'(i_M)| + |q(i) \ltimes J_\varepsilon^{\mathcal{A},q}(i)|
\end{aligned}$$

This concludes the proof. □ □

Lemma 6.1.14. *Let q be a conjunctive query and q' a subquery of q. Let $\mathcal{B}$ be any algorithm that outputs a subset of answers to q' (*i.e. *for every database i, $\mathcal{B}(i) \subseteq q'(i)$). Let I be a*

uniformly at random chosen matching database for q, and $I' = I_{atoms(q')}$ its restriction over the atoms in q'.

If $\mathbf{E}[|\mathcal{B}(I')|] \leqslant \gamma \cdot \mathbf{E}[|q'(I')|]$, *then* $\mathbf{E}[|q(I) \ltimes \mathcal{B}(I')|] \leqslant \gamma \cdot \mathbf{E}[|q(I)|]$.

Proof. Let $\mathbf{y} = \text{vars}(q')$ and $d = |\mathbf{y}|$. By symmetry, the quantity $\mathbf{E}[|\sigma_{\mathbf{y}=\mathbf{a}}(q(I))|]$ is independent of $\mathbf{a}$, and therefore equals $\mathbf{E}[|q(I)|]/n^d$. Notice that by construction $\sigma_{\mathbf{y}=\mathbf{a}}(\mathcal{B}(i')) \subseteq \{\mathbf{a}\}$. We now have:

$$\begin{aligned}
\mathbf{E}[|q(I) \ltimes B(I')|] &= \sum_{\mathbf{a} \in [n]^d} \mathbf{E}[|\sigma_{\mathbf{y}=\mathbf{a}}(q(I)) \ltimes \sigma_{\mathbf{y}=\mathbf{a}}(\mathcal{B}(I'))|] \\
&= \sum_{\mathbf{a} \in [n]^d} \mathbf{E}[|\sigma_{\mathbf{y}=\mathbf{a}}(q(I))|] \cdot P(\mathbf{a} \in \mathcal{B}(I')) \\
&= \left(\mathbf{E}[|q(I)|]/n^d\right) \cdot \sum_{\mathbf{a} \in [n]^d} \mathbb{P}(\mathbf{a} \in \mathcal{B}(I')) \\
&= \mathbf{E}[|q(I)|] \cdot \mathbf{E}[|\mathcal{B}(I')|]/n^d \\
&\leqslant \mathbf{E}[|q(I)|] \cdot (\gamma \cdot \mathbf{E}[|q'(I')|])/n^d \\
&\leqslant \gamma \cdot \mathbf{E}[|q(I)|]
\end{aligned}$$

where the last inequality follows from the fact that $\mathbf{E}[|q'(I')|] \leqslant n^d$ (since for every database i', we have $|q'(i')| \leqslant n^d$). □

Proof of Theorem 6.1.11. Given an (ε, r)-plan $\text{atoms}(q) = M_0 \supset M_1 \supset \ldots \supset M_r$, we define $\hat{M}_k = \overline{M}_k - \overline{M}_{k-1}$, for $k \geqslant 1$. Let $\mathcal{A}$ be an algorithm for q that uses $(r+1)$ rounds.

We start by applying Lemma 6.1.12 for algorithm $\mathcal{A}$ and the ε-good set M_1. Then, for every matching database $i_{\bar{M}_1} = i_{\hat{M}_1}$, there exists an algorithm $\mathcal{A}^{(1)}_{i_{\hat{M}_1}}$ for $q/\hat{M}_1$ that runs in r rounds such that for every matching database i_{M_1} we have:

$$|\mathcal{A}(i)| \leqslant |q(i) \ltimes J^{\mathcal{A},q}_{\varepsilon}(i)| + |\mathcal{A}^{(1)}_{i_{\hat{M}_1}}(i_{M_1})|$$

We can iteratively apply the same argument. For $k = 1, \ldots, r-1$, let us denote $\mathcal{B}^k = \mathcal{A}^{(k)}_{i_{\overline{M}_k}}$

the inductively defined algorithm for query $q/\overline{M}_k$, and consider the ε-good set M_{k+1}. Then, for every matching database $i_{\hat{M}_{k+1}}$ there exists an algorithm $\mathcal{B}^{k+1} = \mathcal{A}^{(k+1)}_{i_{\overline{M}_{k+1}}}$ for $q/\overline{M}_{k+1}$ such that for every matching database $i_{M_{k+1}}$, we have:

$$|\mathcal{B}^k(i_{M_k})| \leqslant |q/\overline{M}_k(i_{M_k}) \ltimes J_\varepsilon^{\mathcal{B}^k, q/\overline{M}_k}(i_{M_k})| + |\mathcal{B}^{k+1}(i_{M_{k+1}})|$$

We can now combine all the above inequalities for $k = 0, \ldots, r-1$ to obtain:

$$\begin{aligned}
|\mathcal{A}(i)| \leqslant\ & |q(i) \ltimes J_\varepsilon^{\mathcal{A},q}(i_{M_r}, i_{\hat{M}_1}, \ldots, i_{\hat{M}_r})| \\
& + |q/\overline{M}_1(i_{M_1}) \ltimes J_\varepsilon^{\mathcal{B}^1, q/\overline{M}_1}(i_{M_r}, i_{\hat{M}_2}, \ldots, i_{\hat{M}_r})| \\
& + \ldots \\
& + |q/\overline{M}_{r-1}(i_{M_{r-1}}) \ltimes J_\varepsilon^{\mathcal{B}^{r-1}, q/\overline{M}_{r-1}}(i_{M_r}, i_{\hat{M}_r})| \\
& + |\mathcal{B}^r(i_{M_r})| \qquad (6.4)
\end{aligned}$$

We now take the expectation of (6.4) over a uniformly chosen matching database I and upper bound each of the resulting terms. Observe first that for all $k = 0, \ldots, r$ we have $\chi(q/\overline{M}_k) = \chi(q)$, and hence, by Lemma 4.2.2, we have $\mathbf{E}[|q(I)|] = \mathbf{E}[|(q/\overline{M}_k)(I_{M_k})|]$.

We start by analyzing the last term of the equation, which is the expected output of an algorithm $\mathcal{B}^r$ that uses one round to compute $q/\overline{M}_r$. By the definition of $\tau^*(\mathcal{M})$, we have $\tau^*(q/\overline{M}_r) \geqslant \tau^*(\mathcal{M})$. Since the number of bits received by each processor in the first round of algorithm $\mathcal{B}^r$ is at most $r+1$ times the bound for the original algorithm $\mathcal{A}$, we can apply Theorem 4.2.1 to obtain that:

$$\begin{aligned}
\mathbf{E}[\mathcal{B}^r(I_{M_r})] &\leqslant p\left(\frac{(r+1)L}{\tau^*(q/\overline{M}_r)M}\right)^{\tau^*(q/\overline{M}_r)} \mathbf{E}[|(q/\overline{M}_r)(I_{M_r})| \\
&\leqslant p\left(\frac{(r+1)L}{\tau^*(q/\overline{M}_r)M}\right)^{\tau^*(\mathcal{M})} \mathbf{E}[|q(I)||
\end{aligned}$$

We next bound the remaining terms. Note that $I_{M_{k-1}} = (I_{M_r}, I_{\hat{M}_k}, \ldots, I_{\hat{M}_r})$ and consider

the expected number of tuples in $J = J_\varepsilon^{\mathcal{B}^{k-1}, q/\overline{M}_{k-1}}(I_{M_{k-1}})$. The algorithm $\mathcal{B}^{k-1} = \mathcal{A}^{(k-1)}_{I_{\overline{M}_{k-1}}}$ itself depends on the choice of $I_{\overline{M}_{k-1}}$; still, we show that J has a small number of tuples. Every subquery q' of $q/\overline{M}_{k-1}$ that is not in Γ^1_ε (and hence contributes to J) has $\tau^*(q') \geqslant \tau^*(\mathcal{M})$. For each fixing $I_{\overline{M}_{k-1}} = i_{\overline{M}_{k-1}}$, the expected number of tuples produced for subquery q' by $B_{q'}$, where $B_{q'}$ is the portion of the first round of $\mathcal{A}^{(k-1)}_{i_{\overline{M}_{k-1}}}$ that produces tuples for q', satisfies $\mathbf{E}[|B_{q'}(I_{M_{k-1}})|] \leqslant \gamma(q') \cdot \mathbf{E}[|q'(I_{M_{k-1}})|]$, where

$$\gamma(q') = p \left(\frac{(r+1)L}{\tau^*(q')M} \right)^{\tau^*(\mathcal{M})}$$

since each processor in a round of $\mathcal{A}^{(k-1)}_{i_{\overline{M}_{k-1}}}$ (and hence $B_{q'}$) receives at most $r+1$ times the communication bound for a round of A. We now apply Lemma 6.1.14 to derive

$$\begin{aligned}
\mathbf{E}[|q(I) \ltimes B_{q'}(I_{M_{k-1}})|] &= \mathbf{E}[|(q/\overline{M}_{k-1})(I_{M_{k-1}}) \ltimes B_{q'}(I_{M_{k-1}})|] \\
&\leqslant \gamma(q') \cdot \mathbf{E}[|(q/\overline{M}_{k-1})(I_{M_{k-1}})|] \\
&= \gamma(q') \cdot \mathbf{E}[|q(I)|].
\end{aligned}$$

Averaging over all choices of $I_{\overline{M}_{k-1}} = i_{\overline{M}_{k-1}}$ and summing over the number of different queries $q' \in \mathcal{S}(q/\overline{M}_{k-1})$, where we recall that $\mathcal{S}_\varepsilon(q/\overline{M}_{k-1})$ is the set of all minimal connected subqueries q' of $q/\overline{M}_{k-1}$ that are not in Γ^1_ε, we obtain

$$\mathbf{E}[|q(I) \ltimes J_\varepsilon^{A^{k-1}, q/\overline{M}_{k-1}}(I_{M_{k-1}})|] \leqslant \sum_{q' \in \mathcal{S}_\varepsilon(q/\overline{M}_{k-1})} \gamma(q') \cdot \mathbf{E}[|q(I)|]$$

Combining the bounds obtained for the $r+1$ terms in (6.4), we conclude that $\mathbf{E}[|A(I)|]$ is at most

$$\left(\left(\frac{1}{\tau^*(q/\overline{M}_r)} \right)^{\tau^*(\mathcal{M})} + \sum_{k=1}^{r} \sum_{q' \in \mathcal{S}_\varepsilon(q/\overline{M}_{k-1})} \left(\frac{1}{\tau^*(q')} \right)^{\tau^*(\mathcal{M})} \right) \left(\frac{(r+1)L}{M} \right)^{\tau^*(\mathcal{M})} p \cdot \mathbf{E}[|q(I)|]$$

$$= \beta(q, \mathcal{M}) \cdot \left(\frac{(r+1)L}{M}\right)^{\tau^*(\mathcal{M})} p \cdot \mathbf{E}[|q(I)|]$$

which proves Theorem 6.1.11. □

6.1.3 Application of the Lower Bound

We show now how to apply Theorem 6.1.8 to obtain lower bounds for several query classes, and compare the lower bounds with the upper bounds.

