Worst-Case Optimal Algorithms for Parallel Query Processing*

Paraschos Koutris[1], Paul Beame[2], and Dan Suciu[3]

1 **University of Washington, Seattle, WA, USA**
pkoutris@cs.washington.edu
2 **University of Washington, Seattle, WA, USA**
beame@cs.washington.edu
3 **University of Washington, Seattle, WA, USA**
suciu@cs.washington.edu

Abstract

In this paper, we study the communication complexity for the problem of computing a conjunctive query on a large database in a parallel setting with p servers. In contrast to previous work, where upper and lower bounds on the communication were specified for particular structures of data (either data without skew, or data with specific types of skew), in this work we focus on *worst-case analysis* of the communication cost. The goal is to find worst-case optimal parallel algorithms, similar to the work of [17] for sequential algorithms.

We first show that for a single round we can obtain an optimal worst-case algorithm. The optimal load for a conjunctive query q when all relations have size equal to M is $O(M/p^{1/\psi^*})$, where ψ^* is a new query-related quantity called the *edge quasi-packing number*, which is different from both the edge packing number and edge cover number of the query hypergraph. For multiple rounds, we present algorithms that are optimal for several classes of queries. Finally, we show a surprising connection to the external memory model, which allows us to translate parallel algorithms to external memory algorithms. This technique allows us to recover (within a polylogarithmic factor) several recent results on the I/O complexity for computing join queries, and also obtain optimal algorithms for other classes of queries.

1998 ACM Subject Classification H.2.4 [Systems] Query Processing

Keywords and phrases conjunctive query, parallel computation, worst-case bounds

Digital Object Identifier 10.4230/LIPIcs.ICDT.2016.8

1 Introduction

The last decade has seen the development and widespread use of massively parallel systems that perform data analytics tasks over big data: examples of such systems are MapReduce [7], Dremel [16], Spark [21] and Myria [10]. In contrast to traditional database systems, where the computational complexity is dominated by the disk access time, the data now typically fits in main memory, and the dominant cost becomes that of communicating data and synchronizing among the servers in the cluster.

In this paper, we present a *worst-case analysis* of algorithms for processing of conjunctive queries (multiway join queries) on such massively parallel systems. Our analysis is based on the Massively Parallel Computation model, or MPC [4, 5]. MPC is a theoretical model where the computational complexity of an algorithm is characterized by both the *number of*

* This work is partially supported by NSF IIS-1247469, AitF 1535565, CCF-1217099 and CCF-1524246.

rounds (so the number of synchronization barriers) and the maximum amount of data, or *maximum load*, that each processor receives at every round.

The focus of our analysis on worst-case behavior of algorithms is a fundamentally different approach from previous work, where optimality of a parallel algorithm was defined for a specific input, or a specific family of inputs. Here we obtain upper bounds on the load of the algorithm across all possible types of input data. To give a concrete example, consider the simple join between two binary relations R and S of size M in bits (and m tuples), denoted $q(x, y, z) = R(x, z), S(y, z)$, and suppose that the number of servers is p. In the case where there is no data skew (which means in our case that the frequency of each value of the z variable in both R and S is at most m/p), it has been shown in [5] that the join can be computed in a single round with load $\tilde{O}(M/p)$ (where the notation $\tilde{O}$ hides a polylogarithmic factor depending on p), by simply hashing each tuple according to the value of the z variable. However, if the z variable is heavily skewed both in R and S (and in particular if there exists a single value of z), computing the query becomes equivalent to computing a cartesian product, for which we need $\Omega(M/p^{1/2})$ load. In this scenario, although for certain instances we can obtain better guarantees for the load, the heavily skewed instance is a *worst-case input*, in the sense that the lower bound $\Omega(M/p^{1/2})$ specifies the worst possible load that we may encounter. Our goal is to design algorithms for single or multiple rounds that are optimal with respect to such worst-case inputs and never incur larger load for any input.

Related Work. Algorithms for joins in the MPC model were previously analyzed in [4, 5]. In [4], the authors presented algorithms for one and multiple rounds on input data without skew (in particular when each value appears exactly once in each relation, which is a called a *matching database*). In [5], the authors showed that the HyperCube (HC) algorithm, first presented by Afrati and Ullman [2], can optimally compute any conjunctive query for a single round on data without skew. The work in [5] also presents one-round algorithms and lower bounds for skewed data but the upper and lower bounds do not necessarily coincide.

Several other computation models have been proposed in order to understand the power of MapReduce and related massively parallel programming paradigms [8, 13, 15, 1]. All these models identify the number of communication steps/rounds as a main complexity parameter, but differ in their treatment of the communication. Previous work [20, 14] has also focused on computing various graph problems in message-passing parallel models. In contrast to this work, where we focus on algorithms that require a constant number of rounds, the authors consider algorithms that need a large number of rounds.

Our setting and worst-case analysis can be viewed as the analogous version of the work of Ngo et al. [17] on worst-case optimal algorithms for multiway join processing. As we will show later, the worst-case instances for a given query q are different for the two settings in the case of one round, but coincide for all the families of queries we examine when we consider multiple rounds.

Our Contributions. We first present in Section 3 tight upper and lower bounds for the worst-case load of one-round algorithms for any full conjunctive query q without self-joins.[1] The optimal algorithm uses a different parametrization (share allocation) of the HyperCube algorithm for different parts of the input data, according to the values that are skewed. In

[1] The restriction to queries without self-joins is not limiting, since we can extend our result to queries with self-joins (by losing a constant factor) by treating copies of a relation as distinct relations. The parallel complexity for queries with projections is however an open question.

the case where all relation sizes are equal to M, the algorithm achieves an optimal load $\tilde{O}(M/p^{1/\psi^*(q)})$, where $\psi^*(q)$ is the *edge quasi-packing number* of the query q. An edge quasi-packing is an edge packing on any vertex-induced projection of the query hypergraph (in which we shrink hyperedges when we remove vertices).

In Section 4, we show that for any full conjunctive query q, any algorithm with a constant number of rounds requires a load of $\Omega(M/p^{1/\rho^*})$, where ρ^* is the *edge cover number*. We then present optimal (within a polylogarithmic factor) multi-round algorithms for several classes of join queries. Our analysis shows that some queries (such as the star query T_k) can be optimally computed using the optimal single-round algorithm from Section 3. However, other classes of queries, such as the cycle query C_k for $k \neq 4$, the line query L_k, or the Loomis-Whitney join LW_k require 2 or more rounds to achieve the optimal load. For example, we present an algorithm for the full query (or clique) K_k that uses $k-1$ rounds to achieve the optimal load (although it is open whether only 2 rounds are sufficient).

Finally, in Section 5 we present a surprising application of our results in the setting of external memory algorithms. In this setting, the input data does not fit into main memory, and the dominant cost of an algorithm is the I/O complexity: reading the data from the disk into the memory and writing data on the disk. In particular, we show that we can simulate an MPC algorithm in the external memory setting, and obtain almost-optimal (within a polylogarithmic factor) external memory algorithms for computing triangle queries; the same technique can be easily applied to other classes of queries.

