Massively Parallel Join Algorithms

Xiao Hu
Duke University
xh102@cs.duke.edu

Ke Yi
HKUST
yike@cse.ust.hk

ABSTRACT

Due to the rapid development of massively parallel data processing systems such as MapReduce and Spark, there have been revived interests in designing algorithms in a massively parallel computational model. Computing multi-way joins, as one of the central algorithmic problems in databases, has received much attention recently. This paper surveys some of the recent algorithms, as well as lower bounds. We focus on multi-round algorithms, while referring readers to [27] for single-round algorithms.

1. INTRODUCTION

1.1 Join and Join-aggregate Queries

A (natural) *join* is represented as a hypergraph $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$, where the vertices $\mathcal{V} = \{x_1, \dots, x_n\}$ model the *attributes* and the hyperedges $\mathcal{E} = \{e_1, \dots, e_m\} \subseteq 2^{\mathcal{V}}$ model the *relations*. Let $\text{dom}(x)$ be the *domain* of attribute $x \in \mathcal{V}$. An *instance* of $\mathcal{Q}$ is a set of relations $\mathcal{R} = \{R(e) : e \in \mathcal{E}\}$, where $R(e)$ is a set of *tuples* on attributes e. We use $\text{IN} = \sum_{e \in \mathcal{E}} |R(e)|$ to denote the size of $\mathcal{R}$. The *join result* of $\mathcal{Q}$ on $\mathcal{R}$ is

$$\mathcal{Q}(\mathcal{R}) = \{t \mid \pi_e t \in R(e), \forall e \in \mathcal{E}\}.$$

Let $\text{OUT} = |\mathcal{Q}(\mathcal{R})|$ be the output size. We study the *data complexity* of algorithms, namely, the query size (i.e., $n + m$) is treated as a constant.

We consider join-aggregate queries over *annotated relations* [17, 23, 25] with one semiring. Let $(\mathbb{R}, \oplus, \otimes)$ be a commutative semiring. We assume that every tuple t is associated with an *annotation* $w(t) \in \mathbb{R}$. The annotation of a join result $t \in \mathcal{Q}(\mathcal{R})$ is

$$w(t) := \otimes_{t_e \in R(e), \pi_e t = t_e, e \in \mathcal{E}} w(t_e).$$

Let $\mathbf{y} \subseteq \mathcal{V}$ be a set of *output attributes* (a.k.a. *free variables*) and $\bar{\mathbf{y}} = \mathcal{V} - \mathbf{y}$ the non-output attributes (a.k.a. *bound variables*). A *join-aggregate query* $\mathcal{Q}_{\mathbf{y}}(\mathcal{R})$ asks us to compute $\oplus_{\bar{\mathbf{y}}} \mathcal{Q}(\mathcal{R}) =$

$$\left\{(t_{\mathbf{y}}, w(t_{\mathbf{y}})) : t_{\mathbf{y}} \in \pi_{\mathbf{y}} \mathcal{Q}(\mathcal{R}), w(t_{\mathbf{y}}) = \oplus_{t \in \mathcal{Q}(\mathcal{R}) : \pi_{\mathbf{y}} t = t_{\mathbf{y}}} w(t)\right\}.$$

In plain language, a join-aggregate query first computes the join $\mathcal{Q}(\mathcal{R})$ and the annotation of each join result, which is the $\otimes$-aggregate of the tuples comprising the join result. Then it partitions $\mathcal{Q}(\mathcal{R})$ into groups by their projection on $\mathbf{y}$. Finally, for each group, it computes the $\oplus$-aggregate of the annotations of the join results.

Many queries can be formulated as special join-aggregate queries. For example, if we take $\mathbb{R}$ to be the domain of integers, $\oplus$ to be addition, $\otimes$ to be multiplication, and set $w(t) = 1$ for all t, then it becomes the `COUNT(*) GROUP BY` $\mathbf{y}$ query; in particular, if $\mathbf{y} = \emptyset$, the query computes $|\mathcal{Q}(\mathcal{R})|$. The join-project query $\pi_{\mathbf{y}} \mathcal{Q}(\mathcal{R})$, also known as a *conjunctive query*, is a special join-aggregate query by discarding the annotations.

1.2 Model of Computation

We adopt the *Massively Parallel Computation* (MPC) model [28, 8, 9, 27]. In the MPC model, there are p servers connected by a complete communication network. Data are initially distributed across p servers with each server holding IN/p tuples. Computation proceeds in *rounds* or *super steps*. In each round, each server first receives messages from other servers (if there are any), performs some local computation, and then sends messages to other servers. The complexity of the algorithm is measured by the number of rounds, and the load, denoted as L, which is the maximum message size received by any server in any round. We will only consider constant-round algorithms. In this case, whether a server is allowed to keep messages it has received from previous rounds is irrelevant: if not, it can just keep sending all these messages to itself over the rounds, increasing the load by a constant factor.

The MPC model can be considered as a simplified version of the *bulk synchronous parallel* (BSP) model [32], but it has enjoyed more popularity in recent years. This is mostly because the BSP model takes too many measures into consideration, such as communication costs, local computation time, memory consumption, etc. The MPC model unifies all these costs with one parameter L, which makes the model much simpler. Meanwhile, although L is defined as the maximum incoming message size of a server, it is also closely related to the local computation time and memory consumption, which are both increasing functions of L. Thus, L serves as a good surrogate of these other cost measures. This is also why the MPC model does not limit the outgoing message size of a server, which is less relevant to other costs.

We will adopt the mild assumption $\text{IN} \geq p^{1+\epsilon}$ where $\epsilon > 0$ is any small constant. This assumption clearly holds on any reasonable values of IN and p in practice;

theoretically, this is the minimum requirement for the model to be able to compute some almost trivial functions, like the "or" of N bits, in $O(1)$ rounds [15]. When $\text{IN} \geq p^{1+\epsilon}$, many basic operations (see Section 2) can be performed in $O(1)$ rounds with $O(\text{IN}/p)$ load, which is often called "linear load", as it is the load needed to shuffle all input data once.