The first class is the queries L_k, where the following corollary is a straightforward application of Theorem 6.1.8 and Lemma 6.1.6.

Corollary 6.1.15. *Any tuple-based MPC algorithm that computes the query L_k (on matching databases) with load $L = O(M/p^{1-\varepsilon})$ requires at least $\lceil \log_{k_\varepsilon} k \rceil$ rounds of computation.*

Observe that this gives a tight lower bound for L_k, since in the previous section we showed that there exists a query plan with depth $\lceil \log_{k_\varepsilon} k \rceil$ and load $O(M/p^{1-\varepsilon})$.

Second, we give a lower bound for tree-like queries, and for that we use a simple observation:

Proposition 6.1.16. *Let q be a tree-like query, and q' be any connected subquery of q. Any algorithm that computes q' with load L needs at least as many rounds to compute q with the same load.*

Proof. Given any tuple-based MPC algorithm A for computing q in r rounds with maximum load L, we construct a tuple-based MPC algorithm A' that computes q' in r rounds with at most load L. A' will interpret each instance over q' as part of an instance for q by using the relations in q' and using the identity permutation ($S_j = \{(1, 1, \ldots), (2, 2, \ldots), \ldots\}$) for each relation in $q \backslash q'$. Then, A' runs exactly as A for r rounds; after the final round, A' projects out for every tuple all the variables not in q'. The correctness of A' follows from the fact that q is tree-like. □

Define $\mathrm{diam}(q)$, the *diameter* of a query q, to be the longest distance between any two nodes in the hypergraph of q. In general, $\mathrm{rad}(q) \leqslant \mathrm{diam}(q) \leqslant 2\ \mathrm{rad}(q)$. For example, $\mathrm{rad}(L_k) = \lfloor k/2 \rfloor$, $\mathrm{diam}(L_k) = k$ and $\mathrm{rad}(C_k) = \mathrm{diam}(C_k) = \lfloor k/2 \rfloor$.

Corollary 6.1.17. *Any tuple-based MPC algorithm that computes a tree-like query q (on matching databases) with load $L = O(M/p^{1-\varepsilon})$ needs at least $\lceil \log_{k_\varepsilon}(diam(q)) \rceil$ rounds.*

Proof. Let q' be the subquery of q that corresponds to the diameter of q. Notice that q' is a connected query, and moreover, it behaves exactly like $L_{\mathrm{diam}(q)}$. Hence, by Corollary 6.1.15 any algorithm needs at least $\lceil \log_{k_\varepsilon}(\mathrm{diam}(q)) \rceil$ to compute q'. By applying Proposition 6.1.16, we have that q needs at least that many rounds as well. □

Let us compare the lower bound $r_{\mathrm{low}} = \lceil \log_{k_\varepsilon}(\mathrm{diam}(q)) \rceil$ and the upper bound $r_{\mathrm{up}} = \lceil \log_{k_\varepsilon}(\mathrm{rad}(q)) \rceil + 1$ from Lemma 6.1.4. Since $\mathrm{diam}(q) \leqslant 2\mathrm{rad}(q)$, we have that $r_{\mathrm{low}} \leqslant r_{\mathrm{up}}$. Additionally, $\mathrm{rad}(q) \leqslant \mathrm{diam}(q)$ implies $r_{\mathrm{up}} \leqslant r_{\mathrm{low}} + 1$. Thus, the gap between the lower bound and the upper bound on the number of rounds is at most 1 for tree-like queries. When $\varepsilon < 1/2$, these bounds are matching, since $k_\varepsilon = 2$ and $2\mathrm{rad}(q) - 1 \leqslant \mathrm{diam}(q)$ for tree-like queries.

Third, we study one instance of a non tree-like query, namely the cycle query C_k. The lemma is a direct application of Lemma 6.1.7.

Lemma 6.1.18. *Any tuple-based MPC algorithm that computes the query C_k (on matching databases) with load $L = O(M/p^{1-\varepsilon})$ requires at least $\lfloor \log_{k_\varepsilon}(k/(m_\varepsilon + 1)) \rfloor + 2$ rounds, where $m_\varepsilon = \lfloor 2/(1-\varepsilon) \rfloor$.*

For cycle queries we also have a gap of at most 1 between this lower bound and the upper bound in Lemma 6.1.4.

Example 6.1.19. *Let $\varepsilon = 0$ and consider two queries, C_5 and C_6. In this case, we have $k_\varepsilon = m_\varepsilon = 2$, and $rad(C_5) = rad(C_6) = 2$.*

For query C_6, the lower bound is then $\lfloor \log_2(6/3) \rfloor + 2 = 3$ rounds, while the upper bound is $\lceil \log_2(3) \rceil + 1 = 3$ rounds. Hence, in the case of C_6 we have tight upper and lower bounds.

For query C_5, the upper bound is again $\lceil \log_2(3) \rceil + 1 = 3$ rounds, but the lower bound becomes $\lfloor \log_2(5/3) \rfloor + 2 = 2$ rounds. The exact number of rounds necessary to compute C_5 is thus open.

Finally, we show how to apply Corollary 6.1.15 to show that transitive closure requires many rounds. In particular, we consider the problem Connected-Components, for which, given an undirected graph $G = (V, E)$ with input a set of edges as a relation $R(x, y)$, the requirement is to label the nodes of each connected component with the same label, unique to that component. More formally, we want to compute an output relation $O(x, c)$, such that any two vertices x, y have the same label c if and only if they belong in the same connected component.

Theorem 6.1.20. *Let G be an input graph of size M. For any $\varepsilon < 1$, there is no algorithm in the tuple-based MPC model that computes* Connected-Components *with p processors and load $L = O(M/p^{1-\varepsilon})$ in fewer than $o(\log p)$ rounds.*

The idea of the proof is to construct input graphs for Connected-Components whose components correspond to the output tuples for L_k for $k = p^\delta$ for some small constant δ depending on ε and use the round lower bound for solving L_k. Notice that the size of the query L_k is not fixed, but depends on the number of processors p.

Proof. Since larger ε implies a more powerful algorithm, we assume without loss of generality that $\varepsilon = 1 - 1/t$ for some integer constant $t > 1$. Let $\delta = 1/(2t(t+2))$. The family of input graphs and the initial distribution of the edges to servers will look like an input to L_k, where $k = \lfloor p^\delta \rfloor$. In particular, the vertices of the input graph G will be partitioned into $k+1$ sets $P_1, \ldots, P_{k+1}$, each partition containing m/k vertices. The edges of G will form permutations (matchings) between adjacent partitions, P_i, P_{i+1}, for $i = 1, \ldots, k$. Thus, G will contain exactly $k \cdot (m/k) = m$ edges. This construction creates essentially k binary relations, each with m/k tuples and size $M_k = (m/k)\log(m/k)$.

Since $k < p$, we can assume that the adversary initially places the edges of the graph so that each server is given edges only from one relation. It is now easy to see that any tuple-

based algorithm in MPC that solves Connected-Components for an arbitrary graph G of the above family in r rounds with load L implies an $(r+1)$-round tuple-based algorithm with the same load that solves L_k when each relation has size M. Indeed, the new algorithm runs the algorithm for connected components for the first r rounds. It then uses an additional round to perform a k-way star join on the labels of the vertices so as to obtain the output of the join query L_k (Since each tuple in L_k corresponds exactly to a connected component in G, the join will recover all the tuples of L_k.) This can be achieved in a single round by hashing each tuple $O(x, c)$ according to c; because each label occurs exactly p^δ times, there will be no skew during hashing and the load for the final round will be $O(M/p)$.

Since the query size is not independent of the number of servers p, we have to carefully compute the constants for our lower bounds. Consider an algorithm for L_k with load $L \leqslant cM/p^{1-\varepsilon}$, where $M = m\log(m)$. Let $r = \lceil \log_{k_\varepsilon} k \rceil - 2$. Observe also that $k_\varepsilon = 2t$ since $\varepsilon = 1 - 1/t$.

We will use the (ε, r)-plan $\mathcal{M}$ for L_k presented in the proof of Lemma 6.1.6, apply Theorem 6.1.11, and compute the factor $\beta(L_k, \mathcal{M})$. First, notice that each query $L_k/\overline{M}_j$ for $j = 0, \ldots, r$ is isomorphic to L_{k/k_ε^j}. Then, the set $\mathcal{S}_\varepsilon(L_{k/k_\varepsilon^j})$ consists of at most k/k_ε^j paths q' of length $k_\varepsilon + 1$. By the choice of r, $L_k/\overline{M}_r$ is isomorphic to L_ℓ where $k_\varepsilon + 1 \leqslant \ell < k_\varepsilon^2$. Further, we have that $\tau^*(\mathcal{M}) = \tau^*(L_{k_\varepsilon+1}) = \lceil (k_\varepsilon + 1)/2 \rceil = t + 1$ since $k_\varepsilon = 2t$.