2 Background

In this section, we introduce the MPC model and present the necessary terminology and technical tools that we will use later in the paper.

2.1 The MPC Model

We first review the *Massively Parallel Computation model (MPC)*, which allows us to analyze the performance of algorithms in parallel environments. In the MPC model, computation is performed by p servers, or processors, connected by a complete network of private channels. The computation proceeds in *steps*, or *rounds*, where each round consists of two distinct phases. In the *communication phase*, the servers exchange data, each by communicating with all other servers. In the *computation phase*, each server performs only local computation.

The input data of size M bits is initially uniformly partitioned among the p servers, that is, each server stores M/p bits of data. At the end of the execution, the output must be present in the union of the output of the p processors.

The execution of a parallel algorithm in the MPC model is captured by two parameters. The first parameter is the *number of rounds* r that the algorithm requires. The second parameter is the *maximum load* L, which measures the maximum amount of data (in bits) received by any server during any round.

All the input data will be distributed during some round, since we need to perform some computation on it. Thus, at least one server will receive at least data of size M/p. On the other hand, the maximum load will never exceed M, since any problem can be trivially solved in one round by simply sending the entire data to server 1, which can then compute the answer locally. Our typical loads will be of the form $M/p^{1-\varepsilon}$, for some parameter ε ($0 \leq \varepsilon < 1$) that depends on the query. For a similar reason, we do not allow the number of rounds to reach $r = p$, because any problem can be trivially solved in p rounds by sending

M/p bits of data at each round to server 1, until this server accumulates the entire data. In this paper we only consider the case $r = O(1)$.

2.2 Conjunctive Queries

In this paper we focus on a particular class of problems for the MPC model, namely computing answers to conjunctive queries over a database. We fix an input vocabulary $S_1, \ldots, S_\ell$, where each relation S_j has a fixed arity a_j; we denote $a = \sum_{j=1}^{\ell} a_j$. The input data consists of one relation instance for each symbol.

We consider full conjunctive queries (CQs) without self-joins, denoted as follows:

$$q(x_1, \ldots, x_k) = S_1(\ldots), \ldots, S_\ell(\ldots)\,.$$

The query is *full*, meaning that every variable in the body appears in the head (for example $q(x) = S(x, y)$ is not full), and *without self-joins*, meaning that each relation name S_j appears only once (for example $q(x, y, z) = S(x, y), S(y, z)$ has a self-join). We use $\mathsf{vars}(S_j)$ to denote the set of variables in the atom S_j, and $\mathsf{vars}(q)$ to denote the set of variables in all atoms of q. Further, k and ℓ denote the number of variables and atoms in q respectively. The *hypergraph* of a conjunctive query q is defined by introducing one node for each variable in the body and one hyperedge for each set of variables that occur in a single atom.

The *fractional 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 does not exceed 1. We let $\mathsf{pk}(q)$ denote the set of all fractional edge packings for q. The *fractional covering number* τ^* is the maximum sum of weights over all possible edge packings, $\tau^*(q) = \max_{\mathbf{u} \in \mathsf{pk}(q)} \sum_j u_j$.

The *fractional edge cover* associates a non-negative weight w_j to each atom S_j, such that for every variable x_i, the sum of the weights of the atoms that contain x_i is at least 1. The *fractional edge cover number* ρ^* is the minimum sum of weights over all possible fractional edge covers. The notion of the fractional edge cover has been used in the literature [3, 17] to provide lower bounds on the worst-case output size of a query (and consequently the running time of join processing algorithms).

For any $\mathbf{x} \subseteq \mathsf{vars}(q)$, we define the *residual query* $q_{\mathbf{x}}$ as the query obtained from q by removing all variables $\mathbf{x}$, and decreasing the arity of each relation accordingly (if the arity becomes zero we simply remove the relation). For example, for the *triangle query* $q(x, y, z) = R(x, y), S(y, z), T(z, x)$, the residual query $q_{\{x\}}$ is $q_{\{x\}}(y, z) = R(y), S(y, z), T(z)$. Similarly, $q_{\{x,y\}}(z) = S(z), T(z)$. Observe that every fractional edge packing of q is also a fractional edge packing of any residual query $q_{\mathbf{x}}$, but the converse is not true in general.

We now define the *fractional edge quasi-packing* to be any edge packing of a residual query $q_{\mathbf{x}}$ of q, where the atoms that have only variables in $\mathbf{x}$ get a weight of 0. Denote by $\mathsf{pk}^+(q)$ the set of all edge quasi-packings. It is straightforward to see that $\mathsf{pk}(q) \subseteq \mathsf{pk}^+(q)$; in other words, any packing is a quasi-packing as well. The converse is not true, since for example $(1, 1, 0)$ is a quasi-packing for the triangle query, but not a packing. The *edge quasi-packing number* ψ^* is the maximum sum of weights over all edge quasi-packings:

$$\psi^*(q) = \max_{\mathbf{u} \in \mathsf{pk}^+(q)} \sum_j u_j = \max_{\mathbf{x} \subseteq \mathsf{vars}(q)} \max_{\mathbf{u} \in \mathsf{pk}(q_{\mathbf{x}})} \sum_j u_j\,.$$

2.3 Previous Results

Suppose that we are given a full CQ q, and input such that relation S_j has size M_j in bits (we use m_j for the number of tuples). Let $\mathbf{M} = (M_1, \ldots, M_\ell)$ be the vector of the relation

sizes. For a given fractional edge packing $\mathbf{u} \in \mathsf{pk}(q)$, we define as in [5]:

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

Let us also define $L^{(q)}(\mathbf{M}, p) = \max_{\mathbf{u} \in \mathsf{pk}(q)} L(\mathbf{u}, \mathbf{M}, p)$. In our previous work [5], we showed that any algorithm that computes q in a single round with p servers must have load $L \geq L^{(q)}(\mathbf{M}, p)$. The instances used to prove this lower bound is the class of *matching databases*, which are instances where each value appears exactly once in the attribute of each relation. Hence, the above lower bound is not necessarily tight; indeed, as we will see in the next section, careful choice of skewed input instances can lead to a stronger lower bound.

The HyperCube algorithm. To compute conjunctive queries in the MPC model, we use the basic primitive of the *HyperCube (HC) algorithm*. The algorithm was first introduced by Afrati and Ullman [2], and was later called the *shares* algorithm; we use the name HC to refer to the algorithm with a particular choice of shares. The HC algorithm initially assigns to each variable x_i 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 $\mathcal{P} = [p_1] \times \cdots \times [p_k]$; in other words, servers are mapped into a k-dimensional hypercube. The HC algorithm then uses k independently chosen hash functions $h_i : \{1, \ldots, n\} \to \{1, \ldots, p_i\}$ (where n is the domain size) and sends each tuple t of relation S_j to all servers in the destination subcube of t:

$$\mathcal{D}(t) = \{\mathbf{y} \in \mathcal{P} \mid \forall x_i \in \mathsf{vars}(S_j) : h_i(t[x_i]) = \mathbf{y}_i\}$$

where $t[x_i]$ denotes the value of tuple t at the position of the variable x_i. After the tuples are received, each server locally computes q for the subset of the input that it has received.