When proving lower bounds, we confine ourselves to *tuple-based* join algorithms, i.e., the tuples are atomic elements that must be processed and communicated in their entirety. The only way to create a tuple is by making a copy, from either the original tuple or one of its copies. We say that an MPC algorithm computes the join $\mathcal{Q}$ on instance $\mathcal{R}$ if the following is achieved: For any join result $t \in \mathcal{Q}(\mathcal{R})$, the tuples (or their copies) t_e such that $t_e \in R(e), \pi_e \in R(e)$ for all $e \in \mathcal{E}$ must be present on the same server at some point. Then the server will emit the join result. Recall that we allow a server to keep all messages it has received, so this requirement is equivalent to requiring that the tuples t_e all arrive at some server. For join-aggregate queries, we assume that the only way for a server to create new semiring elements is by multiplying and adding existing semiring elements currently residing on the same server.

1.3 Classification of Join Queries

Various classes of join queries have been studied in the literature. The relationships of join queries to be mentioned are illustrated in Figure 2.

Acyclic joins [10]. A join $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ is *acyclic* (a.k.a. α-acyclic) if there exists a tree $\mathcal{T}$ whose nodes are in one-to-one correspondence with the hyperedges in $\mathcal{E}$, such that for any vertex $v \in \mathcal{V}$, all nodes containing v are connected in $\mathcal{T}$. Such a tree $\mathcal{T}$ is called the *join tree* of $\mathcal{Q}$. Note that the join tree may not be unique for a given $\mathcal{Q}$.

Hierarchical joins [14]. A join $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ is *hierarchical* if for every pair of vertices x, y, there is $\mathcal{E}_x \subseteq \mathcal{E}_y$, or $\mathcal{E}_y \subseteq \mathcal{E}_x$, or $\mathcal{E}_x \cap \mathcal{E}_y = \emptyset$, where $\mathcal{E}_x = \{e \in \mathcal{E} : x \in e\}$ is the set of hyperedges containing attribute x. This is equivalent to the condition that all attributes can be organized into a forest, such that x is a descendant of y iff $\mathcal{E}_x \subseteq \mathcal{E}_y$.

r-hierarchical joins [20]. We consider a slightly larger class of hierarchical joins. A *reduce* procedure on a hypergraph $(\mathcal{V}, \mathcal{E})$ is to remove an edge $e \in \mathcal{E}$ if there exists another edge $e' \in \mathcal{E}$ such that $e \subseteq e'$. We can repeatedly apply the reduce procedure until no more edge can be reduced, and the resulting hypergraph is said to be *reduced*. A join is *r-hierarchical* if its reduced join hypergraph is hierarchical. A hierarchical join must be r-hierarchical, but not vice versa. For example, the join $R_1(A) \bowtie R_2(A, B) \bowtie R_3(B)$ is r-hierarchical but not hierarchical. On the other hand, an r-hierarchical join must be acyclic.

Note that the reduction procedure can be done in linear load using semijoins (see Section 2).

Tall-flat joins [28]. A join $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ is *tall-flat* if one can order the attributes as $x_1, x_2, \cdots, x_h, y_1, y_2, \cdots, y_l$ such that (1) $\mathcal{E}_{x_1} \supseteq \mathcal{E}_{x_2} \supseteq \cdots \supseteq \mathcal{E}_{x_h}$; (2) $\mathcal{E}_{x_h} \supseteq \mathcal{E}_{y_j}$ for $j = 1, 2, \cdots, l$; and (3) $|\mathcal{E}_{y_j}| = 1$ for $j = 1, 2, \cdots, l$. Obviously, a tall-flat join is hierarchical. For a tall-flat join, this attribute forest takes the form of a special tree, which consists of a single "stem" plus a number of leaves at the bottom. Consider two examples: $\mathcal{Q}_1 = R_1(x_1) \bowtie R_2(x_1, x_2) \bowtie R_3(x_1, x_2, x_3, x_4) \bowtie R_4(x_1, x_2, x_3, x_5) \bowtie R_5(x_1, x_2, x_3, x_6)$ is a tall-flat join, and $\mathcal{Q}_2 = R_1(x_1, x_2) \bowtie R_2(x_1, x_3, x_4) \bowtie R_3(x_1, x_3, x_5)$ is a hierarchical join (but not tall-flat). Their attribute forests (actually, trees for these cases) are shown in Figure 1.

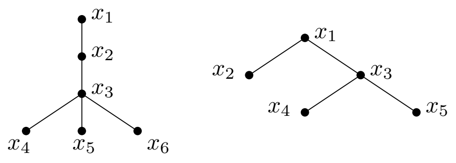

Figure 1: Tall-flat and hierarchical join.

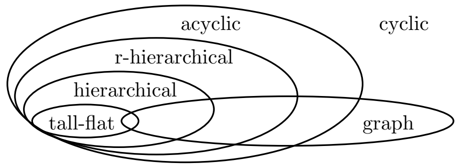

Figure 2: Relationships of joins.

Graph joins [24, 31]. A join is a *graph join* if every relation contains at most two attributes. If each relation contains exactly two attributes, the hypergraph becomes an ordinary graph. If a graph join is also acyclic, it is a *tree join*, i.e., the hypergraph is a tree.

Free-connex join-aggregate queries [6]. With respect to join-aggregate queries, *free-connex* queries are an important subclass. To define a free-connex query, we introduce the notion of a *width-1 GHD*, which can be considered as a generalized join tree. A *width-1 GHD* of a hypergraph $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ is a tree $\mathcal{T}$, where each node $u \in \mathcal{T}$ is a subset of $\mathcal{V}$, such that (1) for each attribute $x \in \mathcal{V}$, the nodes containing x are connected in $\mathcal{T}$; (2) for each hyperedge $e \in \mathcal{E}$, there exists a node $u \in \mathcal{T}$ such that $e \subseteq u$; and (3) for each node $u \in \mathcal{T}$, there exists a hyperedge $e \in \mathcal{E}$ such that $u \subseteq e$.

Given a set of output attributes $\mathbf{y}$, $\mathcal{T}$ is free-connex if there is a subset of connected nodes of $\mathcal{T}$, denoted as $\mathcal{T}'$ (such a $\mathcal{T}'$ is said to be a connex subset), such that $\mathbf{y} = \bigcup_{u \in \mathcal{T}'} u$. A join-aggregate query $\mathcal{Q}_{\mathbf{y}}$ is free-connex if it has a free-connex width-1 GHD. For example, the join-project query $\pi_A R_1(A, B) \bowtie R_2(B, C)$ is free-connex while $\sum_B R_1(A, B) \bowtie R_2(B, C)$ is not.