Thus, we have

$$\begin{aligned}
\beta(L_k, \mathcal{M}) &= \left(\frac{1}{\tau^*(L_k/\overline{M}_r)}\right)^{\tau^*(\mathcal{M})} + \sum_{j=1}^{r} \sum_{q' \in \mathcal{S}_\varepsilon(q/\overline{M}_{k-1})} \left(\frac{1}{\tau^*(q')}\right)^{\tau^*(\mathcal{M})} \\
&\leqslant (1-\varepsilon)^{\tau^*(\mathcal{M})} \left(1 + \sum_{j=1}^{r} \frac{k}{k_\varepsilon^{j-1}}\right) \\
&\leqslant (2k+1)(1-\varepsilon)^{\tau^*(\mathcal{M})}.
\end{aligned}$$

Observe now that $M/M_k = 1/(1/k - \log(k)/(k\log(m))) \leqslant 2k$, assuming that $m \geqslant p^{2\delta}$. Consequently, Theorem 6.1.11 implies that any tuple-based MPC algorithm using at most

$\lceil \log_{k_\varepsilon} k \rceil - 1$ rounds reports at most the following fraction of the required output tuples for the L_k query:

$$\begin{aligned}
\beta(L_k, \mathcal{M}) \cdot p \left(\frac{(r+1)L}{M_k} \right)^{\tau^*(\mathcal{M})} &\leqslant (2k+1)(2ck(r+1)/t)^{\tau^*(\mathcal{M})} \cdot p^{1-\tau^*(\mathcal{M})(1-\varepsilon)} \\
&\leqslant c' k^{t+2} (\log_2 k)^{c''} \cdot p^{1-(1+t)(1-\varepsilon)} \\
&\leqslant c' (\delta \log_2 p)^{c''} \cdot p^{\delta(t+2)+1-(1+t)(1-\varepsilon)} \\
&= c' (\delta \log_2 p)^{c''} \cdot p^{\delta(t+2)-1/t} \\
&= c' (\delta \log_2 p)^{c''} \cdot p^{-1/2t}
\end{aligned}$$

where c, c'' are constants. Since $t > 1$, the fraction of the output tuples is $o(1)$ as a function of the number of processors p. This implies that any algorithm that computes CONNECTED-COMPONENTS on G requires at least $\lceil \log_{k_\varepsilon} \lfloor p^\delta \rfloor \rceil - 2 = \Omega(\log p)$ rounds. □

6.2 Input Data with Skew

In this section, we briefly discuss the case of input data with skew. We first present a back-of-the-envelope calculation on the lower bound for the tuple-based MPC model, in the scenario where all relations have size equal to M.

Given a query q, we can construct a worst-case instance I with the maximum possible output, according to the construction of [22]. We know then that the output will be M^{ρ^*}, where ρ^* is the maximum edge cover for q. Now, assume that algorithm $\mathcal{A}$ computes q with load L (in tuples) in r rounds. Since each server receives at most $r \cdot L$ tuples from each relation S_j, we can use the AGM bound from [22] to argue that the total number of output tuples will be at most $p(r \cdot L)^{\rho^*}$. Hence, if $\mathcal{A}$ outputs all tuples, we must have that $p(r \cdot L)^{\rho^*} \geqslant M^{\rho^*}$, or equivalently $L \geqslant M/(rp^{1/\rho^*})$. We should note here that this is not a formal proof, and that also we have considered only deterministic algorithms; we present this idea in order to give the reader some intuition about what kind of lower bounds one should expect in the worst case. For the example of the triangle query C_3, since the maximum edge

cover is $\rho^* = 3/2$, we obtain a lower bound $\Omega(M/p^{2/3})$ for a constant number of rounds.

We will next show that we can in fact match this informal bound for the triangle query

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

in 2 rounds, up to logarithmic factors. We will present an algorithm that achieves an $\tilde{O}(M/p^{2/3})$ load under any input distribution, in the case where all relations have the same size M. This is a *worst-case analysis*, in the sense that there will be inputs where we will be able to do better (in the previous section we showed that we can compute C_3 on matching databases in 2 rounds with load $O(M/p)$), but we can guarantee that with any input we can do at least as well as the upper bound of $\tilde{O}(M/p^{2/3})$.

To build the algorithm, we need a result on computing the join $S_1(x, z), S_2(y, z)$ in a single round, for the case where skew appears only in one of the two relations.

Lemma 6.2.1. *Let $q = S_1(x, z), S_2(y, z)$, and let M_1 and M_2 be the relation sizes of S_1, S_2 respectively. Let $M = \max\{M_1, M_2\}$. If the degree of every value of the variable z in S_1 is at most M/p, then we can compute q in a single round with load $\tilde{O}(M/p)$.*

Proof. The algorithm is based on the ideas that we presented in Subsection 5.2.1 for computing star queries with skew. We say that a value h is a heavy hitter in S_2 if the degree of h is $M_{S_2}(z) \geqslant M/p$. By our assumption, there are no heavy hitter values in S_1.

For the values h that are not heavy hitters in S_2, we can compute the join by applying the vanilla HC algorithm; the load analysis of Section A.2 will give us a load of $\tilde{O}(M/p)$ with high probability in this case.

For every heavy hitter h, the algorithm computes the subquery $q[h/z] = S_1(x, h), S_2(y, h)$, which is equivalent to computing the *residual query* $q_z = S_1'(x), \ldots, S_2'(y)$, where $S_1'(x) = S_1(x, h)$ and $S_2'(y) = S_2(y, h)$. We know that $|S_2'| = M_{S_2}(h)$ and $|S_1'| \leqslant M/p$ by our assumption. The algorithm now allocates $p_h = \lceil p \cdot M_{S_2}(h)/M \rceil$ exclusive servers to compute $q[h/z]$ for each heavy hitter h. To compute $q[h/z]$ with p_h we simply use the simple broadcast

join that assigns a share of 1 to x and 0 to y. A simple analysis will give us that the load for each heavy hitter h is

$$\tilde{O}\left(\frac{|S_2'|}{p_h} + |S_1'|\right) = \tilde{O}\left(\frac{M_{S_2}(h)}{pM_{S_2}(h)/M} + M/p)\right) = \tilde{O}(M/p)$$

Finally, observe that $\sum_h p_h \leqslant 2p$, hence we have used an appropriate amount of available servers. □

Hence, we can optimally compute joins in a single round, even in the presence of one-sided skew. We can now present the main algorithm for computing triangles.

Proposition 6.2.2. *The triangle query $C_3 = S_1(x_1, x_2), S_2(x_2, x_3), S_3(x_3, x_1)$ on input with sizes $M_1 = M_2 = M_3 = M$ can be computed by an MPC algorithm in 2 rounds with $\tilde{O}(M/p^{2/3})$ maximum load, under any input data distribution.*

Proof. We say that a value h is heavy if for some relation S_j, we have $M_j(h) > M/p^{1/3}$. We first compute the answers for the tuples that have no heavy values. Indeed, if for every value we have that the degree is at most $M/p^{1/3}$, then the load analysis of Section A.2 tells us that we can compute the output in a single round with load $\tilde{O}(M/p^{2/3})$ using the HC algorithm.

Thus, it remains to output the tuples for which at least one variable has a heavy value. Without loss of generality, consider the case where variable x_1 has heavy values and observe that there are at most $p^{1/3}$ heavy x_1-values. Let $R_1'(x_1, x_2)$ be the subset of relation R_1 where x_1 takes only heavy values.

In the first round, the algorithm computes the simple join $R_1'(x_1, x_2), R_2(x_2, x_3)$. By construction, the degree of any value of x_2 in R_1' is at most $p^{1/3}$. Thus, assuming that $p^{1/3} \leqslant M/p$ (equivalently $M \geqslant p^{4/3}$), we can apply Lemma 6.2.1 to obtain that we can compute the join with load $\tilde{O}(M/p)$. The resulting relation $R_{12}(x_1, x_2, x_3) = R_1'(x_1, x_2), R_2(x_2, x_3)$ has size at most $p^{1/3}M$.

In the second round, we compute the join $R_{12}(x_1, x_2, x_3), R_3(x_1, x_3)$, where $|R_{12}| \leqslant p^{1/3}M$ and $|R_3| = M$. Notice that the join is on the pair of variables (x_1, x_3), which has a degree

of 1 in relation R_3 (since we use set semantics). Hence, we can apply again Lemma 6.2.1 to obtain that we can compute the join in one round with load $\tilde{O}(p^{1/3}M/p) = \tilde{O}(M/p^{2/3})$. □

The reader may notice that the 2-round algorithm achieves a better load than the 1-round algorithm in the worst-case scenario. Indeed, in the previous section we proved that there exist instances for which we can not achieve load better than $O(M/p^{1/2})$ in a single round. By using an additional round, we can beat this bound and achieve a lower load. This confirms our intuition that with more rounds we can reduce the maximum load, even in the case of data with skew.

It is an intriguing question to explore whether this approach can be extended to other conjunctive queries. What is the minimum number of rounds that are needed for any query to achieve a worst-case optimal load?

Chapter 7

MPC AND THE EXTERNAL MEMORY MODEL

In this chapter, we explore the connection of the MPC model with the *external memory model*, which is a computational model first introduced in [17] by Aggarwal and Vitter.

7.1 The External Memory Model

In the external memory model, we model computation in the setting where the input data does not fit into main memory, and the dominant cost is reading the data from the disk into the memory and writing data on the disk.

Formally, we have an external memory (disk) of unbounded size, and an internal memory (main memory) that consists of M *words*. The processor can only use data stored in the internal memory to perform computation, and data can be moved between the two memories in blocks of consecutive B words. The *I/O complexity* of an algorithm is the number of input/output blocks that are moved during the algorithm, both from the internal memory to the external one, and vice versa.

We will use the result from [78] that the I/O complexity for sorting n elements in the external memory model is $sort(n) = O(\frac{n \log(n/B)}{B \log M} + \frac{n}{B})$.

The external memory model has been recently used in the context of databases to analyze algorithms for large datasets that do not fit in the main memory, with the main application being *triangle listing* [34, 49, 67, 48]. In this setting, the input is an undirected graph, and the goal is to list all triangles in the graph. In [67] and [48], the authors consider the related problem of *triangle enumeration*, where instead of listing triangles (and hence writing them to the external memory), for each triangle in the output we call an *emit*() function. The best result comes from [48], where the authors design a deterministic algorithm that enumerates

triangles in $O(|E|^{3/2}/(\sqrt{M}B))$ I/Os, where E is the number of edges in the graph. The authors in [48] actually consider a more general class of join problems, the so-called *Loomis-Whitney enumeration*. We should also mention [72], where the author presents external memory algorithms for enumerating subgraph patterns in graphs other than triangles.