If the input data has no skew, the above vanilla version of the HC algorithm is optimal for a single round. The lemma below presents the specific conditions that define skew, and will be frequently used throughout the paper.

▶ **Lemma 1** (Load Analysis for HC [5]). *Let $\mathbf{p} = (p_1, \ldots, p_k)$ be the optimal shares of the HC algorithm. Suppose that for every relation S_j and every tuple t over the attributes $U \subseteq [a_j]$ we have that the frequency of t in relation S_j is $m_{S_j}(t) \leq m_j / \prod_{i \in U} p_i$. Then with high probability the maximum load per server is $\tilde{O}(L^{(q)}(\mathbf{M}, p))$.*

3 One-Round Algorithms

In this section, we present tight upper and lower bounds for the worst-case load of one-round algorithms that compute conjunctive queries. Thus, we identify the database instances for which the behavior in a parallel setting is the worst possible. Surprisingly, these instances are often different from the ones that provide a worst-case running time in a non-parallel setting.

As an example, consider the triangle query $C_3 = R(x, y), S(y, z), T(z, x)$, where all relations have m tuples (and M in bits). It is known from [3] that the class of inputs that will give a worst-case output size, and hence a worst-case running time, is one where each relation is a $\sqrt{m} \times \sqrt{m}$ fully bipartite graph. In this case, the output has $m^{3/2}$ tuples. The load needed to compute C_3 on this input in a single round is $\Omega(M/p^{2/3})$, and can be achieved by using the HyperCube algorithm [4] with shares $p^{1/3}$ for each variable. Now, consider the instance where relations R, T have a single value at variable x, which participates in all the m tuples in R and T; S is a matching relation with m tuples. In this case, the output has m

tuples (and so M bits), and thus is smaller than the worst-case output. However, as we will see next, we can show that any one-round algorithm that computes the triangle query for the above input structure requires $\Omega(M/p^{1/2})$ maximum load.

3.1 An Optimal Algorithm

We present here a worst-case optimal one-step algorithm that computes a conjunctive query q. Recall that the HC algorithm achieves an optimal load on data without skew [5]. In the presence of skew, we will distinguish different cases, and for each case we will apply a different parametrization of the HC algorithm, using different shares.

We say that a value h in relation S_j is a *heavy hitter* in S_j if the frequency of this particular value in S_j, denoted $m_{S_j}(h)$, is at least m_j/p, where m_j is the number of tuples in the relation. Given an output tuple t, we say that t is *heavy at variable* x_i if the value $t[x_i]$ is a heavy hitter in at least one of the atoms that include variable x_i.

We can now classify each tuple t in the output depending on the positions where t is heavy. In particular, for any $\mathbf{x} \subseteq \mathsf{vars}(q)$, let $q^{[\mathbf{x}]}(I)$ denote the subset of the output that includes only the output tuples that are heavy at exactly the variables in $\mathbf{x}$. Observe that the case $q^{[\emptyset]}(I)$ denotes the case where the tuples are light at all variables; we know from an application of Lemma 1 that this case can be handled by the standard HC algorithm. For each of the remaining $2^k - 1$ possible sets $\mathbf{x} \subseteq \mathsf{vars}(q)$, we will run a different variation of the HC algorithm with different shares, which will allow us to compute $q^{[\mathbf{x}]}(I)$ with the appropriate load. Our algorithm will compute all the partial answers in parallel for each $\mathbf{x} \subseteq \mathsf{vars}(q)$, and thus requires only a single round.

The key idea is to apply the HC algorithm by giving a non-trivial share only to the variables that are not in $\mathbf{x}$; in other words, every variable in $\mathbf{x}$ gets a share of 1. In particular, we will assign to the remaining variables the shares we would assign if we would execute the HC algorithm for the residual query $q_{\mathbf{x}}$. We will thus choose the shares by assigning $p_i = p^{e_i}$ for each $x_i \in \mathbf{x}$ and solving the following linear program:

$$\begin{aligned} \text{minimize} \quad & \lambda \\ \text{subject to} \quad & \sum_{i:x_i \notin \mathbf{x}} -e_i \geq -1 \\ \forall j \text{ s.t. } S_j \in \mathsf{atoms}(q_{\mathbf{x}}) : & \sum_{i:x_i \in \mathsf{vars}(S_j) \setminus \mathbf{x}} e_i + \lambda \geq \mu_j \\ \forall i \text{ s.t. } x_i \notin \mathbf{x} : & e_i \geq 0, \quad \lambda \geq 0 \end{aligned} \tag{2}$$

For each variable $x_i \in \mathbf{x}$, we set $e_i = 0$ and thus the share is $p_i = 1$. We next present the analysis of the load for the above algorithm.

▶ **Theorem 2.** *Any full conjunctive query q with input relation sizes $\mathbf{M}$ can be computed in the MPC model in a single round using p servers with maximum load*

$$L = \tilde{O}\left(\max_{\mathbf{x} \subseteq \mathit{vars}(q)} L^{(q_{\mathbf{x}})}(\mathbf{M}, p)\right).$$

Proof. Let us fix a set of variables $\mathbf{x} \subseteq \mathsf{vars}(q)$; we will show that the load of the algorithm that computes $q^{[\mathbf{x}]}(I)$ is $\tilde{O}(L^{(q_{\mathbf{x}})}(\mathbf{M}, p))$. The upper bound then follows from the fact that we are running in parallel algorithms for all partial answers.

Indeed, let us consider how each relation S_j is distributed using the shares assigned. We distinguish two cases. If an atom S_j contains variables that are only in $\mathbf{x}$, then the whole

relation will be broadcast to all the p servers. However, observe that the part of S_j that contributes to $q^{[\mathbf{x}]}(I)$ is of size at most p^{a_j}, where a_j is the arity of the relation.