1.4 Complexity Measures

In worst-case analysis, the entire space of instances is divided into classes by the input size IN, and the running time is measured on the worst instance in each class. Let $\mathfrak{R}(\text{IN})$ be the class of instances with input size IN. The load of an MPC Algorithm $\mathbb{A}$ is a function of IN, defined as

$$L_{\mathcal{A}}(\text{IN}) = \max_{\mathcal{R}\in\mathfrak{R}(\text{IN})} L_{\mathcal{A}}(\mathcal{R}),$$

where Algorithm $\mathcal{A}$ is *worst-case optimal* if

$$L_{\mathcal{A}}(\text{IN}) = O(L_{\mathcal{A}'}(\text{IN})),$$

for every algorithm $\mathcal{A}'$.

A more refined approach is *parameterized analysis*, which further subdivides the instance space into smaller classes by introducing more parameters that can better characterize the difficulty of each class. For the join problem, the output size OUT is a commonly used parameter, and each class of instances share the same input and output size. Let $\mathfrak{R}(\text{IN}, \text{OUT})$ be the class of instances with input size IN and output size OUT. Then the load of an MPC algorithm $\mathcal{A}$ is thus a function of both IN and OUT, defined as

$$L_{\mathcal{A}}(\text{IN}, \text{OUT}) = \max_{\mathcal{R}\in\mathfrak{R}(\text{IN},\text{OUT})} L_{\mathcal{A}}(\mathcal{R}),$$

Algorithm $\mathcal{A}$ is *output-optimal* if

$$L_{\mathcal{A}}(\text{IN}, \text{OUT}) = O(L_{\mathcal{A}'}(\text{IN}, \text{OUT})),$$

for every algorithm $\mathcal{A}'$.

Further subdividing the instance space leads to more refined analyses. In the extreme case when each class contains just one instance, we obtain instance-optimal algorithms. Algorithm $\mathcal{A}$ is *instance-optimal* if

$$L_{\mathcal{A}}(\mathcal{R}) = O(L_{\mathcal{A}'}(\mathcal{R})),$$

for every instance $\mathcal{R}$ and every algorithm $\mathcal{A}'$.

By definition, an instance-optimal algorithm must be output-optimal, and an output-optimal algorithm must be worst-case optimal, but the reverse direction may not be true.

In the RAM model, the Yannakakis algorithm [34] can compute any acyclic join in time $O(\text{IN} + \text{OUT})$, which is both output-optimal and instance-optimal, because on any instance $\mathcal{R}$, any algorithm has to spend at least $\Omega(\text{IN})$ time to read inputs and $\Omega(\text{OUT})$ time to enumerate the outputs. Thus, the two notions of optimality coincide (but both are stronger than worst-case optimality). Fundamentally, this is because the difficulty of any instance $\mathcal{R}$ is precisely characterized by its input size and output size, and all instances in $\mathfrak{R}(\text{IN}, \text{OUT})$ have exactly the same complexity $O(\text{IN} + \text{OUT})$.

1.5 Overview of Results

Earlier efforts have been devoted to one-round MPC algorithms; please see [27] for an excellent survey. However, for many queries, there can be a polynomial difference between one-round and multi-round algorithms. For example, the optimal load for the triangle join $\mathcal{Q}_\triangle = R_1(B,C), R_2(A,C), R_3(A,B)$ is $O(\frac{\text{IN}}{p^{1/2}})$, while a two-round algorithm can achieve $O(\frac{\text{IN}}{p^{2/3}})$ [26].

In this paper, we focus on multi-round (but still a constant) algorithms in the MPC model. We give a brief overview of results below, while describing some selective algorithms in more detail in later sections. Tables 1 and 2 provide a summary of the results.

Instance-optimal join algorithms. We start from computing the Cartesian product of m sets of sizes $N_1, \ldots, N_m$. Since the output size is $\prod_{i=1}^m N_i$ and each server can emit at most L^m results if the load is L, an immediate lower bound on L is $(\frac{\prod_{i=1}^m N_i}{p})^{\frac{1}{m}}$. In addition, any algorithm computing the full Cartesian product must also implicitly compute the Cartesian product of any subset of the m sets. Applying the same argument for each subset S, we obtain a lower bound of

$$L_{\text{Cartesian}}(p, \mathcal{R}) := \max_{S\subseteq\{1,\ldots,m\}} \left(\frac{\prod_{i\in S} N_i}{p}\right)^{\frac{1}{|S|}}. \quad (1)$$

It has been shown that the HyperCube algorithm [2] incurs a load of $L_{\text{Cartesian}}(p, \mathcal{R}) \cdot \log^{O(1)} p$ on any instance $\mathcal{R}$ [8]. Thus, it can be considered as an instance-optimal algorithm for computing Cartesian products, with an optimality ratio of $\log^{O(1)} p$.

We can extend the Cartesian product lower bound to a join query $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$. For any subset of relations $S \subseteq \mathcal{E}$, define

$$\mathcal{Q}(\mathcal{R}, S) := (\Join_{e\in S} R(e)) \ltimes \mathcal{Q}(\mathcal{R}),$$

i.e., the join results of relations in S that are part of a full join result. Clearly, any algorithm computing $\mathcal{Q}(\mathcal{R})$ must implicitly also compute $\mathcal{Q}(\mathcal{R}, S)$ for every S. Because each join result in $\mathcal{Q}(\mathcal{R}, S)$ consists of $|S|$ tuples, one from each relation in S, a server can emit at most $O(L^{|S|})$ join results of $\mathcal{Q}(\mathcal{R}, S)$, so we must have $p \cdot L^{|S|} = \Omega(|\mathcal{Q}(\mathcal{R}, S)|)$. Thus, we obtain the following per-instance lower bound on the load:

$$L_{\text{instance}}(p, \mathcal{R}) := \max_{S\subseteq\mathcal{E}} \left(\frac{|\mathcal{Q}(\mathcal{R}, S)|}{p}\right)^{\frac{1}{|S|}}. \quad (2)$$

It has been shown that r-hierarchical queries are precisely the class of queries that admit instance-optimal algorithms [8, 20]. More precisely, there is an algorithm with load $O(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R}))$ for computing any r-hierarchical query, while for every acyclic join that is not r-hierarchical, there is an instance $\mathcal{R}$ with $L_{\text{instance}}(p, \mathcal{R}) = O(\frac{\text{IN}}{p})$ but any multi-round algorithm must incur a load of $\tilde{\Omega}(\frac{\text{IN}}{p^{1/2}})$ on $\mathcal{R}$. Section 3 gives more details on these results.