The problem we consider in the context of external memory algorithms is a generalization of triangle enumeration. Given a full conjunctive query q, we want to *enumerate* all possible tuples in the output, by calling the $emit()$ function for each tuple in the output of query q. We assume that each tuple in the input can be represented by a single word.

7.2 From MPC to External Memory Algorithms

In this section, we will show how a parallel algorithm in the MPC model can help us construct an external memory algorithm. The idea behind the construction is that the distribution of the data to the servers can be used to decide which input data will be loaded into the memory; hence, the load L will correspond to the size of the internal memory M. Similarities between hash-join algorithms used for parallel processing and the variants of hash-join used for out-of-core processing have been already known, where the common theme is to create partitions and then process them one at a time. Here we generalize this idea to the processing of any conjunctive query in a rigorous way.

Let $\mathcal{A}$ be an MPC algorithm that computes query q over input I using r rounds with load $L(I, p)$. We assume that the communication for algorithm $\mathcal{A}$ takes a particular form: each tuple t during round k is sent to a set of servers $\mathcal{D}(t, k)$, where $\mathcal{D}$ depends only on the data statistics that are available to the algorithm from the start. Such statistical information can be the size of the relations, or even information about the heavy hitters in the data.[1] Observe that this is the tuple-based MPC model, with the additional assumption that we restrict the communication in the first round to be tuple-based as well. All of the algorithms that we have presented in this dissertation satisfy the above assumption. We will show how

[1]Even if this information is not available initially to the algorithm, we can easily obtain it by performing a single pass over the input data, which will cost $O(|I|/B)$.

to construct an external memory algorithm $\mathcal{B}$ based on the algorithm $\mathcal{A}$.

Simulation. The external memory algorithm $\mathcal{B}$ simulates the computation of algorithm $\mathcal{A}$ during each of the r rounds: round k, for $k = 1, \ldots, r$ simulates the total computation of the p servers during round k of $\mathcal{A}$. We pick a parameter p for the number of servers that we show how to compute later. The algorithm will use the tuple structure (t, s) to denote that tuple t resides in server $s = 1, \ldots, p$.

To initialize the algorithm $\mathcal{B}$, we will first assign the input data to the p servers (we can do this in any arbitrary way, as long as the data is equally distributed). More precisely, we read each tuple t of the input relations and then produce a tuple (t, s), where $s = 1, \ldots, p$ in a round-robin fashion, such that in the end each server is assigned $|I|/B$ data items. To achieve this, we load each relation in chunks of size B in the memory.

After the initialization, the algorithm $\mathcal{B}$, for each round $k = 1, \ldots, r$, performs the following steps:

1. All tuples, that will be of the form (t, s) are sorted according to the second attribute s, which is the destination server.

2. We load the tuples (t, s) in memory in chunks of size M, in the order by which they were sorted in the external memory. If we choose p such that $r \cdot L(I, p) \leqslant M$, we can fit in the internal memory all the tuples of any server s at round k. Hence, we first read into the internal memory the tuples for server 1, then server 2, and so on. For each server s, we perform the computation in the internal memory replicating the execution of algorithm $\mathcal{A}$ in server s at round k.

3. For each tuple t in server s (including the ones that are newly produced), we compute the tuples $\{(t, s') \mid s' \in \mathcal{D}(t, k)\}$, and we write them into the external memory in blocks of size B.

In other words, the writing into the internal memory and to the external memory simulates the communication step, where data is exchanged between servers. The algorithm $\mathcal{B}$ produces the correct result, since by choosing p in the right way we have guaranteed that we can load enough data in the main memory to simulate the local computation of $\mathcal{A}$ at each server. Observe that we do not need to write the final result to the external memory, since at the end of the last round we can just call $emit()$ for each tuple in the output.

Let us now look at the choice for p; recall that we must make sure that $r \cdot L(I,p) \leqslant M$. Hence, we must choose p_o such that

$$p_o = \min_p \{L(I,p) \leqslant M/r\}.$$

We next analyze the I/O cost of algorithm $\mathcal{B}$ for this choice of p_o.

Analysis. The initialization cost for the algorithm is $|I|/B$. Let us now analyze the cost for a given round $k = 1, \ldots, r$.

To analyze the cost, we will measure first the size of the data that will be sorted and then loaded into memory at round k. For this, observe that at every round of algorithm $\mathcal{B}$, the total amount of data that is communicated is at most $p_o \cdot L(I,p_o)$. Hence, the total amount of data that will be loaded into memory will be at most $k \cdot p_o \cdot L(I,p_o) \leqslant p_o M$, from our definition of p_o.

Thus, the first step of sorting the data has a cost of $sort(p_o M)$ in I/Os. The second step of loading the tuples into memory has a cost of $p_o M/B$, since we are loading the data using chunks of size B; we can do this since the data has been sorted according to the destination server. As for the third step of writing the data into the external memory, observe that the total number of tuples written will be equal to the number of tuples communicated to the servers at round $k+1$, which will be at most $p_o L(I,p_o) \leqslant p_o M/r$. Hence, the cost will be $p_o M/(rB)$ I/Os.

Summing the I/O cost of all three steps and over the r rounds, we obtain that the I/O

cost of the constructed algorithm $\mathcal{B}$ will be at most:

$$O\left(\frac{|I|}{B}+\sum_{k=1}^{r}\left(\frac{2p_oM}{B}+\frac{p_oM}{rB}+\frac{p_oM\log(p_oM/B)}{B\log(M)}\right)\right)$$
$$=O\left(\frac{|I|}{B}+\frac{(2r+1)p_oM}{B}+\frac{rp_oM\log(p_oM/B)}{B\log(M)}\right)$$

We have thus shown the following theorem.

Theorem 7.2.1. *Let $\mathcal{A}$ be a tuple-based MPC algorithm that computes query q over input I using r rounds with load $L(I,p)$. Then, there exists an external memory algorithm $\mathcal{B}$ that computes q over the same input I with I/O cost:*

$$O\left(\frac{|I|}{B}+\frac{(2r+1)p_oM}{B}+\frac{rp_oM\log(p_oM/B)}{B\log(M)}\right)$$

where $p_o=\min_p\{L(I,p)\leqslant M/r\}$.

To make sense of the above formula, we can simplify the above I/O cost in the context of computing conjunctive queries. In all of our algorithms we used a constant number of rounds r, and the load is typically $L(I,p)\geqslant|I|/p$. Hence, we can rewrite the I/O cost as $\tilde{O}\left(p_oM/B\right)$, where the $\tilde{O}$ notation hides some polylogarithmic factors.

We next present two applications of Theorem 7.2.1 to query processing in the external memory model.

Example 7.2.2. *Recall that we have showed in Chaoter 4 that we can compute any CQ q on input without skew in a single round with load $L=\tilde{O}((\prod_j m_j^{u_j}/p)^{1/\sum_j u_j})$ for any fractional edge packing $\mathbf{u}$. By choosing*

$$p_o=\tilde{O}\left(\prod_{j=1}^{\ell}\left(\frac{m_j}{M}\right)^{u_j}\right)$$

the application of Theorem 7.2.1 gives an algorithm in the external memory model where the number of I/Os is:

$$\tilde{O}\left(\frac{M}{B}\cdot\prod_{j=1}^{\ell}\left(\frac{m_j}{M}\right)^{u_j}\right)$$

For example, if we apply this to the cycle query C_k when all relation sizes are equal to m, we obtain an algorithm with I/O cost $\tilde{O}(\frac{m^{k/2}}{BM^{1/2}})$.

Example 7.2.3. *In Section 6.2, we presented a 2-round algorithm that computes triangles for any input data with load $L = O(m/p^{3/2})$, in the case where all relations have size m. By applying Theorem 7.2.1, we obtain an external memory algorithm that computes triangles with $\tilde{O}(m^{3/2}/(BM^{1/2}))$ I/O cost. Notice that this cost matches the I/O cost for triangle computation from [67] up to polylogarithmic factors.*

We will end this section with some observations and discussion. First, notice that the above two examples obtain an algorithm that has an I/O cost with polylogarithmic factors, which are not present when we design directly external memory algorithms, as in [67, 48]. One reason for this discrepancy is that in our design of parallel algorithms, we apply randomized hashing to distribute the load, hence the maximum load can exceed the expected load by some logarithmic factor. On the other hand, in the external memory model we can monitor the load because of the centralized nature of computation and hence we can make sure that the load always fits in memory. An interesting direction for future research is whether there is a generic mechanism for load distribution that translates better between the two computational models.

A different question is whether we can obtain optimal algorithms for the external memory model by simulating MPC algorithms. A 1-round algorithm cannot give an optimal external memory algorithm, as in the case of triangle computation the 1-round algorithm with worst-case input has optimal load $\tilde{O}(m/\sqrt{p})$, which implies a $\tilde{O}(m^2/(BM))$ external memory algorithm. However, we showed in the example that a 2-round algorithm implies a better algorithm with load $\tilde{O}(m^{3/2}/(BM^{1/2}))$, which is essentially optimal over worst-case inputs. But can we obtain an optimal external memory algorithm using the simulation of a multi-round MPC algorithm for $r \geqslant 2$?

This question is also tied to whether there exists a reverse simulation: given an external memory algorithm $\mathcal{B}$, can we construct an MPC algorithm $\mathcal{A}$ that simulates the execution

of $\mathcal{B}$? Given the centralized and sequential nature of the external memory model, it is hard to imagine that such a reduction exists. However, the computational task of answering conjunctive queries is highly parallel (since the complexity is in AC_0), which means it is possible that the optimal algorithm for the external memory model can have highly parallel structure that mimics the execution in the MPC model. We leave this question as an exciting direction for future research.

Chapter 8

CONCLUSION AND FUTURE OUTLOOK

In this dissertation, we propose a new approach in order to model query processing in modern massively parallel systems, and introduce the MPC model. This model captures the basic parameters of parallel query processing algorithms: the number of synchronization steps, and the communication complexity, which we measure as the maximum amount of data, or load, that can be received by any processor. Using the framework of the MPC model, we explore the tradeoff between the number of rounds and maximum load for the computation of multiway joins. Our results include novel algorithms and techniques to handle data skew, as well as lower bounds that match our upper bound for several classes of queries and input distributions. We have thus painted a large part of the landscape for join processing in massively parallel systems, even though several questions still remain open, especially regarding computing queries on input data with skew and worst-case analysis of the load.