Otherwise, we will show that for every tuple J of values over variables $\mathbf{v} \subseteq \mathsf{vars}(S_j)$, we have that the frequency of J is at most $m_j / \prod_{i:x_i \in \mathbf{v}} p_i$. Indeed, if $\mathbf{v}$ contains only variables from $\mathbf{x}$, then by construction $\prod_{i:x_i \in \mathbf{v}} p_i = 1$; we observe then that the frequency is always at most m_j. If $\mathbf{v}$ contains some variable $x_i \in \mathbf{v} \setminus \mathbf{x}$, then the tuple J contains at position x_i a value that appears at most m_j/p times in relation S_j, and since $\prod_{i:x_i \in \mathbf{v}} p_i \leq p$ the claim holds. We can now apply Lemma 1 to obtain that for relation S_j, the load will be $\tilde{O}(M_j/(\prod_{i:x_i \in \mathsf{vars}(S_j)\setminus \mathbf{x}} p_i))$. Summing over all atoms in the residual query $q_{\mathbf{x}}$, and assuming that $m_j \gg p$ (and in particular that p^{a_j} is always much smaller than the load), we obtain that the load will be $\tilde{O}(\max_{j:S_j \in \mathsf{atoms}(q_{\mathbf{x}})} M_j/(\prod_{i:x_i \in \mathsf{vars}(S_j)\setminus \mathbf{x}} p_i))$, which by an LP duality argument is equal to $\tilde{O}(L^{(q_{\mathbf{x}})}(\mathbf{M}, p))$. ◀

When all relation sizes are equal, that is, $M_1 = M_2 = \cdots = M_\ell = M$, the formula for the maximum load becomes $\tilde{O}(M/p^{1/\psi^*(q)})$, where $\psi^*(q)$ is the edge quasi-packing number, which we have defined as $\psi^*(q) = \max_{\mathbf{x} \subseteq \mathsf{vars}(q)} \max_{\mathbf{u} \in \mathsf{pk}(q_{\mathbf{x}})} \sum_j u_j$. We will discuss about the quantity $\psi^*(q)$ in detail in Section 3.3. We will see next how the above algorithm applies to the triangle query C_3.

▶ **Example 3.** We will describe first how the algorithm works when each relation has size M (and m tuples). There are three different share allocations, for each choice of heavy variables (all other cases are symmetrical).

$\mathbf{x} = \emptyset$: we consider only tuples with values of frequency $\leq m/p$. The HC algorithm will assign a share of $p^{1/3}$ to each variable, and the maximum load will be $\tilde{O}(M/p^{2/3})$.

$\mathbf{x} = \{x\}$: the tuples have a heavy hitter value at variable x, either in relation R or T or in both. The algorithm will give a share of 1 to x, and shares of $p^{1/2}$ to y and z. The load will be $\tilde{O}(M/p^{1/2})$.

$\mathbf{x} = \{x, y\}$: both x and y are heavy. In this case we broadcast the relation $R(x, y)$, which will have size at most p^2, and assign a share of p to z. The load will be $\tilde{O}(M/p)$.

Notice finally that the case where $\mathbf{x} = \{x, y, z\}$ can be handled by broadcasting all necessary information. The load of the algorithm is the maximum of the above quantities, thus $\tilde{O}(M/p^{1/2})$. When the size vector is $\mathbf{M} = (M_1, M_2, M_3)$, the load achieved becomes $\tilde{O}(L)$, where: $L = \max\left\{\frac{M_1}{p}, \frac{M_2}{p}, \frac{M_3}{p}, \sqrt{\frac{M_1 M_2}{p}}, \sqrt{\frac{M_2 M_3}{p}}, \sqrt{\frac{M_1 M_3}{p}}\right\}$.

3.2 Lower Bounds

We present here a worst-case lower bound for the load of one-step algorithms for computing conjunctive queries in the MPC model, when the information known is the cardinality statistics $\mathbf{M} = (M_1, \ldots, M_\ell)$. The lower bound matches the upper bound in the previous section, hence proving that the one-round algorithm is worst-case optimal. We give a self-contained proof of the result in the full version of this paper, but many of the techniques used can be found in previous work [4, 5], where we proved lower bounds for skew-free data and for input data with known information about the heavy hitters.

▶ **Theorem 4.** *Fix cardinality statistics* $\mathbf{M}$ *for a full conjunctive query* q*. Consider any deterministic MPC algorithm that runs in one communication round on* p *servers and has maximum load* L *in bits. Then, for any* $\mathbf{x} \subseteq \mathsf{vars}(q)$*, there exists a family of (random) instances for which the load* L *will be:*

$$L \geq \min_j \frac{1}{4a_j} \cdot L^{(q_{\mathbf{x}})}(\mathbf{M}, p)\,.$$

Since $a_j \geq 1$, Theorem 4 implies that for any query q there exists a family of instances such that any one-round algorithm that computes q must have load $\Omega(\max_{\mathbf{x} \subseteq \mathsf{vars}(q)} L^{(q_{\mathbf{x}})}(\mathbf{M}, p))$.

3.3 Discussion

We present here several examples for the load of the one-round algorithm for various classes of queries, and also discuss the edge quasi-packing number $\psi^*(q)$ and its connection with other query-related quantities.

Recall that we showed that when all relation sizes are equal to M, the load achieved is of the form $\tilde{O}(M/p^{1/\psi^*(q)})$, where $\psi^*(q)$ is the quantity that maximizes the sum of the weights of the edge quasi-packing. $\psi^*(q)$ is in general different from both the fractional covering number $\tau^*(q)$, and from the fractional edge cover number $\rho^*(q)$. Indeed, for the triangle query C_3 we have that $\rho^*(C_3) = \tau^*(C_3) = 3/2$, while $\psi^*(C_3) = 2$. Here we should remind the reader that τ^* describes the load for one-round algorithms on data without skew, which is $O(M/p^{1/\tau^*(q)})$. Also, ρ^* characterizes the maximum possible output of a query q, which is $M^{\rho^*(q)}$. We can show the following relation between the three quantities:

▶ **Lemma 5.** *For every conjunctive query q, $\psi^*(q) \geq \max\{\tau^*(q), \rho^*(q)\}$.*

Proof. Since any edge packing is also an edge quasi-packing, it is straightforward to see that $\tau^*(q) \leq \psi^*(q)$ for every query q.

To show that $\rho^*(q) \leq \psi^*(q)$, consider the optimal (minimum) edge cover $\mathbf{u}$; we will show that this is also an edge quasi-packing. First, observe that for every atom S_j, there must exist at least one variable $x \in \mathsf{vars}(S_j)$ such that $\sum_{j:x \in \mathsf{vars}(S_j)} u_j = 1$. Indeed, suppose that for every variable in S_j we have that the sum of the weights strictly exceeds 1; then, we can obtain a better edge cover by slightly decreasing u_j, which is a contradiction.

Now, let $\mathbf{x}$ be the set of variables such that their cover in $\mathbf{u}$ strictly exceeds 1, and consider the residual query $q_{\mathbf{x}}$. By our previous claim, every relation in q is still present in $q_{\mathbf{x}}$, since every relation includes a variable with cover exactly one. Further, for every variable $x \in \mathsf{vars}(q_{\mathbf{x}})$ we have $\sum_{j:x \in \mathsf{vars}(S_j)} u_j = 1$, and hence $\mathbf{u} \in \mathsf{pk}(q_{\mathbf{x}})$. ◀

In Table 1 we have computed the quantities τ^*, ρ^*, ψ^* for several classes of queries of interest: the star query T_k, the spiked star query SP_k, the cycle query C_k, the line query L_k, the Loomis-Whitney join LW_k, the generalized semi-join query W_k and the clique (or full) query K_k. We next present some of these queries in more detail.