Output-sensitive algorithms. By plugging the two-way join algorithm [8, 22] into the classical Yannakakis algorithm [34], one can compute any acyclic join with load $O(\frac{\text{IN}}{p} + \frac{\text{OUT}}{p})$ [1], but this is not output-optimal. As mentioned, an instance-optimal algorithm must also be output-optimal, so we have automatically obtained output-optimal algorithms for r-hierarchical joins. In fact, it has been shown that $L_{\text{instance}}(p, \mathcal{R}) = O(\frac{\text{IN}}{p} +$

[2]For a join query $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$, the edge quasi-packing number is defined as follows. Let $x \subseteq \mathcal{V}$ be any subset of vertices of $\mathcal{V}$. Define the residual hypergraph derived by removing attributes in x as $\mathcal{Q}_x = (\mathcal{V}_x, \mathcal{E}_x)$, where $\mathcal{V}_x = \mathcal{V} - x$ and $\mathcal{E}_x = \{e - x : e \in E\}$. The edge quasi-packing number of $\mathcal{Q}$ is the maximum optimal fractional edge packing number over all $\mathcal{Q}_x$'s, i.e., $\psi^* = \max_{x\subseteq\mathcal{V}} \tau^*(\mathcal{Q}_x)$.

Joins	Instance-optimal		Output-optimal		Worst-case Optimal	
	one-round	multi-round	one-round	multi-round	one-round	multi-round
tall-flat	$\widetilde{O}\left(L^*\right)$ [8]	$\Theta\left(L^*\right)$ [20]	$O\left(\frac{\text{IN}}{p^{1/\max\{1,k^*-1\}}}+\left(\frac{\text{OUT}}{p}\right)^{\frac{1}{k^*}}\right)$ [20]		$\widetilde{O}\left(\frac{\text{IN}}{p^{1/\psi^*}}\right)$ [26]	$\widetilde{O}\left(\frac{\text{IN}}{p^{1/\rho^*}}\right)$ [18]
r-hierarchical w/o dangling tuples						
r-hierarchical w/ dangling tuples	$\omega\left(L^*\right)$ [20]		$\omega\left(\frac{\text{IN}}{p}+\frac{\text{OUT}}{p}\right)$ [28]			
acyclic				$\Theta\left(\frac{\text{IN}}{p}+\frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p}\right)$ [20] LB for $\text{OUT} \leq p \cdot \text{IN}$		
cyclic				$\tilde{\Omega}\left(\min\{\frac{\text{IN}+\text{OUT}}{p}, \frac{\text{IN}}{p^{2/3}}\}\right)$ for triangle join [20]		$\widetilde{O}\left(\frac{\text{IN}}{p^{1/\rho^*}}\right)$ for LW join [26], graph join [24, 31]
						$\Omega\left(\frac{\text{IN}}{p^{1/\tau^*}}\right)$ for $\boxminus$-join [18]

Table 1: Join algorithms in the MPC model. IN is the input size, OUT is the output size and p is the number of servers in the MPC model. For instance-optimal algorithms, $L^* = \frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R})$. For output-optimal algorithm on r-hierarchical joins, $k^* = \lceil \log_{\text{IN}} \text{OUT} \rceil$. ψ^* is the optimal fractional edge quasi-packing number[2]; ρ^* is the optimal fractional edge cover number; τ^* is the optimal fractional edge packing number.

$\sqrt{\frac{\text{OUT}}{p}})$ for all r-hierarchical joins [20], improving the Yannakakis algorithm by a quadratic factor.

Unfortunately, such a quadratic improvement is not possible beyond r-hierarchical joins, even for the line-3 join $R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D)$ [22], which is the simplest non-r-hierarchical join. Nevertheless, one can achieve a load of $O(\frac{\text{IN}}{p} + \frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p})$ [20] for all acyclic joins (see Section 4.1 for more details), which improves upon Yannakakis algorithm as long as $\text{OUT} > \text{IN}$. This bound has also been shown to be optimal for $\text{OUT} = O(p \cdot \text{IN})$. Note that some restriction on OUT is inherent, because the $O(\frac{\text{IN}}{p} + \frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p})$ bound cannot be optimal for all values of OUT. When OUT is large enough, a worst-case optimal algorithm will take over, as will be seen next.

What kind of output-sensitive bounds are achievable for cyclic joins remains an open problem, even for the triangle join. While there is an output-sensitive algorithm for the triangle join in the RAM model [11], we currently don't have any non-trivial upper bounds in the MPC model. On the lower bound side, a lower bound of $\tilde{\Omega}(\min\{\frac{\text{IN}+\text{OUT}}{p}, \frac{\text{IN}}{p^{2/3}}\})$ has been shown for cyclic joins in the MPC model [20]; please see Section 7.1 for more details. This shows a separation from acyclic joins, i.e., cyclic joins are harder than acyclic ones by at least a factor of $\tilde{\Omega}(\sqrt{\frac{\text{OUT}}{\text{IN}}})$.

Worst-case Optimal Join Algorithms. For worst-case optimal algorithms, we consider the simpler case where all relations $R(e)$ have equal size. In the RAM model, there is a unified algorithm computing all joins in $O(\text{IN}^{\rho^*})$ time [30, 33, 8], where ρ^* is the optimal fractional edge cover of the query hypergraph, i.e., it is the optimal solution of the following linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{e\in\mathcal{E}} x_e \\ \text{subject to} \quad & \sum_{e:v\in e, e\in\mathcal{E}} x_e \geq 1, \forall v \in \mathcal{V} \\ & x_e \geq 0, \forall e \in \mathcal{E}. \end{aligned}$$

The conjecture had been an MPC algorithm with load $O(\frac{\text{IN}}{p^{1/\rho^*}})$. Such a load can be easily shown to be optimal: Each server can only produce $O(L^{\rho^*})$ join results in a constant number of rounds when the load is limited to L (implied by the AGM bound [4]), so all the p servers can produce at most $O(p \cdot L^{\rho^*})$ join results. On the other hand, the output size can be as large as IN^{ρ^*} on certain instances, so the load L has to be $\Omega(\frac{\text{IN}}{p^{1/\rho^*}})$ in the worst case. Indeed, this load has been achieved on certain classes of joins, such as graph joins [24, 31], Loomis-Whitney (LW) joins [26], and acyclic joins [18]; we describe some of these algorithms in Section 4.2 and Section 6.