In the next few sections, we discuss some of the open questions in this work, and describe some exciting directions for research that this dissertation has opened.

8.1 From Theory to Practice

Several of the algorithmic ideas presented in this work are already implemented as part of the Myria data management system [4]. In [33], the authors show how to implement the HyperCube algorithm in a modern parallel system, and demonstrate that it outperforms in several cases traditional query plans that decompose the execution into pipelined hash-joins, or use semi-join methods. Transitioning from the theory to a practical algorithm presents several challenges, such as rounding the shares to integer numbers, and handling

the communication efficiently.

An interesting research direction is whether the idea of computing multiple joins in a single step can be incorporated in a parallel query optimizer, in which case the HyperCube algorithm will function as a new operator in the query plan. Additionally, one can ask whether the vanilla HyperCube algorithm can handle the skew in a practical setting with real-world data, or it is necessary to employ our skew-handling techniques to reduce the maximum load and avoid stragglers.

8.2 Beyond Joins

The focus of this work was the study of the class of full conjunctive queries in the MPC model. However, such queries cover only a relatively small (although important) class of data processing tasks. Investigating the computation of larger classes of queries, such as queries with projections, unions, aggregation and negation is an interesting direction to explore in the future. For example, when computing queries with projections (such as boolean queries), one can use known techniques such as tree-decompositions of queries, which translate to early projection of variables: this can potentially reduce communication and achieve a better load than just the naive algorithm of computing the answer to the full query and then performing a final step that aggregates all the tuples and projects out the other variables. As another example, consider the query $q = R(x, y), \neg S(y)$, and notice that one can use the vanilla HyperCube algorithm and obtain a correct algorithm for q; but now is this algorithm optimal or can we use some other technique? Several other interesting classes of queries, where some work has already been done, include:

- *Theta-joins*: these are a non-natural joins of the form $R(x, z), S(x, z')$, where an arbitrary condition $\theta(z, z')$ is imposed, such as $z \neq z'$ or $z \leqslant z'$ (see [65] for parallel algorithms for theta joins in the context of MapReduce). A common application in this context is the *band-join*, where the condition is of the form $|z - z'| \leqslant c$ for some constant c.

- *Skyline queries*: given a multidimensional set, we want to find the points for which no other point exists that is at least as good along every dimension [29]. In [12], we study skyline computation in a model very similar to the MPC model (where each round has an additional phase where a small amount of data can be communicated). We show that one can compute the skyline of M points with load $O(M/p)$ using two rounds, but we leave open whether the skyline can be computed in one round for any dimension bigger than 3.
- *Datalog queries*: this class of queries adds iteration/recursion to relational algebra, which as we have seen is often a desired property for modern data management systems. The parallel evaluation of Datalog has been studied in the context of the PRAM model and circuit complexity [75, 53] and other models [43]. More recently, [70, 16] shed some light in the computation of recursive queries in the MapReduce framework. The fundamental question is again to identify the tradeoff between the load and the number of rounds in the MPC model. Recall here that we have showed in Chapter 6 that we need $\Omega(\log p)$ rounds to compute the connected components with a non-trivial load; hence, we have a non-constant number of rounds, which implies that we need different techniques to analyze the behavior of such recursive tasks.

8.3 Beyond the MPC model

The MPC model captures the computation for systems that have a *synchronous model*, where the computation is split into well-defined rounds followed by a synchronization barrier. However, not all systems choose this architecture; systems like GraphLab [58], or Myria [4] can run asynchronously. In an *asynchronous system*, the lack of coordination means that the data is exchanged without any bottleneck, but on the other hand makes proof of correctness and analysis of complexity more challenging tasks.

There have been recent efforts [20, 21] to describe the class of queries that can be expressed in such a distributed asynchronous system; however, this line of work focuses on the correctness of the algorithms, while ignoring the cost of communication. A fascinating

research direction is to look at how how we can design and analyze asynchronous algorithms that are not only provably correct, but also provably efficient in terms of the communication complexity.

BIBLIOGRAPHY

[1] Apache Hama. `https://hama.apache.org`.

[2] Greenplum. `http://pivotal.io/big-data/pivotal-greenplum-database`.

[3] Hadoop. `http://hadoop.apache.org`.

[4] Myria. `http://myria.cs.washington.edu/`.

[5] Netezza. `http://www-01.ibm.com/software/data/netezza/`.

[6] Teradata. `http://www.teradata.com`.

[7] Vertica. `http://www.vertica.com`.

[8] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[9] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[10] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. Enumerating subgraph instances using map-reduce. In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *ICDE*, pages 62–73. IEEE Computer Society, 2013.

[11] Foto N. Afrati, Manas Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround join algorithm in mapreduce. *CoRR*, abs/1410.4156, 2014.

[12] Foto N. Afrati, Paraschos Koutris, Dan Suciu, and Jeffrey D. Ullman. Parallel skyline queries. In Alin Deutsch, editor, *ICDT*, pages 274–284. ACM, 2012.

[13] Foto N. Afrati, Anish Das Sarma, Anand Rajaraman, Pokey Rule, Semih Salihoglu, and Jeffrey D. Ullman. Anchor-points algorithms for hamming and edit distances using mapreduce. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 4–14. OpenProceedings.org, 2014.

[14] Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4):277–288, 2013.

[15] Foto N. Afrati and Jeffrey D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.

[16] Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari, editors, *EDBT*, pages 132–143. ACM, 2012.

[17] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[18] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[19] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Nicola Onose, Pouria Pirzadeh, Rares Vernica, and Jian Wen. Asterix: An open source system for "big data" management and analysis. *PVLDB*, 5(12):1898–1901, 2012.

[20] Tom J. Ameloot, Bas Ketsman, Frank Neven, and Daniel Zinn. Weaker forms of monotonicity for declarative networking: a more fine-grained answer to the calm-conjecture. In Hull and Grohe [50], pages 64–75.

[21] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15, 2013.

[22] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748. IEEE Computer Society, 2008.

[23] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In Richard Hull and Wenfei Fan, editors, *PODS*, pages 273–284. ACM, 2013.

[24] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In Hull and Grohe [50], pages 212–223.

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

[26] Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini, editors. *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*. ACM, 2012.

[27] George Bennett. Probability inequalities for the sum of independent random variables. *Journal of the American Statistical Association*, 57(297):pp. 33–45, 1962.

[28] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE*, pages 1151–1162. IEEE Computer Society, 2011.

[29] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In Dimitrios Georgakopoulos and Alexander Buchmann, editors, *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 421–430. IEEE Computer Society, 2001.

[30] Joshua Brody, Amit Chakrabarti, Ranganath Kondapally, David P. Woodruff, and Grigory Yaroslavtsev. Beyond set disjointness: the communication complexity of finding the intersection. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 106–113. ACM, 2014.

[31] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[32] Surajit Chaudhuri. What next?: a half-dozen data management research goals for big data and the cloud. In Benedikt et al. [26], pages 1–4.

[33] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In Timos Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 63–78. ACM, 2015.

[34] Shumo Chu and James Cheng. Triangle listing in massive networks. *TKDD*, 6(4):17, 2012.

[35] Fan R. K. Chung, Zoltán Füredi, M. R. Garey, and Ronald L. Graham. On the fractional covering number of hypergraphs. *SIAM J. Discrete Math.*, 1(1):45–49, 1988.

[36] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *PPOPP*, pages 1–12, 1993.

[37] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[38] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB*, pages 228–237. Morgan Kaufmann, 1986.

[39] Stephan Ewen, Sebastian Schelter, Kostas Tzoumas, Daniel Warneke, and Volker Markl. Iterative parallel data processing with stratosphere: an inside look. In Ross et al. [69], pages 1053–1056.

[40] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On distributing symmetric streaming computations. In Shang-Hua Teng, editor, *SODA*, pages 710–719. SIAM, 2008.

[41] Ehud Friedgut. Hypergraphs, entropy, and inequalities. *The American Mathematical Monthly*, 111(9):749–760, 2004.

[42] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

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

[44] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[45] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In Takao Asano, Shin-Ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *ISAAC*, volume 7074 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2011.

[46] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 289–298. ACM Press, 2006.

[47] Dirk Van Gucht, Ryan Williams, David P. Woodruff, and Qin Zhang. The communication complexity of distributed set-joins with applications to matrix multiplication. In Milo and Calvanese [61], pages 199–212.

[48] Xiaocheng Hu, Miao Qiao, and Yufei Tao. Join dependency testing, loomis-whitney join, and triangle enumeration. In Milo and Calvanese [61], pages 291–301.

[49] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. Massive graph triangulation. In Ross et al. [69], pages 325–336.

[50] Richard Hull and Martin Grohe, editors. *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*. ACM, 2014.

[51] Neil Immerman. Expressibility and parallel complexity. *SIAM J. Comput.*, 18(3):625–638, 1989.

[52] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *EuroSys*, pages 59–72. ACM, 2007.

[53] Paris C. Kanellakis. Logic programming and parallel complexity. In *Foundations of Deductive Databases and Logic Programming.*, pages 547–585. Morgan Kaufmann, 1988.

[54] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In Moses Charikar, editor, *SODA*, pages 938–948. SIAM, 2010.

[55] Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In Maurizio Lenzerini and Thomas Schwentick, editors, *PODS*, pages 223–234. ACM, 2011.

[56] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, Cambridge, England ; New York, 1997.

[57] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome A. Rolia. Skewtune: mitigating skew in mapreduce applications. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *SIGMOD Conference*, pages 25–36. ACM, 2012.

[58] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[59] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[60] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[61] Tova Milo and Diego Calvanese, editors. *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 2015.

[62] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Technical report, University of Washington, 2014.

[63] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In Benedikt et al. [26], pages 37–48.

[64] Hung Q. Ngo, Christopher Re, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *CoRR*, abs/1310.3314, 2013.

[65] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD Conference*, pages 949–960. ACM, 2011.

[66] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In Jason Tsong-Li Wang, editor, *SIGMOD Conference*, pages 1099–1110. ACM, 2008.