▶ **Example 6.** Consider the star query T_k, which generalizes the simple join between relations. As we can see, the optimal edge packing cannot be more than 1, since every relation includes the variable z. To obtain the maximum edge quasi-packing, we simply consider the residual query q_z that removes the common variable z: then, we can pack each relation with weight one, thus achieving a sum of k. Notice that this is an example which shows that τ^* and ψ^* cannot be within a constant factor.

▶ **Example 7.** Consider the full/clique query K_k, which includes all possible binary relations among the k variables. Here the optimal edge packing is achieved by assigning a weight of $1/(k-1)$ to each relation; the corresponding share allocation for the HC algorithm assigns an equal share of $p^{1/k}$ to each variable. For the optimal edge quasi-packing, consider the residual query $(K_k)_{x_1}$, and notice that it includes $(k-1)$ unary relations, one for each of $x_2, \ldots, x_k$. Hence, we can obtain an edge packing by assigning a weight of 1 to each, which shows that $\psi^*(K_k) = k$.

Table 1 Computing the optimal edge packing τ^*, edge cover ρ^* and edge quasi-packing ψ^* for several classes of conjunctive queries.

conjunctive query	τ^*	ρ^*	ψ^*
$T_k = \bigwedge_{j=1}^{k} S_j(z, x_j)$	1	k	k
$SP_k = \bigwedge_{i=1}^{k} R_i(z, x_i), S_i(x_i, y_i)$	k	$k+1$	$k+1$
$C_k = \bigwedge_{j=1}^{k} S_j(x_j, x_{(j \bmod k)+1})$	$k/2$	$k/2$	$\lceil 2(k-1)/3 \rceil$
$L_k = \bigwedge_{j=1}^{k} S_j(x_{j-1}, x_j)$	$\lceil k/2 \rceil$	$\lceil (k+1)/2 \rceil$	$\lceil 2k/3 \rceil$
$LW_k = \bigwedge_{I \subseteq [k], \lvert I \rvert = k-1} S_I(\bar{x}_I)$	$k/(k-1)$	$k/(k-1)$	2
$W_k = R(x_1, \ldots, x_k) \bigwedge_{j=1}^{k} S_j(x_j)$	k	1	k
$K_k = \bigwedge_{1 \le i < j \le k} S_{i,j}(x_i, x_j)$	$k/2$	$k/2$	k

▶ **Example 8.** Consider the cycle query C_k. The optimal edge packing assigns a weight of $1/2$ to each edge; the corresponding share allocation for the HC algorithm gives an equal share of $p^{1/k}$ to each variable.

To find the best $\mathbf{x}$ for the optimal edge quasi-packing, we will pick every third variable: $x_1, x_4, \ldots$. This creates $\lfloor k/3 \rfloor$ copies of the query $S_1(x_1), S_2(x_1, x_2), S_3(x_2)$, which has an edge packing of size 2 (assign weight 1 to S_1, S_3). If $k = 3m$ or $k = 3m+1$, these copies cover the whole query. If $k = 3m + 2$, we can add one more edge with weight 1 to the packing.

4 Multi-round Algorithms

In this section, we present algorithms for multi-round computation of several conjunctive queries in the case where the relation sizes are all equal to M. We also prove a lower bound that proves that they are (almost) optimal.

4.1 Multi-round Lower Bound

We prove here a general lower bound for any algorithm that computes conjunctive queries using a constant number of rounds. Observe that the lower bound is expressed in terms of number of tuples (and not bits); our upper bounds will be expressed in terms of bits, and thus will be a $\log(n)$ factor away from the lower bound, where n is the domain size.

▶ **Theorem 9.** *Let q be a conjunctive query. Then, there exists a family of instances where relations have the same size M in bits (and m in tuples) such that every algorithm that computes q with p servers using a constant number of rounds requires load $\Omega(m/p^{1/\rho^*(q)})$.*

Proof. In order to prove the lower bound, we will use a family of instances that give the maximum possible output when every input relation has at most m tuples, which is $m^{\rho^*(q)}$ (see [3]). We also know how we can construct such a worst-case instance: for each variable x_i we assign an integer n_i (which corresponds to the domain size of the variable), and we define each relation as the cartesian product of the domains of the variables it includes: $\times_{i:x_i \in \mathsf{vars}(S_j)}[n_i]$. The output size then will be $\prod_i n_i = m^{\rho^*(q)}$ (using a LP duality argument).

We now define the following random instance I as input for the query q: for each relation S_j, we choose each tuple from the full cartesian product of the domains independently at random with probability $1/2$. It is straightforward to see that the expected size of the output is $E[|q(I)|] = (1/2)^{\beta} \prod_i n_i$, where β is the maximum number of relations where any variable

occurs (and thus a constant depending on the query). Using Chernoff's bound we can claim an even stronger result: the output size will be $\Theta(m^{\rho^*(q)})$ with high probability (the failure probability is exponentially small in m).

Now, assume that algorithm $\mathcal{A}$ computes q with load L (in bits) in r rounds. Then, each server receives at most $L' = r \cdot L$ bits. Fix some server and let msg be the whole sequence of bits received by this server during the computation; hence, $|\mathsf{msg}| \leq L'$. We will next compute how many tuples from S_j are known by the server, denoted $K_{\mathsf{msg}}(S_j)$. W.l.o.g. we can assume that all L' bits of msg contain information from relation S_j.

We will show that the probability of the event $K_{\mathsf{msg}}(S_j) > (1+\delta)L'$ is exponentially small on δ. Let $m_j = \prod_{i:x_i \in \mathsf{vars}(S_j)} n_i \leq m$. Observe first that the total number of message configurations of size L' is at most $2^{L'}$. Also, since the size of the full cartesian product is m_j, msg can encode at most $2^{m_j-(1+\delta)L'}$ relations S_j (if $m_j < (1+\delta)L'$, then trivially the probability of the event is zero, and S_j will have "few" tuples). It follows that

$$P(K_{\mathsf{msg}} > (1+\delta)L') < 2^{L'} \cdot 2^{m_j-(1+\delta)L'} \cdot (1/2)^{m_j} = (1/2)^{\delta L'}.$$

So far we have shown that with high probability each server knows at most L' tuples from each relation S_j, and further that the total number of output tuples is $\Theta(m^{\rho^*(q)})$. However, if a server knows L' tuples from each relation, using the AGM bound from [3], it can output at most $(rL)^{\rho^*(q)}$ tuples. The result follows by summing over the output of all p servers, and using the fact that the algorithm has only a constant number of rounds. ◀

The theorem implies that whenever $\psi^*(q) = \rho^*(q)$ the one-round algorithm is essentially worst-case optimal, and using more rounds will not result in an algorithm with better load. As a result, and following our discussion in the previous section, the classes of queries T_k and SP_k can be optimally computed in a single round. This may seem counterintuitive, but recall that we study worst-case optimal algorithms; there may be instances where using more rounds is desirable, but our goal is to match the load for the worst such instance.

We will next present algorithms that match (within a logarithmic factor) the above lower bound using strictly more than one round. We start with the algorithm for the triangle query C_3, in order to demonstrate our novel technique and prove a key result (Lemma 10) that we will use later in the section.