Until very recently, an $\Omega(\frac{\text{IN}}{p^{1/\tau^*}})$ lower bound has been proved for the $\boxminus$-join $\mathcal{Q}_{\boxminus} = R_1(A,B,C) \bowtie R_2(D,E,F) \bowtie R_3(A,D) \bowtie R_4(B,E) \bowtie R_5(C,F)$ [18], where τ^* is the optimal fractional edge packing of the query, i.e., the optimal solution of the following linear program:

$$\begin{aligned} \text{maximize} \quad & \sum_{e\in\mathcal{E}} x_e \\ \text{subject to} \quad & \sum_{e:v\in e, e\in\mathcal{E}} x_e \leq 1, \forall v \in \mathcal{V} \\ & x_e \geq 0, \forall e \in \mathcal{E}. \end{aligned}$$

On $\mathcal{Q}_{\boxminus}$, we have $\rho^* = 2$ and $\tau^* = 3$, so this result rules out the possibility of achieving $O\left(\frac{\text{IN}}{p^{1/\rho^*}}\right)$ for all joins. Please see Section 7.2 for more details.

Join-Aggregate Algorithms. For a join-aggregate query, computing the full join and then performing the aggregation can be far from optimal, since the full join size can be much larger than the final output size OUT. In the RAM model, the Yannakakis algorithm can be modified to push down the aggregation as early as possible. Specifically, after removing the dangling tuples, we perform a bottom-up traversal of the join tree. For each $R(e)$ and $R(e')$ such that e is a leaf and e' is the parent of e, and we replace $R(e')$ with $\pi_{\mathbf{y}\cup \text{anc}(e')} R(e) \bowtie R(e')$, where $\text{anc}(e')$ is the set of attributes in e' that appear in the ancestors of e'. Then $R(e)$ is removed and the step is repeated until only one relation remains. It has been noted that this algorithm can be easily modified to handle join-aggregate queries, by replacing the projection $\pi_{\mathbf{y}\cup \text{anc}(e')}$ by an aggregation [23].

The running time of this algorithm is proportional to the largest intermediate join size $|R(e) \bowtie R(e')|$. It is known that if the query is *free-connex*, then the maximum intermediate join size is $O(\text{OUT})$ [23, 7]. For non-free-connex queries, Yannakakis gave an upper bound of $O(\text{IN} \cdot \text{OUT})$ in his original paper [34]. For matrix multiplication $\sum_B R_1(A, B) \bowtie R_2(B, C)$, which is the simplest non-free-connex query, this has been tightened to $O(\text{IN}\sqrt{\text{OUT}})$ [3], which is also shown to be optimal in the semiring model, as there are instances with $\Omega(\text{IN}\sqrt{\text{OUT}})$ elementary products. This bound is also extended to star queries[3], for which the bound becomes $O(\text{IN} \cdot \text{OUT}^{1-1/n})$.

Plugging the optimal two-way join algorithm to the Yannakakis algorithm, together with the MPC primitives for semi-join and aggregation (Section 2), we are able to compute join-project or join-aggregate queries in the MPC model. This is referred to as *distributed Yannakakis algorithm* in [23, 1]. The load of this algorithm is $O(\frac{\text{IN}}{p} + \frac{J}{p})$, where J is the maximum intermediate join size. Combined with the previously known bounds on J [34, 23, 7, 3], this implies that it can compute free-connex queries with load $O(\frac{\text{IN}}{p} + \frac{\text{OUT}}{p})$, matrix multiplication with load $O(\frac{\text{IN}}{p} + \frac{\text{IN}\sqrt{\text{OUT}}}{p})$, star queries with load $O(\frac{\text{IN}}{p} + \frac{\text{IN}\cdot\text{OUT}^{1-1/n}}{p})$, and general acyclic join-aggregate queries with load $O(\frac{\text{IN}}{p} + \frac{\text{IN}\cdot\text{OUT}}{p})$.

For any free-connex query, we can reduce it to a full acyclic join and then invoke the output-sensitive algorithm. This improves the load to $O(\frac{\text{IN}}{p} + \frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p})$ (see Section 4.3). However, for non-free-connex queries, such a reduction is not possible. In fact, matrix multiplication, which is the simplest non-free-connex query, already requires a different treatment. Recently, it has been shown that matrix multiplication can be solved in $O(1)$ rounds with load $\widetilde{O}(\min\{\frac{\text{IN}}{\sqrt{p}}, \frac{\text{IN}}{p} + \frac{\text{IN}^{2/3}\cdot\text{OUT}^{1/3}}{p^{2/3}}\})$ which is also optimal [21]; Section 5 describes this algorithm in more detail. Improved bounds have also been obtained for other non-free-connex queries, as summarized in Table 2.

[3]A star query is defined as $\sum_B R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \cdots \bowtie R_m(A_m, B)$.

2. MPC PRIMITIVES

Assume $\text{IN} > p^{1+\epsilon}$ where $\epsilon > 0$ is any small constant. We first introduce the following primitives in the MPC model, all of which can be computed with linear load $O(\frac{\text{IN}}{p})$ in $O(1)$ rounds.

Sorting [16]: Given IN elements, redistribute them so that each server has $O(\frac{\text{IN}}{p})$ elements in the end, while any element on server i is smaller than or equal to any element on server j, for any $i < j$.

Reduce-by-key [22]: Given IN (key, value) pairs, compute the "sum" of values for each key, where the "sum" is defined by any associative operator. An aggregate $\oplus_{\mathbf{y}} \mathcal{R}$ can be computed as a reduce-by-key operation.

This primitive will also be frequently used to compute data statistics, for example the degree information. The degree of value $a \in \text{dom}(x)$ in relation $R(e)$ is defined as the number of tuples in $R(e)$ having this value in attribute x, i.e., $|\sigma_{x=a} R(e)|$. Each tuple $t \in R(e)$ is considered to have "key" $\pi_x t$ and "value" 1.

Multi-search [22]: Given N_1 elements $x_1, x_2, \cdots, x_{N_1}$ as set X and N_2 elements $y_1, y_2, \cdots, y_{N_2}$ as set Y, where all elements are drawn from an ordered domain. Set $\text{IN} = N_1 + N_2$. For each x_i, find its predecessor in Y, i.e., the largest element in Y but smaller than x_i.

Semi-Join: Given two relations R_1 and R_2 with a common attribute x, the semijoin $R_1 \ltimes R_2$ returns all the tuples in R_1 whose value on x matches that of at least one tuple in R_2. This can be reduced to a multi-search problem: For each $t \in R_1$, if its predecessor on the x attribute in R_2 is the same as that of t, then it is in the semi-join.