[67] Rasmus Pagh and Francesco Silvestri. The input/output complexity of triangle enumeration. In Hull and Grohe [50], pages 224–233.

[68] Jeff M. Phillips, Elad Verbin, and Qin Zhang. Lower bounds for number-in-hand multiparty communication complexity, made easy. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 486–501. SIAM, 2012.

[69] 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.

[70] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In Pablo Barceló and Reinhard Pichler, editors, *Datalog*, volume 7494 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2012.

[71] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *MSST*, pages 1–10. IEEE Computer Society, 2010.

[72] Francesco Silvestri. Subgraph enumeration in massive graphs. *CoRR*, abs/1402.3444, 2014.

[73] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *WWW*, pages 607–614. ACM, 2011.

[74] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[75] Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. In *FOCS*, pages 438–454. IEEE Computer Society, 1986.

[76] Michael Stonebraker University and Michael Stonebraker. The case for shared nothing. *Database Engineering*, 9:4–9, 1986.

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

[78] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[79] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings.*, pages 537–548. Morgan Kaufmann, 1991.

[80] David P. Woodruff and Qin Zhang. When distributed computation is communication expensive. In Yehuda Afek, editor, *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2013.

[81] 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. [69], pages 13–24.

[82] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1043–1052, New York, NY, USA, 2008. ACM.

[83] Andrew Chi-Chih Yao. Lower bounds by probabilistic arguments (extended abstract). In *FOCS*, pages 420–428. IEEE Computer Society, 1983.

[84] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[85] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

Appendix A

ADDITIONAL MATERIAL

A.1 Probability Bounds

In this section, we show how to obtain lower bounds on the probability of failure using bounds on the expected output. We start by proving a lemma regarding the distribution of the query output for random matching databases.

Lemma A.1.1. *Let I be a random matching database for a connected conjunctive query q, and let $\mu = \mathbf{E}[|q(I)|]$. Then, for any $\alpha \in [0, 1)$ we have:*

$$P(|q(I)| > \alpha\mu) \geqslant (1-\alpha)^2 \frac{\mu}{\mu+1}$$

Proof. To prove the bound, we will use the Paley-Zygmund inequality for the random variable $|q(I)|$:

$$P(|q(I)| > \alpha\mu) \geqslant (1-\alpha)^2 \frac{\mu^2}{\mathbf{E}[|q(I)|^2]}$$

To bound the quantity $\mathbf{E}[|q(I)|^2]$, we construct a query q' that consists of q plus a copy of q with new variables. For example, if $q = R(x,y), S(y,z)$, we define $q' = R(x,y), S(y,z), R(x',y'), S(y',z')$. We now have:

$$\begin{aligned}
\mathbf{E}[|q(I)|^2] = \mathbf{E}[|q'(I)|] &= \sum_{\mathbf{a},\mathbf{a}' \in [n]^k} \prod_{j=1}^{\ell} P(\mathbf{a}_j \in S_j \wedge \mathbf{a}'_j \in S_j) \\
&= \sum_{\mathbf{a} \neq \mathbf{a}' \in [n]^k} \prod_{j=1}^{\ell} P(\mathbf{a}_j \in S_j \wedge \mathbf{a}'_j \in S_j) + \sum_{\mathbf{a} \in [n]^k} \prod_{j=1}^{\ell} P(\mathbf{a}_j \in S_j) \\
&= \sum_{\mathbf{a} \neq \mathbf{a}' \in [n]^k} \prod_{j=1}^{\ell} P(\mathbf{a}_j \in S_j) P(\mathbf{a}'_j \in S_j \mid \mathbf{a}_j \in S_j) + \mu
\end{aligned}$$

Now, observe that when $\mathbf{a}, \mathbf{a}'$ differ in all positions, since the database is a matching, the event $\mathbf{a}'_j \in S_j$ is independent of the event $\mathbf{a}_j \in S_j$ for every relation S_j; in this case, $P(\mathbf{a}'_j \in S_j \mid \mathbf{a}_j \in S_j) = P(\mathbf{a}'_j \in S_j)$ for every S_j. On the other hand, if $\mathbf{a}, \mathbf{a}'$ agree in at least one position, then since q is connected it will be that $P(\mathbf{a}'_j \in S_j \mid \mathbf{a}_j \in S_j) = 0$ for some relation S_j. Thus, we can write:

$$\begin{aligned}
\mathbf{E}[|q(I)|^2] &\leqslant \sum_{\mathbf{a}\neq\mathbf{a}'\in[n]^k} \prod_{j=1}^{\ell} P(\mathbf{a}_j \in S_j) P(\mathbf{a}'_j \in S_j) + \mu \\
&= (n^{2k} - n^k) \prod_j (m_j/n^{a_j})^2 + \mu \\
&= \left(1 - n^{-k}\right) \mu^2 + \mu \\
&\leqslant \mu^2 + \mu
\end{aligned}$$

□

For a deterministic algorithm A that computes the answers to a query q over a randomized instance I, let $fail$ denote the event that $|q(I) \backslash A(I)| > 0$, *i.e.* the event that the algorithm A fails to return all the output tuples. The next lemma shows how we can use a bound on the expectation to obtain a bound on the probability of failure.

Lemma A.1.2. *Let I be a random matching database for a connected query q. Let A be a deterministic algorithm such that $\mathbf{E}[|A(I)|] \leqslant f\mathbf{E}[|q(I)|]$, where $f \leqslant 1$. Let $\mu = \mathbf{E}[|q(I)|]$ and let C_α denote the event that $|q(I)| > \alpha\mu$. Then,*

$$P(fail \mid C_{1/3}) \geqslant 1 - 9f$$

Proof. We start by writing

$$\begin{aligned}
P(fail \mid C_\alpha) &= P(|q(I) \backslash A(I)| > 0 \mid C_\alpha) \\
&\geqslant P(|A(I)| \leqslant \alpha\mu \mid C_\alpha)
\end{aligned}$$

$$= 1 - P(|A(I)| > \alpha\mu \mid C_\alpha)$$

Additionally, we have:

$$\begin{aligned}
\mathbf{E}[|A(I)] &= \mathbf{E}[|A(I)| \mid C_\alpha] \cdot P(C_\alpha) + \mathbf{E}[|A(I)| \mid \neg C_\alpha] \cdot P(\neg C_\alpha) \\
&\geqslant \mathbf{E}[|A(I)| \mid C_\alpha] \cdot P(C_\alpha) \\
&= P(C_\alpha) \sum_{t=\lfloor \alpha\mu \rfloor + 1}^{\infty} t \cdot P(|A(I)| = t \mid C_\alpha) \\
&\geqslant P(C_\alpha)(\lfloor \alpha\mu \rfloor + 1) P(|A(I)| > \alpha \mid C_\alpha)
\end{aligned}$$

Combining the above two inequalities, we can now write

$$P(fail \mid C_\alpha) \geqslant 1 - \frac{\mathbf{E}[|A(I)|]}{(\lfloor \alpha\mu \rfloor + 1) P(C_\alpha)} \geqslant 1 - \frac{f\mu}{(\lfloor \alpha\mu \rfloor + 1) P(C_\alpha)}$$

We can now apply Lemma A.1.1 to obtain $P(C_\alpha) = P(|q(I)| > \alpha\mu) \geqslant (1-\alpha)^2\mu/(\mu+1)$. Thus,

$$P(fail \mid C_\alpha) \geqslant 1 - \frac{f\mu}{\lfloor \alpha\mu \rfloor + 1} \cdot \frac{\mu + 1}{\mu(1-\alpha)^2} = 1 - \frac{f(\mu+1)}{(\lfloor \alpha\mu \rfloor + 1)(1-\alpha)^2}$$

We can now choose $\alpha = 1/3$ to obtain that

$$P(fail \mid C_{1/3}) \geqslant 1 - (9/4) f \frac{\mu + 1}{\lfloor \mu/3 \rfloor + 1}$$

The final step is to show that the quantity $\frac{\mu+1}{\lfloor \mu/3 \rfloor + 1}$ is upper bounded by 4 for any (positive) value of μ. We distinguish here two cases:

- If $\mu < 3$, then $\lfloor \mu/3 \rfloor = 0$. Thus, $\frac{\mu+1}{\lfloor \mu/3 \rfloor + 1} = \mu + 1 < 4$.
- If $\mu \geqslant 3$, we use the fact $\mu/3 \leqslant \lfloor \mu/3 \rfloor + 1$ to obtain that $\frac{\mu+1}{\lfloor \mu/3 \rfloor + 1} \leqslant (\mu+1)/(\mu/3) = 3(1 + 1/\mu) \leqslant 3(1 + 1/3) = 4$.

This concludes the proof of the lemma. □

A.2 Hashing

In this section, we present a detailed analysis of the behavior of the HyperCube algorithm for input distributions with various guarantees. Throughout this section, we assume that a hash function is chosen randomly from a *strongly universal family* of hash functions. Recall that a strongly universal set of hash function is a set $\mathcal{H}$ of functions with range $[p]$ such that, for any $n \geqslant 1$, any distinct values $a_1, \ldots, a_n$ and any bins $B_1, \ldots, B_n \in [p]$, we have that $P(h(a_1) = B_1 \wedge \cdots \wedge h(a_n) = B_n) = 1/p^n$, where the probability is over the random choices of $h \in \mathcal{H}$.

A.2.1 Basic Partition

We start by examining the following scenario. Suppose that we have a set of weighted balls which we hash-partition into p bins; what is the maximum load among all the bins? Assuming that the sum of the weights is m, it is easy to see that the expected load for each bin is m/p. However, this does not tell us anything about the maximum load. In particular, in the case where we have one ball of weight m, the maximum load will always be m, which is far from the expected load.

In order to obtain meaningful bounds on the distribution of the maximum load, we thus have to put a restriction on the maximum weight of a ball. The following theorem provides such a tail bound on the probability distribution.