4.2 Warmup: Computing Triangles in 2 Rounds

The main component of the algorithm that computes triangles is a parallel algorithm that computes the join $S_1(x,z), S_2(y,z)$ in a single round, for the case where skew appears exclusively in one of the two relations. If the relations have size M_1, M_2 respectively, then we have shown that the load can be as large as $\sqrt{M_1 M_2/p}$. However, in the case of one-sided skew, we can compute the join with maximum load only $\tilde{O}(\max\{M_1, M_2\}/p)$.

▶ **Lemma 10.** *Let $q = S_1(x,z), S_2(y,z)$, and let m_1 and m_2 be the relation sizes (in tuples) 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, $m_{S_1}(z)$, is at most m/p, then we can compute q in a single round with p servers and load (in bits) $\tilde{O}(M/p)$, where $M = 2m\log(n)$ (n is the domain size).*

Proof. We say that a value h is a heavy hitter in S_2 if the degree of h in S_2 is $m_{S_2}(h) > m/p$. By our assumption, there are no heavy hitters in relation S_1.

For the values h that are not heavy hitters in S_2, we can compute the join by applying the standard HC algorithm (which is a hash-join that assigns a share of p to z); the load analysis of Lemma 1 will give us a load of $\tilde{O}(M/p)$ with high probability.

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 $q_z = S_1'(x), S_2'(y)$, 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 $|S_1'| \leq m/p$ 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 servers, we simply use the simple broadcast join that assigns a share of p to variable x and 1 to y. A simple analysis will give us that the load (in tuples) for each heavy hitter h is

$$\tilde{O}\left(\frac{|S_2'|}{p_h} + |S_1'|\right) = \tilde{O}\left(\frac{m_{S_2}(h)}{p \cdot m_{S_2}(h)/m} + m/p)\right) = \tilde{O}(m/p)\,.$$

Finally, observe that the total number of servers we need is $\sum_h p_h \leq 2p$, hence we have used an appropriate amount of the available p servers. ◀

Thus, we can optimally compute joins in a single round in the presence of one-sided skew. We can apply Lemma 10 to obtain a useful corollary for the *semi-join* query $q = R(z), S(y,z)$. Indeed, notice that we can extend R to a binary relation $R'(x,z)$, where x is a dummy variable that takes a single value; then, the semi-join becomes essentially a join, where R' has no skew, since the degree of z in R' will be always one. Consequently:

▶ **Corollary 11.** *Consider the semi-join query $q = R(z), S(y,z)$, and let M_1 and M_2 be the relation sizes of R, S respectively in bits. Then we can compute q in a single round with p servers and load $\tilde{O}(\max\{M_1, M_2\}/p)$.*

We now outline the algorithm for computing triangles using two rounds. The central idea in the algorithm is to identify the values that create skew in the computation, and spread this computation into more rounds.

▶ **Theorem 12.** *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 $\tilde{O}(M/p^{2/3})$ 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 are not heavy at any variable. Indeed, if for every value we have that the degree is at most $m/p^{1/3}$, then the load analysis (Lemma 1) tells us that we can compute the output in a single round with load $\tilde{O}(M/p^{2/3})$ using the HC algorithm that allocates a share of $p^{1/3}$ to each variable.

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 $2p^{1/3}$ such heavy values for x_1 ($p^{1/3}$ for S_1 and $p^{1/3}$ for S_3). For each heavy value h, we assign an *exclusive* set of $p' = p^{2/3}$ servers to compute the query $q[h/x_1] = S_1(h,x_2), S_2(x_2,x_3), S_3(x_3,x_1)$, which is equivalent to computing the residual query $q' = S_1'(x_2), S_2(x_2,x_3), S_3'(x_3)$.

To compute q' with p' servers, we use 2 rounds. In the first round, we compute in parallel the semi-join queries $S_{12}(x_2,x_3) = S_1'(x_2), S_2(x_2,x_3)$ and $S_{23}(x_2,x_3) = S_2(x_2,x_3), S_3'(x_3)$. Since $|S_1'| \leq m$ and $|S_2'| \leq m$, we can apply Corollary 11 for semi-join computation to obtain that we can achieve this computation with load (in tuples) $\tilde{O}(m/p') = \tilde{O}(m/p^{2/3})$. Observe that the intermediate relations S_{12}, S_{23} have size at most m. In the second round, we simply perform the intersection of the relations S_{12}, S_{23}; this can be achieved with tuple load $O(m/p') = O(m/p^{2/3})$.[2] ◀

[2] Observe that the load for computing the intersection of two or more relations does not have any additional logarithmic factors.

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 better load. This confirms our intuition that with more rounds we can reduce the maximum load. Moreover, observe that the load achieved matches the multi-round lower bound (within a polylogarithmic factor).

4.3 Computing General CQs

We now generalize the ideas of the above example, and extend our results to several standard classes of conjunctive queries. Throughout this section, we assume that all relations have the same size M in bits (and m in tuples). We present in detail optimal multiround algorithms for odd and even cycles, which both achieve a maximum load of $\tilde{O}(M/p^{2/k})$ for C_k. The algorithm uses as a component an optimal algorithm that computes the line query L_k.

▶ **Lemma 13.** *The line query $L_k = S_1(x_0, x_1), S_2(x_1, x_2), \ldots, S_k(x_{k-1}, x_k)$ can be computed by an MPC algorithm with a constant number of rounds and load $\tilde{O}(M/p^{1/\lceil (k+1)/2 \rceil})$.*

We then briefly present our algorithmic results for Loomis-Whitney joins and Clique queries; the detailed proofs of the desired load are in the full version of this paper.

4.3.1 Odd Cycles

We will first show how we can compute any odd cycle C_k; the algorithm is a generalization of the method for computing triangle queries presented as a warmup example.

We say that a value h is *heavy* for variable x_i if for relation S_{i-1} or S_i, we have $m_i(h) > m/p^{1/k}$ or $m_{i-1}(h) > m/p^{1/k}$. We first compute the answers for the tuples that are not heavy at any position. Lemma 1 implies that we can compute the output in a single round with load $\tilde{O}(M/p^{2/k})$, by applying the vanilla HC algorithm for cycles, where each variable has equal share $p^{1/k}$.