Note that we can remove all *dangling tuples*, i.e., those do not appear in the join results, of an acyclic-join [34] by a constant number of semi-joins, so it can be done in $O(1)$ rounds with linear load. Moreover, semi-joins are also used to *reduce* a join and join-aggregate query. If there exists a pair of relations $R(e), R(e')$ such that $e \subset e'$, then we can replace $R(e')$ with $R(e) \bowtie R(e')$ and then discard $R(e)$. Note that by the earlier definition, the annotation of a join result is the $\otimes$-aggregate of the annotations of tuples comprising the join result, so the annotations in $R(e)$ are aggregated into those in $R(e')$ correctly. As mentioned, a join query can be reduced by applying a set of semi-joins until no relation is a subset of another relation.

Parallel-packing [22]: Given IN numbers $x_1, x_2, \cdots, x_{\text{IN}}$ where $0 < x_i \leq 1$ for $i = 1, 2, \cdots, \text{IN}$, group them into m sets $Y_1, Y_2, \cdots, Y_m$ such that $\sum_{i \in Y_j} x_i \leq 1$ for all j, and $\sum_{i \in Y_j} x_i \geq \frac{1}{2}$ for all but one j. Initially, the IN numbers are distributed arbitrarily across all servers, and the algorithm should produce all pairs (i, j) if $i \in Y_j$ when done. Note that $m \leq 1 + 2\sum_i x_i$.

Join Aggregate Query	The Yannakakis algorithm	New results [21]
Matrix Multiplication	$O\left(\frac{\mathrm{IN}}{p} + \frac{\mathrm{IN}\cdot\sqrt{\mathrm{OUT}}}{p}\right)$ [23, 3]	$\widetilde{O}\left(\frac{N_1+N_2}{p} + \min\left\{\sqrt{\frac{N_1 N_2}{p}}, \frac{N_1^{1/3}\cdot N_2^{1/3}\cdot\mathrm{OUT}^{1/3}}{p^{2/3}}\right\}\right)$ (1) optimal for $N_1, N_2 \geq 2$ and $\max\{N_1, N_2\} \leq \mathrm{OUT} \leq N_1 N_2$; (2) $\widetilde{O}\left(\frac{\mathrm{IN}}{p} + \min\left\{\frac{\mathrm{IN}}{\sqrt{p}}, \frac{\mathrm{IN}^{2/3}\cdot\mathrm{OUT}^{1/3}}{p^{2/3}}\right\}\right)$ if $N_1 = N_2$;
Star	$O\left(\frac{\mathrm{IN}}{p} + \frac{\mathrm{IN}\cdot\mathrm{OUT}^{1-\frac{1}{n}}}{p}\right)$ [23, 3]	$\widetilde{O}\left((\frac{\mathrm{IN}\cdot\mathrm{OUT}}{p})^{2/3} + \frac{\mathrm{IN}\cdot\mathrm{OUT}^{1/2}}{p} + \frac{\mathrm{IN}+\mathrm{OUT}}{p}\right)$
Line	$O\left(\frac{\mathrm{IN}}{p} + \frac{\mathrm{IN}\cdot\mathrm{OUT}}{p}\right)$ [23]	
Tree		$\widetilde{O}\left(\frac{\mathrm{IN}\cdot\mathrm{OUT}^{2/3}}{p} + \frac{\mathrm{IN}+\mathrm{OUT}}{p}\right)$

Table 2: Summary of Join-Aggregate Queries in the MPC model. In the sparse matrix multiplication, N_1, N_2 are the number of non-zero entries in two input matrices respectively. Generally, any instance for the join-aggregate query has input size IN and output size OUT. p is the number of servers.

Output size computation [20, 23, 21]: For any acyclic join $\mathcal{Q}$, the value of OUT can be computed exactly as a special case of our join-aggregate algorithm, which will be described in Section 4.3. This result also applies for free-connex join-aggregate queries after applying a primitive transformation.

However, for cyclic full join and non-free-connex join-aggregate queries, how to compute OUT effectively is still open. Fortunately, a constant-factor approximation of OUT for line join-aggregate queries (including matrix multiplication as a special case) can be computed in $O(1)$ rounds with linear load. A similar idea has been used by [13] in the RAM model.

3. R-HIERARCHICAL JOINS

It is known that r-hierarchical joins are precisely the class of joins that admit instance-optimal algorithms. In this section, we first present an instance-optimal algorithm, and then the lower bound. There are two such algorithms. The BinHC algorithm [8] is randomized and has some extra polylogarithmic factors, while the one in [20] is deterministic without any logarithmic factors.

Note that the BinHC algorithm is a generalization of the HyperCube algorithm to general joins. The load of the BinHC algorithm is parameterized by the degrees of all subsets of attribute values. Beame et al. [8] show that BinHC is optimal (up to polylog factors) within the class of instances sharing the same degrees, among all one-round MPC algorithms. A stronger analysis of the BinHC algorithm shows that it is actually instance-optimal (up to polylog factors) for (1) all tall-flat joins, and (2) all r-hierarchical joins provided that the instance does not contain dangling tuples. Furthermore, since the per-instance lower bound (2) also holds for multi-round algorithms, these instance-optimality results extend to multi-round algorithms as well. For r-hierarchical joins with dangling tuples, one-round algorithms cannot achieve $O(\frac{\mathrm{IN}}{p}+L_{\mathrm{instance}}(p,\mathcal{R}))$ load. But, removing dangling tuples (an MPC primitive) first and then running BinHC algorithm, leads to a multi-round algorithm of $O((\frac{\mathrm{IN}}{p} + L_{\mathrm{instance}}(p,\mathcal{R}))\cdot\log^{O(1)} p)$ load, where the $O(1)$ exponent depends on the query size, and is at least m, the number of relations.

Below we describe the latter deterministic algorithm with load $O(\frac{\mathrm{IN}}{p} + L_{\mathrm{instance}}(p,\mathcal{R}))$, i.e., improving the instance-optimality ratio from $\log^{O(1)} p$ to $O(1)$.

3.1 An Instance-Optimal Algorithm

We give a sketch of the recursive algorithm in [20]. It chooses a fixed threshold L, whose value will be determined later.