Theorem A.2.1 (Weighted Balls in Bins)**.** *Let S be a set where each element i has weight w_i and $\sum_{i \in S} w_i \leqslant m$. Let $p > 0$ be an integer. Suppose that for some $\alpha > 0$, $\max_{i \in S}\{w_i\} \leqslant \frac{m}{\alpha p}$. We hash-partition S into p bins. Then for any $\delta > 0$*

$$P(\textit{maximum weight of any bin} \geqslant (1+\delta)\frac{m}{p}) \leqslant p \cdot e^{-\alpha h(\delta)} \tag{A.1}$$

where $h(x) = (1+x)\ln(1+x) - x$.

Theorem A.2.1 immediately follows from the following lemma (which implies the bound for a single bin), together with a union bound over all p bins.

Lemma A.2.2. *Let I be an index set. Let $p \geqslant 2$. Let $w \in \mathbb{R}^I$ satisfy $w \geqslant 0$, $||w||_1 \leqslant m_1$, and $||w||_\infty \leqslant m_\infty = m_1/(\alpha p)$. Let $(Y_i)_{i \in I}$ be a vector of i.i.d. random indicator variables with $P(Y_i = 1) = 1/p$. Then*

$$P(\sum_{i \in I} w_i Y_i > (1+\delta)\frac{m_1}{p}) \leqslant e^{-\alpha \cdot h(\delta)}$$

where $h(x) = (1+x)\ln(1+x) - x$.

We prove Lemma A.2.2 using Bennett's inequality [27].

Theorem A.2.3 (Bennett's Inequality)**.** *Let X_i be random variables such that $\mathbf{E}[X_i] = 0$ and $|X_i| \leqslant d$ for all $i \in [n]$. Then, for all $t \geqslant 0$,*

$$P(\sum_{i \in [n]} X_i > t) \leqslant \exp\left(-\frac{\sum_i Var(X_i)}{d^2} \cdot h\left(\frac{td}{\sum_i Var(X_i)}\right)\right)$$

where $h(x) = (1+x)\ln(1+x) - x$.

of Lemma A.2.2. Let $Z_i = Y_i - 1/p$ and $X_i = w_i Z_i$. Then $\mathbf{E}[X_i] = w_i \mathbf{E}[Z_i] = 0$ and $|X_i| = w_i|Z_i| \leqslant w_i(1-1/p) \leqslant ||w||_\infty(1-1/p) = (1-1/p)\frac{m_1}{\alpha p}$. Also, $\mathrm{Var}(X_i) = \mathbf{E}[X_i^2] = w_i^2 \mathbf{E}[Z_i^2] = w_i^2(\mathbf{E}[Y_i^2] - 2\mathbf{E}[Y_i]/p + 1/p^2) = w_i^2(1/p - 1/p^2)$.

Observe that the probability of the event $\sum_{i \in I} w_i Y_i > (1+\delta)m_1/p$ is precisely the probability that $\sum_{i \in I} X_i > \delta m_1/p$. By setting $d = (1-1/p)\frac{m_1}{\alpha p}$ all the variables X_i satisfy the condition $|X_i| \leqslant d$ in Bennett's theorem, and therefore:

$$P(\sum_{i \in I} X_i > t) \leqslant \exp\left(-\frac{t}{d} \cdot \frac{h(x)}{x}\right) \qquad \text{where } x = \frac{td}{\sum_{i \in I} \mathrm{Var}(X_i)}$$

We set $t = \delta m_1/p$. Then we have $x \geqslant \delta$ because:

$$\sum_{i \in I} \mathrm{Var}(X_i) \leqslant \sum_{i \in I} w_i^2(1/p - 1/p^2) \leqslant \frac{d}{p}\sum_{i \in I} w_i \leqslant \frac{dm_1}{p}$$

Observe that the derivative of $h(x)$ is $h'(x) = \ln(1+x)$. The function $h(x)/x$ is increasing, because $(h(x)/x)' = (xh'(x) - h(x))/x^2 = (x\ln(1+x) - (1+x)\ln(1+x) + x)/x^2 = (x - \ln(1+x))/x^2 \geqslant 0$, and therefore $h(x)/x \geqslant h(\delta)/\delta$. This implies that

$$P(\sum_i X_i > t) \leqslant \exp\left(-\frac{t}{d} \cdot \frac{h(\delta)}{\delta}\right) = \exp\left(-\frac{m_1}{pd} \cdot h(\delta)\right) \leqslant e^{-\alpha h(\delta)}$$

as required. □

We also will find the following extension of Lemma A.2.2 to be useful.

Theorem A.2.4. *Let I be an index set and $p \geqslant 2$. Let $(w^{(j)})_j$ be a sequence of vectors in $\mathbb{R}^I$ satisfying $w^{(j)} \geqslant 0$, $||w^{(j)}||_1 \leqslant m_1$, and $||w^{(j)}||_\infty \leqslant m_\infty = m_1/(\alpha p)$. Suppose further that $||\sum_j w^{(j)}||_1 \leqslant km_1$. Let $(Y_i)_{i\in I}$ be a vector of i.i.d. random indicator variables with $\mathbb{P}(Y_i = 1) = 1/p$. Then*

$$P(\exists j \ \sum_{i\in I} w_i^{(j)} Y_i > (1+\delta)\frac{m_1}{p}) \leqslant 2k \cdot e^{-\alpha \cdot h(\delta)}$$

where $h(x) = (1+x)\ln(1+x) - x$.

The proof of this theorem follows easily from the following lemma.

Lemma A.2.5. *Let $(w^{(j)})_j$ be a sequence of vectors in $\mathbb{R}^I$ satisfying $w^{(j)} \geqslant 0$, $||w^{(j)}||_1 \leqslant m_1$, and $||w^{(j)}||_\infty \leqslant m_\infty$. Suppose further that $||\sum_j w^{(j)}||_1 \leqslant km_1$. Then, there exists a sequence of at most $2k$ vectors $u^{(1)}, \ldots, u^{(2k)} \in \mathbb{R}^I$ such that each $u^{(\ell)} \geqslant 0$, $||u^{(\ell)}||_1 \leqslant m_1$, and $||u^{(\ell)}||_\infty \leqslant m_\infty$, and for every j, there is some $\ell \in [2k]$ such that $w^{(j)} \leqslant u^{(\ell)}$, where the inequality holds only if it holds for every coordinate.*

Proof. The construction goes via the first-fit decreasing algorithm for bin-packing. Sort the vectors $w^{(j)}$ in decreasing order of $||w^{(j)}||_1$. Then greedily group them in bins of capacity m_1. That is, we begin with $w^{(1)}$ and continue to add vectors until we find the largest j_1 such that $\sum_{j=1}^{j_1} ||w^{(j)}||_1 \leqslant m_1$. Define $u^{(1)}$ by $u_i^{(1)} = \max_{1\leqslant j\leqslant j_1} w_i^{(j)}$ for each $i \in I$. Now $||u^{(1)}|| \leqslant m_1$

and $||u^{(1)}||_\infty \leqslant \max_j ||w^{(j)}||_\infty \leqslant m_\infty$. Moreover, for each $j \in [1, j_1]$, $w^{(j)} \leqslant u^{(1)}$ by definition. Then repeat beginning with $w^{(j_1+1)}$ until the largest j_2 such that $\sum_{j=j_1+1}^{j_2} ||w^{(j)}||_1 \leqslant m_1$, and define $u^{(2)}$ by $u_i^{(2)} = \max_{j_1+1 \leqslant j \leqslant j_2} w_i^{(j)}$ for each $i \in I$ as before, and so on. Since the contribution of each subsequent $||w^{(j)}||_1$ is at most that of its predecessor, if it cannot be included in a bin, then that bin is more than half full so we have $||u^{(\ell)}||_1 > m_1/2$ for all ℓ. Since $\sum_\ell ||u^{(\ell)}||_1 \leqslant \sum_j ||w^{(j)}||_1 \leqslant km_1$, there must be at most $2k$ such $u^{(\ell)}$. □

of Theorem A.2.4. We apply Lemma A.2.5 to the vectors $w^{(j)}$ to construct $u^{(1)}, \ldots, u^{(2k)}$. We then apply a union bound to the application of Lemma A.2.2 to each of the vectors $u^{(\ell)}$. The total probability that there exists some $\ell \in [2k]$ such that $\sum_{i \in I} u_i^{(\ell)} Y_i > (1+\delta)m_1/(\alpha p)$ is at most $2k \cdot e^{-\alpha h(\delta)}$. Now for each j, there is some ℓ such that $w^{(j)} \leqslant u^{(\ell)}$ and hence $\sum_{i \in I} w_i^{(j)} Y_i \leqslant \sum_{i \in I} u_i^{(\ell)} Y_i$. Therefore if there exists a j such that $\sum_{i \in I} w_i^{(j)} Y_i > (1+\delta)m_1/p$ then there exists an ℓ such that $\sum_{i \in I} u_i^{(\ell)} Y_i > (1+\delta)m_1/p$. □

A.2.2 HyperCube Partition

Before we analyze the load of the HC algorithm, we present some useful notation. Even though the analysis in the main paper assumes that 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 of $[r]$-tuples. Define:

$$
\begin{aligned}
m(S) &= |S| && \text{the size of the bag } S\text{, counting duplicates} \\
\Pi_U(S) &= \{\pi_U(J) \mid J \in S\} && \text{duplicates are kept, thus } |\Pi_U(S)| = |S| \\
\sigma_J(S) &= \{K \in S \mid \pi_U(K) = J\} && \text{bag of tuples that contain } J \\
d_J(S) &= |\sigma_J(S)| && \text{the degree of the tuple } J
\end{aligned}
$$

Given shares $p_1, \ldots, p_r$, such that $\prod_u p_u = p$, let $p_U = \prod_{u \in U} p_u$ for any attribute set U. Let $h_1, \ldots, h_r$ be independently chosen hash functions, with ranges $[p_1], \ldots, [p_r]$, respectively. The *hypercube hash-partition of* S sends each element $(i_1, \ldots, i_r)$ to the bin $(h_1(i_1), \ldots, h_r(i_r))$.

HyperCube Partition without Promise

We prove the following:

Theorem A.2.6. *Let S be a bag of tuples of $[n]^r$ such that each tuple in S occurs at most $m/(\alpha^r p)$ times, for some constant $\alpha > 0$. Then for any $\delta > 0$:*

$$P\left(\textit{maximum size any bin} > (1+\delta)\frac{m(S)}{\min_u p_u}\right) \leqslant r \cdot p \cdot e^{-\alpha \cdot h(\delta)}$$

where the bin refers to the HyperCube partition of S using shares $p_1, \ldots, p_r$.