We next compute the tuples that are heavy at variable x_1 (we similarly do this for every variable x_i); observe that there are at most $2p^{1/k}$ such values. For each such heavy value h, we will assign an exclusive number of $p' = p^{1-1/k}$ servers, such that the total number of servers we use is $(2p^{1/k}) \cdot p' = \Theta(p)$, and using these servers we will compute the query $q[h/x_1] = S_1(h, x_2), \ldots, S_k(x_k, h)$, which amounts to computing the residual query $q' = q_{x_1}$:

$$q' = S_1'(x_2), S_2(x_2, x_3), \ldots, S_{k-1}(x_{k-1}, x_k), S_k'(x_k)\,.$$

To compute q' with p' servers we need two rounds of computation. In the first round, we compute in parallel the two semi-joins

$$S_{1,2}(x_2, x_3) = S_1'(x_2), S_2(x_2, x_3), \qquad S_{k,k-1}(x_{k-1}, x_k) = S_{k-1}(x_{k-1}, x_k), S_k'(x_k)$$

which can be achieved with tuple load $\tilde{O}(m/p') = \tilde{O}(m/p^{1-1/k})$, since $|S_1'| \leq m$ and $|S_k'| \leq m$ (by applying Corollary 11). Since for any $k \geq 3$ we have $1 - 1/k \geq 2/k$, the load for the first round will be $\tilde{O}(M/p^{2/k})$. For the second round, we compute the query

$$q'' = S_{1,2}(x_2, x_3), S_3(x_3, x_4), \ldots, S_{k-1}(x_{k-1}, x_k), S_{k,k-1}(x_{k-1}, x_k)$$

which is equivalent to computing the line query L_{k-2}, where each relation has size at most m; we know from Lemma 13 that we can compute such a query with tuple load $\tilde{O}(m/p'^{1/\lceil (k-1)/2 \rceil})$ using multiple rounds. For the final step of the proof, recall that $p' = p^{1-1/k}$. Then:

$$\frac{k-1}{k} \cdot \frac{1}{\lceil (k-1)/2 \rceil} = \frac{k-1}{k} \cdot \frac{2}{k-1} = 2/k\,.$$

Thus, the load for the second round will be $\tilde{O}(M/p^{2/k})$ as well.

4.3.2 Even Cycles

For even length cycles, our previous argument does not work, and we have to use a different approach. We say that a value h is δ-heavy, for some $\delta \in [0,1]$, if the degree of h is at least m/p^δ in some relation. We distinguish two different cases:

1. Suppose that there exist two variables $x_i, x_{i'}$ such that $(i - i')$ is an odd number, x_i is δ-heavy, $x_{i'}$ is δ'-heavy, and $\delta + \delta' \leq 2/k$. Observe that there are at most $p^{\delta+\delta'} \leq p^{2/k}$ such pairs of heavy values: for each such pair, we assign $p' = p^{1-2/k}$ explicit servers to compute the residual query $q' = (C_k)_{x_i,x_j}$ in two rounds. We now consider two subcases. If $i' = i+1$, then $x_i, x_{i'}$ belong in the same relation S_i. Then, by performing the semi-join computations in the first round, we reduce the computation of the next rounds to the residual query L_{k-3}, which requires tuple load $\tilde{O}(m/p'^{1/\lceil (k-2)/2 \rceil}) = \tilde{O}(m/p^{2/k})$, since k is even. Otherwise, if $x_i, x_{i'}$ are not in the same relation, we still do the semi-joins in the first round, and then notice that in the subsequent rounds we need to compute the cartesian product of two line queries, L_α, L_β, where $\alpha + \beta = k - 4$ and both are odd numbers. To perform this cartesian product, we will split the p' servers into a $p^{(\alpha+1)/k} \times p^{(\beta+1)/k}$ grid, and within each row/column compute the line queries. Then, the tuple load will be $\tilde{O}(m/p^{((\alpha+1)/k)\cdot(1/\lceil(\alpha+1)/2\rceil)}) = \tilde{O}(m/p^{((\beta+1)/k)\cdot(1/\lceil(\beta+1)/2\rceil)}) = \tilde{O}(m/p^{2/k})$.
2. Otherwise, define δ_{even} as the largest number in $[0,1]$ such that for every even variable the frequency is at most $m/p^{\delta_{even}}$. Similarly define δ_{odd}. Since we do not fall in the previous case, it must be that $\delta_{even} + \delta_{odd} \geq 2/k$. W.l.o.g. assume that $\delta_{even} \geq \delta_{odd}$. Then, consider the HC algorithm with the following share allocation: for odd variables assign $p_o = p^{\delta_{odd}}$, and for even variables assign $p_e = p^{2/k - \delta_{odd}}$. Since the odd variables have degree at most $m/p^{\delta_{odd}}$, there are no skewed values there. As for the even variables, their degree is at most $m/p^{\delta_{even}} \leq m/p^{2/k-\delta_{odd}} = m/p_e$. Hence, the tuple load achieved will be $\tilde{O}(m/(p_o p_e) = \tilde{O}(m/p^{2/k})$. In the case where p_e is ill-defined because $\delta_{odd} > 2/k$, we also have that $\delta_{even} > 2/k$ and in this case we can just apply the standard HC algorithm that assigns a share of $p^{1/k}$ to every variable.

4.3.3 Other Conjunctive Queries

For the Loomis-Whitney (LW) join, the algorithmic idea is the same as the one we used for even cycles (notice that LW_3 is the triangle query C_3).

▶ **Lemma 14.** *The LW join $LW_k = S_1(x_2, \ldots, x_k), S_2(x_1, x_3, \ldots, x_k), \ldots, S_k(x_1, \ldots, x_{k-1})$ can be computed by an MPC algorithm in 2 rounds with load $\tilde{O}(M/p^{1-1/k})$.*

For the clique queries, we have the following result:

▶ **Lemma 15.** *The clique query $K_k = \bigwedge_{1 \leq i < j \leq k} S_{i,j}(x_i, x_j)$ can be computed by an MPC algorithm in $k-1$ rounds with load $\tilde{O}(M/p^{2/k})$ for any $k \geq 3$.*

Finally, we show an almost optimal algorithm for queries q that contain an atom which includes all the variables in the body of q.

▶ **Lemma 16.** *Let q be a query that contains an atom R, such that $\mathsf{vars}(R) = \mathsf{vars}(q)$. Then, q can be computed by an MPC algorithm with $\tilde{O}(M/p)$ load.*

Notice that the generalized semi-join query W_k satisfies the property of the above lemma, and hence we can compute W_k with load $\tilde{O}(M/p)$ using two rounds (while using one round the load is $\Omega(M/p^{1/k})$).

5 Applications to 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 W *words*.[3] 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 B consecutive 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.

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* [6, 12, 18, 11]. In this setting, the input is an undirected graph, and the goal is to list all triangles in the graph. In [18] and [11], 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 [11], where the authors design a deterministic algorithm that enumerates triangles in $O(|E|^{3/2}/(\sqrt{W}B))$ I/Os, where E is the number of edges in the graph. The authors in [11] actually consider a more general class of join problems, the so-called *Loomis-Whitney enumeration*. In [19], 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.

5.1 Simulating an MPC Algorithm

We will show how a parallel algorithm in the tuple-based MPC model can help us construct an external memory algorithm. The *tuple-based MPC model* is a restriction of the MPC model, where only tuples from subqueries of q can be communicated, and moreover the communication can take a very specific 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 initially available to the algorithm. Such statistical information is the size of the relations, or information about the heavy hitters in the data.[4] All of the algorithms that we have presented so far in the previous sections satisfy the above assumption.