Inductive hypothesis: For an r-hierarchical join $\mathcal{Q}$ with an instance $\mathcal{R}$, the join results $\mathcal{Q}(\mathcal{R})$ can be computed using $\Psi(\mathcal{Q},\mathcal{R},L)$ servers in $O(1)$ rounds with $O(L)$ load, where

$$\Psi(\mathcal{Q},\mathcal{R},L) = \max_{S\subseteq\mathcal{E}} \left\lceil \frac{\Join_{e\in S} R(e)}{L^{|S|}} \right\rceil.$$

Base case: When $\mathcal{Q}$ has just one relation, say $\mathcal{E} = \{e\}$, in which case the algorithm simply emits all tuples in the relation. This step can be done using $O(\frac{|R(e)|}{L})$ servers with $O(L)$ load. Note that $\frac{|R(e)|}{L} \leq \Psi(\mathcal{Q},\mathcal{R},L)$.

General case: In general, we first reduce the query and remove all the dangling tuples, which can be done by MPC primitives. Then we are left with a hierarchical join $\mathcal{Q}$ on an instance $\mathcal{R}$ with no dangling tuples. Note that $\mathcal{Q}$ has an attribute forest, denoted as $\mathcal{T}$, where each relation corresponds to a root-to-leaf path of $\mathcal{T}$. The algorithm will proceed by the following two cases.

Case (1): $\mathcal{Q}$ is connected. Suppose the root attribute of $\mathcal{T}$ is x. Since x is included in all relations, we can decompose the original join into disjoint subsets by the value on x. Each $a \in \mathrm{dom}(x)$ induces a sub-instance $\mathcal{R}_a = \{\sigma_{x=a}R(e) : e \in \mathcal{E}\}$. A sub-instance is *heavy* if it contains more than L tuples, and *light* otherwise.

All light sub-instances are packed into groups (an MPC primitive) and send a group as whole to one server for computation. Then, it remains to compute $\mathcal{Q}_x$ on each heavy $\mathcal{R}_a$, where $\mathcal{Q}_x$ is the residual query by removing x from all relations in $\mathcal{Q}$. The challenge is to allocate p servers in total to these residual queries appropriately so as to compute all $\mathcal{Q}_x(\mathcal{R}_a)$'s in parallel while ensuring a uniform load of $O(L)$. To do so, we allocate for instance $\mathcal{R}_a$

$$p_a = \max_{S \subseteq \mathcal{E}} \left\lceil \frac{\mathcal{Q}_x(\mathcal{R}_a, S)}{L^{|S|}} \right\rceil$$

servers and compute $\mathcal{Q}_x(\mathcal{R}_a)$'s recursively in parallel. By hypothesis, for each heavy sub-instance $\mathcal{R}_a$, $\mathcal{Q}_x(\mathcal{R}_a)$ can be computed using p_a servers with $O(L)$ load.

Case (2): $\mathcal{Q}$ is disconnected. Let $\mathcal{Q}_1, \mathcal{Q}_2, \cdots, \mathcal{Q}_k$ be the connected components of $\mathcal{Q}$. In this case, the join becomes a Cartesian product $\mathcal{Q}_1(\mathcal{R}_1) \times \cdots \times \mathcal{Q}_k(\mathcal{R}_k)$, where each $\mathcal{Q}_i(\mathcal{R}_i)$ is a join under the Case (1).

The idea is to arrange servers into a $p_1 \times p_2 \times \cdots \times p_k$ hypercube, where each server is identified with coordinates $(c_1, c_2, \cdots, c_k)$, for $c_i \in [p_i]$. For every combination $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_k$, the p_i servers with coordinates $(c_1, \cdots, c_{i-1}, *, c_{i+1}, \cdots, c_k)$ form a group to compute $\mathcal{Q}_i(\mathcal{R}_i)$ (using the algorithm under Case (1)). Yes, each $\mathcal{Q}_i(\mathcal{R}_i)$ is computed $p_1 \cdots p_{i-1} p_{i+1} \cdots p_k$ times, which seems to be a lot of redundancy. However, as we shall see, there will be no redundancy in terms of the final join results, and it is exactly due to this redundancy that we avoid the shuffling of the intermediate result and achieve an optimal load. Consider a particular server $(c_1, \ldots, c_k)$. It participates in k groups, one for each $\mathcal{Q}_i(\mathcal{R}_i), i = 1, \ldots, k$. For each $\mathcal{Q}_i(\mathcal{R}_i)$, it emits a subset of its join results, denoted $\mathcal{Q}_i(\mathcal{R}_i, c_1 \ldots, c_k)$. Then the server emits the Cartesian product $\mathcal{Q}_1(\mathcal{R}_1, c_1 \ldots, c_k) \times \cdots \times \mathcal{Q}_k(\mathcal{R}_k, c_1 \ldots, c_k)$. Note that for each group of servers computing $\mathcal{Q}_i(\mathcal{R}_i)$, the p_i servers in the group emit $\mathcal{Q}_i(R_i)$ with no redundancy, so there is no redundancy in emitting the Cartesian product.

It remains to show how to allocate the p servers for each sub-query so that $p_1 \cdots p_k = O(p)$ If $|\mathcal{R}_i| < L$ is light, we set $p_i = 1$; otherwise set

$$p_i = \max_{S \subseteq \mathcal{E}_i} \left\lceil \frac{|\mathcal{Q}_i(\mathcal{R}_i, S)|}{L^{|S|}} \right\rceil.$$

By hypothesis, $\mathcal{Q}_i(\mathcal{R}_i)$ can be computed using p_i servers with $O(L)$ load. Although each server participates in k sub-queries, it still has a load of $O(L)$.

Combining two cases completes the inductive proof.

Choosing L. At last, we show how to choose an appropriate value of L. For an input join $\mathcal{Q}$ and an instance $\mathcal{R}$, we first compute the value of $L_{\text{instance}}(p, \mathcal{R})$ by a constant number of MPC primitives. Setting $L = L_{\text{instance}}(p, \mathcal{R}) + \frac{\text{IN}}{p}$ will yield $\Psi(\mathcal{Q}, \mathcal{R}, L) = O(p)$, thus leading to an instance-optimal algorithm.