Notice that there is no promise on how large the degrees can be. The only promise is on the number of repetitions in the bag S, which is automatically satisfied when S is a set, since it is at most one.

Proof. We prove the theorem by induction on r. If $r = 1$ then it follows immediately from Theorem A.2.1 by letting the weight of a ball i be the number of elements in S containing it. Assume now that $r > 1$. We partition the domain $[n]$ into two sets:

$$D_{small} = \{i \mid d_{r \mapsto i}(S) \leqslant m/(\alpha p_r)\} \text{ and } D_{large} = \{i \mid d_{r \mapsto i}(S) > m/(\alpha p_r)\}$$

Here $r \mapsto i$ denotes the tuple (i); in other words $\sigma_{r \mapsto i}(S)$ returns the tuples in S whose last (r-th) attribute has value i. We then partition the bag S into two sets S_{small}, S_{large}, where S_{small} consists of tuples t where $\pi_r(t) \in D_{small}$, and S_{large} consists of those where $\pi_r(t) \in D_{large}$. The intuition is that we can apply Theorem A.2.1 directly to show that S_{small} is distributed well by the hash function h_r. On the other hand, there cannot be many $i \in D_{large}$, in particular

$|D_{large}| \leqslant \alpha p_r$, and hence the projection of any tuple in S_{large} onto $[r-1]$ has at most D_{large} extensions in S_{large}. Thus, we can obtain a good inductive distribution of S_{large} onto $[r-1]$ using $h_1, \ldots, h_{r-1}$.

Formally, for $U \subset [r]$ and $T \subseteq S$, let $M_U(T)$ denote the maximum number of tuples of T that have any particular fixed value under $h_U = \times_{j \in U} h_j$. With this notation, $M_{[r]}(S)$ denotes the the maximum number of tuples from S in any bin. Hence, our goal is to show that $P(M_{[r]}(S) > (1+\delta)m(S)/\min_{u \in [r]} p_u) < r \cdot p \cdot e^{-\alpha h(\delta)}$. Now, by Theorem A.2.1,

$$P(M_{\{r\}}(S_{small}) > (1+\delta)m(S_{small})/p_r) \leqslant p \cdot e^{-\alpha h(\delta)}$$

and consequently

$$P(M_{[r]}(S_{small}) > (1+\delta)m(S_{small})/\min_{u \in [r]} p_u) \leqslant p \cdot e^{-\alpha h(\delta)}.$$

Let $S' = \Pi_{[r-1]}(S_{large})$. Since projections keep duplicates, we have $m(S') = m(S_{large})$ and $M_{[r-1]}(S') = M_{[r-1]}(S_{large})$. By the assumption in the theorem statement, each tuple in S, and hence in S_{large}, occurs at most $m/(\alpha^r p)$ times. Then, since $|D_{large}| \leqslant \alpha p_r$, each tuple in S' occurs at most $m/(\alpha^{r-1} p')$ times where $p' = \prod_{u \in [r-1]} p_u$. Therefore we can apply the inductive hypothesis to S' to yield

$$P(M_{[r-1]}(S') > (1+\delta)m(S')/\min_{u \in [r-1]} p_u) \leqslant (r-1) \cdot p \cdot e^{-\alpha h(\delta)}$$

and hence

$$P(M_{[r]}(S_{large}) > (1+\delta)m(S_{large})/\min_{u \in [r]} p_u) \leqslant (r-1) \cdot p \cdot e^{-\alpha h(\delta)}.$$

Since $m(S) = m(S_{small}) + m(S_{large})$ and $M_{[r]}(S) = M_{[r]}(S_{small}) + M_{[r]}(S_{large})$,

$$P(M_{[r]}(S) > (1+\delta)m(S)/\min_{u \in [r]} p_u)) \leqslant p \cdot e^{-\alpha h(\delta)} + (r-1) \cdot e^{-\alpha h(\delta)} = r \cdot p \cdot e^{-\alpha h(\delta)}$$

as required. □

HyperCube Partition with Promise

The following theorem extends Theorem A.2.6 to the case when we have a promise on the degrees in the bag (or set) S.

Theorem A.2.7. *Let S be a bag of tuples of $[n]^r$, and suppose that for every U-tuple J we have $d_J(S) \leqslant \frac{m}{\alpha^{|U|} p_U}$ where $\alpha > 0$. Consider a hypercube hash-partition of S into p bins. Then, for any $\delta \geqslant 0$:*

$$\mathbb{P}\left(\textit{maximum size of any bin} > (1+\delta)^r \frac{m(S)}{p}\right) \leqslant f(p,r,\alpha)\cdot e^{-\alpha\cdot h(\delta)}$$

where the bin refers to the HyperCube partition of S using shares $p_1,\ldots,p_r$ and

$$f(p,r,\alpha) = 2p \sum_{j=1}^{r} \prod_{u\in[j-1]} (\alpha + 1/p_u) \leqslant 2p\ \frac{(\alpha+\varepsilon)^r - 1}{\alpha+\varepsilon-1}, \tag{A.2}$$

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

We will think of r as a constant, p_u as being relatively large, and α as $\log^{O(1)} p$.

Proof. We prove the theorem by induction on r. The base case $r = 1$ follows immediately from Theorem A.2.1 since an empty product evaluates to 1 and hence $f(p,1,\alpha) = 2p$.

Suppose that $r > 1$. There is one bin for each r-tuple in $[p_1]\times\cdots\times[p_r]$. We analyze cases based on the value $b \in [p_r]$. Define

$$S^{r\to b} = \bigcup_{i\in[n]:h_r(i)=b} \sigma_{r\mapsto i}(S) \qquad \text{and } S'(b) = \Pi_{[r-1]}(S^{r\to b})$$

Here $r \mapsto i$ denotes the tuple (i). $S^{r\to b}$ is a random variable depending on the choice of the hash function h_r that represents the bag of tuples sent to bins whose first projection is b.

$S'(b)$ is essentially the same bag where we drop the last coordinate, which, strictly speaking, we need to do to apply induction. Then $m(S'(b)) = m(S^{r\to b})$.

We will 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 $S'(b)$ and $m' = (1+\delta)m(S)/p_r$ instead of S and m. More precisely, we need to argue that, *with high probability, for every* $U \subseteq [r-1]$ *and every* U*-tuple* J,

$$d_J(S'(b)) \leqslant \frac{m'}{\alpha^{|U|}p_U} = (1+\delta)\frac{m}{\alpha^{|U|}p_U p_r} \tag{A.3}$$

Fix a subset $U \subseteq [r-1]$. The case for $U = \varnothing$ is precisely the bound for the size $m(S'(b))$ of $S'(b)$. Since the promise of the theorem statement with $U = \{r\}$ implies that $d_{\{r\}}(S) \leqslant m/(\alpha p_r)$, by Theorem A.2.1 we have that $P(m(S'(b)) > m') \leqslant e^{-\alpha h(\delta)}$.

Assume next that $U \neq \varnothing$. Observe that $d_J(S'(b))$ is precisely the number of tuples of S consistent with (J,i) such that $h_r(i) = b$. Using Theorem A.2.4, we upper bound the probability that there is some U-tuple J such that (A.3) fails. Let $k(U) = \alpha^{|U|}p_U$. For each fixed (J,i), the promise for coordinates $U \cup \{r\}$ implies that there are at most $\frac{m}{\alpha^{|U|+1}p_U p_r} = \frac{m}{\alpha p_r k(U)}$ tuples in S consistent with (J,i). Further, the promise for coordinates U implies that there are at most $\frac{m}{\alpha^{|U|}p_U} = \frac{m}{k(U)}$ tuples in S consistent with J. For each such J define vector $w^{(J)}$ by letting $w_i^{(J)}$ be the number of tuples consistent with (J,i). Thus $||w^{(J)}||_\infty \leqslant \frac{m}{\alpha p_r k(U)}$ for all J and $||w^{(J)}||_1 \leqslant \frac{m}{k(U)}$ for all J. Finally note that since there are $m = m(S)$ tuples in S, $\sum_J ||w^{(J)}||_1 \leqslant m$. We therefore we can apply Theorem A.2.4 with $k = k(U)$, $m_1 = m/k(U)$ and $m_\infty = m_1/(\alpha p_r)$ to say that the probability that there is some U-tuple J such that $d_J(S'(b)) > (1+\delta)m_1/p_1 = (1+\delta)m/(p_r k(U))$ is at most $2k(U)\cdot e^{-\alpha h(\delta)}$.

For a fixed b, we now use a union bound over the possible sets $U \subseteq [r-1]$ to obtain a total probability that (A.3) fails for some set U and some U-tuple J of at most

$$2\sum_{U\subseteq[r-1]} \alpha^{|U|}p_U \cdot e^{-\alpha h(\delta)} = 2\prod_{u\in[r-1]}(1+\alpha p_u)\cdot e^{-\alpha h(\delta)}$$

$$= 2(p/p_r) \prod_{u \in [r-1]} (\alpha + 1/p_u) \cdot e^{-\alpha h(\delta)}$$

If $m(S'(b)) \leqslant m'$ and (A.3) holds for all $U \subseteq [r-1]$ and U-tuples J, then we apply the induction hypothesis (A.2) to derive that the probability that some bin that has b in its last coordinate has more than $(1+\delta)^{r-1} m'/(p/p_r) = (1+\delta)^r m/p$ tuples is at most $f(p/p_r, r-1, \alpha) \cdot e^{-\alpha h(\delta)}$.

Since there are p_r choices for b, we obtain a total failure probability at most $g \cdot e^{-\alpha h(\delta)}$ where

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

The final bound uses geometric series sum upper bound. □

VITA

Paraschos Koutris was born in Athens, Greece, on July 13, 1986. After graduating from Ionios High School in 2004, he studied Electrical and Computer Engineering at the National Technical University of Athens, from where he received a Diploma in 2009. He completed a Master of Science in Logic, Algorithms and Computation at the University of Athens in 2010, with a thesis titled "Infrastructure Leasing Problems". He entered the graduate program in Computer Science and Engineering at the University of Washington, Seattle, in 2010.