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 memory; hence, the load L will correspond to the size of the internal memory W. 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

[3] The size of the main memory is typically denoted by M, but we use W to distinguish from the relation size in the previous sections.

[4] 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 $O(|I|/B)$ I/Os.

a rigorous way. We should also note that previous work [9] has studied the simulation of MapReduce algorithms on a parallel external memory model.

Let $\mathcal{A}$ be a tuple-based MPC algorithm that computes query q over input I using r rounds with load $L(I,p)$. We show next how 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 store tuples of the form (t, s) to denote that tuple t resides in server s.

To initialize $\mathcal{B}$, we 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, which will be of the form (t, s), are sorted according to the attribute s.
2. All tuples are loaded in memory in chunks of size W, in the order by which they were sorted in the external memory. If we choose p such that $r \cdot L(I,p) \leq W$, we can fit in the internal memory all the tuples of any server s at round k. [5] Hence, we first read into the internal memory the tuples for server 1, then server 2, and so on. For each server s, we replicate in the internal memory 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, writing to the internal and external memory simulates the communication step, where data is exchanged between servers. The algorithm $\mathcal{B}$ produces the correct result, since by the choice of p we guarantee that we can load enough data in the memory to simulate the local computation of $\mathcal{A}$ at each server. Observe that we do not need to write the final result back 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 identify the choice for p; recall that we must make sure that $r \cdot L(I,p) \leq W$. Hence, we must choose p_o such that $p_o = \min_p\{L(I,p) \leq W/r\}$. We next analyze the I/O cost of algorithm $\mathcal{B}$ for this choice of p_o.

Analysis. The initialization I/O cost for the algorithm is $|I|/B$. To analyze the cost for a given round $k = 1, \ldots, r$, 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 $k \cdot p_o \cdot L(I,p_o) \leq p_o W$, from our definition of p_o.

For the first step that requires sorting the data, we will not use a sorting algorithm, but instead we will partition the data into p parts, and then concatenate the parts (this is

[5] The quantity $L(I,p)$ measures the maximum amount of data received during any round. Since data is not destroyed, over r rounds a server can receive as much as $r \cdot L(I,p)$ data. All of this data must fit into the memory of size W, since the decisions of each server depend on all the data received.

possible only if p_o is smaller than the memory W, i.e. it must be $p_o \leq W$). We can do this with a cost of $O(p_o W/B)$ I/Os. The second step of loading the tuples into memory has a cost of $p_o W/B$, 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 $p_o L(I, p_o) \leq p_o W/r$. Hence, the I/O cost will be $p_o W/(rB)$.

Summing the I/O cost of all three steps over r rounds, we obtain that the I/O cost of the constructed algorithm $\mathcal{B}$ will be:

$$O\left(\frac{|I|}{B} + \sum_{k=1}^{r}\left(\frac{p_o W}{B} + \frac{p_o W}{rB}\right)\right) = O\left(\frac{|I|}{B} + \frac{r p_o W}{B}\right)$$

We have thus proved the following theorem:

▶ **Theorem 17.** *Let $\mathcal{A}$ be a tuple-based MPC algorithm that computes query q over input I using r rounds with load $L(I,p)$. For internal memory size W, let $p_o = \min_p\{L(I,p) \leq W/r\}$. If $W \geq p_o$, then there exists an external memory algorithm $\mathcal{B}$ that computes q over the same input I with I/O cost:*

$$O\left(\frac{|I|}{B} + \frac{r p_o W}{B}\right).$$

We can simplify the above I/O cost further 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 $L(I,p) \geq |I|/p$. Then, we can rewrite the I/O cost as $O\left(p_o W/B\right)$.

We can apply Theorem 17 to any of the optimal multi-round algorithms we presented in the previous sections, and obtain state-of-the-art external memory algorithms for several classes of conjunctive queries. We show next an application for the case of query C_3.

▶ **Example 18.** We presented a 2-round algorithm that computes triangles for any input data with load (in tuples) $L = \tilde{O}(m/p^{3/2})$, in the case where all relations have size m. By applying Theorem 17, we obtain an external memory algorithm that computes triangles with $\tilde{O}(m^{3/2}/(BW^{1/2}))$ I/O cost for any $W \geq m^{2/5}$. Notice that this cost matches the I/O cost for triangle computation from [18] up to polylogarithmic factors.

6 Conclusion

In this work, we present the first worst-case analysis for parallel algorithms that compute conjunctive queries, using the MPC model as the theoretical framework for the analysis. We also show an interesting connection with the external memory computation model, which allows us to translate many of the techniques from the parallel setting to obtain algorithms for conjunctive queries with (almost) optimal I/O cost.

The central remaining open question is to design worst-case optimal algorithms for multiple rounds for any conjunctive query. We also plan to investigate further the connection between the parallel setting and external memory setting. It is an interesting question whether our techniques can lead to optimal external memory algorithms for any conjunctive query, and also whether we can achieve a reverse simulation of external memory algorithms in the MPC model.

Acknowledgements. We would like to thank Ke Yi for pointing out an error in the computation of the edge quasi-packing of the query L_k.

References

1 Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *CoRR*, abs/1206.4377, 2012.

2 Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010. doi:10.1145/1739041.1739056.

3 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748, 2008. doi:10.1109/FOCS.2008.43.

4 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *PODS*, pages 273–284, 2013. doi:10.1145/2463664.2465224.

5 Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In *PODS*, pages 212–223, 2014. doi:10.1145/2594538.2594558.

6 Shumo Chu and James Cheng. Triangle listing in massive networks. *TKDD*, 6(4):17, 2012. doi:10.1145/2382577.2382581.

7 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

8 Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms*, 6(4), 2010.

9 Gero Greiner and Riko Jacob. The efficiency of mapreduce in parallel external memory. In *Proceedings of the 10th Latin American International Conference on Theoretical Informatics*, LATIN'12, pages 433–445, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-29344-3_37.

10 Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the Myria big data management service. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 881–884. ACM, 2014. doi:10.1145/2588555.2594530.

11 Xiaocheng Hu, Miao Qiao, and Yufei Tao. Join dependency testing, Loomis-Whitney join, and triangle enumeration. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 – June 4, 2015*, pages 291–301, 2015. doi:10.1145/2745754.2745768.

12 Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. Massive graph triangulation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 325–336, 2013. doi:10.1145/2463676.2463704.

13 Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.

14 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'15, pages 391–410. SIAM, 2015.

15 Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011. doi:10.1145/1989284.1989310.

16 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.

17 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012. doi:10.1145/2213556.2213565.

18 Rasmus Pagh and Francesco Silvestri. The input/output complexity of triangle enumeration. In *PODS*, pages 224–233, 2014. doi:10.1145/2594538.2594552.

19 Francesco Silvestri. Subgraph enumeration in massive graphs. *CoRR*, abs/1402.3444, 2014. URL: http://arxiv.org/abs/1402.3444.

20 DavidP. Woodruff and Qin Zhang. When distributed computation is communication expensive. In Yehuda Afek, editor, *Distributed Computing*, volume 8205 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-41527-2_2.

21 M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.