3.2 Lower Bound

The instance-optimal load $O(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R}))$ is not achievable beyond r-hierarchical joins. More precisely, the following lower bound is proved in [20]:

THEOREM 1. *For any $\text{IN} \geq p^{3/2}$, there exists an instance $\mathcal{R}$ with input size $\Theta(\text{IN})$ for any acyclic but non-r-hierarchical join, such that any tuple-based algorithm that computes the join in $O(1)$ rounds must have a load of $\Omega\left(\frac{\text{IN}}{p^{1/2} \log \text{IN}}\right)$, while $L_{instance}(p, \mathcal{R}) = O(\frac{\text{IN}}{p})$.*

4. ACYCLIC JOIN

Theorem 1 has ruled out the possibility of achieving an instance-optimal algorithm for non-r-hierarchical joins. In this section, we review two algorithms for general acyclic joins, one is worst-case optimal algorithm [18] and the other is output-sensitive [20] (but optimal on specific range of OUT). We illustrate the high-level idea of these two algorithms through the line-3 join $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$, which is the simplest acyclic but non-r-hierarchical join. Finally, we turn to free-connex join-aggregate query. After applying a linear transformation procedure, any free-connex join-aggregate query can be reduced to a full join query, thus benefits from any results achieved for acyclic joins.

4.1 Output-sensitive algorithm

Note that in the RAM model, the Yannakakis algorithm first removes all the dangling tuples and then performs pairwise joins in some arbitrary order. It is shown that the join order does not affect the asymptotic running time: After dangling tuples have been removed, any intermediate join result is part of a full join result, so the running time of the last join, which is $\Theta(\text{OUT})$, dominates that of any intermediate join. Interestingly, the join order does matter in the MPC model. Moreover, it is shown that a global best order may not exist even for the line-3 join. The basic idea of the output-sensitive algorithm is to decompose the join into multiple pieces, and find a provably good join order of the Yannakakis algorithm for each piece.

Line-3 join. The value of OUT should be computed in advance (an MPC primitive). Set $\tau = \sqrt{\frac{\text{OUT}}{\text{IN}}}$. We first compute degrees for values of attribute B in relation R_1. A value $b \in \text{dom}(B)$ is *heavy* if it has degree greater than τ in R_1, and *light* otherwise. Let B^H, B^L be the set of heavy and light values in B respectively. In this way, we decompose the join into the following two parts as $\mathcal{Q}_1, \mathcal{Q}_2$ and compute them with aggregated Yannakakis algorithm using different join orderings:

$$\mathcal{Q}_1 = R_1(A, B^H) \bowtie \left(R_2(B^H, C) \bowtie R_3(C, D)\right)$$
$$\mathcal{Q}_2 = \left(R_1(A, B^L) \bowtie R_2(B^L, C)\right) \bowtie R_3(C, D)$$

The observation is that the intermediate join $R_2(B^H, C) \bowtie R_3(C, D)$ has its size bounded by $\frac{\text{OUT}}{\tau}$, since each intermediate join result has a heavy B value, so it joins with at least τ tuples in R_1. Meanwhile, the intermediate join $R_1(A, B^L) \bowtie R_2(B^L, C)$ has its size bounded by $\text{IN} \cdot \tau$, since each tuple from R_2 can join with at most τ tuples from R_1. The load of computing two sub-queries is $O(\frac{\text{IN}}{p} + \frac{\text{IN} \cdot \tau}{p} + \frac{\text{OUT}}{p \cdot \tau} + \sqrt{\frac{\text{OUT}}{p}}) = O(\frac{\text{IN}}{p} + \frac{\sqrt{\text{IN} \cdot \text{OUT}}}{p})$. Note that the value of τ is set to achieve the minimum.

This algorithm can be extended to arbitrary acyclic joins with the same load complexity, but the decomposition is much more complicated based on the join tree of acyclic join. The challenge is still to bound the size of any intermediate join result as $O(\mathrm{IN}\sqrt{\mathrm{OUT}})$. We refer readers to [20] for algorithmic details. Meanwhile, a lower bound has been presented for any non-r-hierarchical acyclic join: For any $\mathrm{IN} \leq \mathrm{OUT} \leq c \cdot p \cdot \mathrm{IN}$ for some constant c, there exists an instance $\mathcal{R}$ with input size $\Theta(\mathrm{IN})$ and output size $\Theta(\mathrm{OUT})$, such that any tuple-based algorithm computing it in $O(1)$ rounds must have a load of $\Omega(\min\{\frac{\sqrt{\mathrm{IN}\cdot\mathrm{OUT}}}{p\cdot\log\mathrm{IN}}, \frac{\mathrm{IN}}{\sqrt{p}}\})$. This establishes the output-optimality of the output-sensitive algorithm for $\mathrm{OUT} = O(p \cdot \mathrm{IN})$.

Remark. We have obtained a complete understanding of line-3 join in terms of output-optimality: (1) when $\mathrm{OUT} \leq \mathrm{IN}$, the Yannakakis algorithm has linear load $O(\frac{\mathrm{IN}}{p})$; (2) when $\mathrm{IN} < \mathrm{OUT} \leq c\cdot p\cdot\mathrm{IN}$, the lower bound becomes $\tilde{\Omega}(\frac{\sqrt{\mathrm{IN}\cdot\mathrm{OUT}}}{p})$, which is matched by the output-sensitive algorithm; (3) when $\mathrm{OUT} \geq c\cdot p\cdot\mathrm{IN}$, the lower bound is $\Omega(\frac{\mathrm{IN}}{\sqrt{p}})$, which is matched by the worst-case optimal algorithm [19, 26, 20]. In particular, this means that when OUT is large enough, the load complexity of the join is no longer output-sensitive. This also stands in contrast with the RAM model, where the complexity of acyclic joins always grows linearly with OUT. On more complicated joins, the worst-case optimal algorithms have a higher load, and the output-optimality for OUT values in the middle is still unclear.

4.2 Worst-case Optimal Algorithm

In the output-sensitive algorithm, the original join is divided into $O(2^{|\mathcal{E}|})$ pieces, which is still a constant as long as the query has constant size. However, to target the worst-case optimal algorithm, a more fine-grained decomposition of the original join is needed, and intermediate join results should be dealt with more carefully.

The framework of this worst-case optimal algorithm is also based on the join tree of acyclic join: Each time it peels one leaf relation off and reduces the original join into a smaller one until it becomes empty. Eventually, each relation is divided into disjoint partitions of size $\Theta(L)$, where $L = O(\frac{\mathrm{IN}}{p^{1/\rho^*}})$ is the target of the worst-case optimal algorithm for computing acyclic joins, and each piece of subjoin query involves exactly one partition from each relation. In this way, each subjoin can be computed locally and join results are emitted directly without generating any intermediate join result.

Line-3 join [19, 26, 20]. We next present an algorithm for line-3 join with load $O(L)$, where $L = \frac{\mathrm{IN}}{\sqrt{p}}$.

Similarly, we first compute degrees for values of attribute B in relation R_1. A value $b \in \mathrm{dom}(B)$ is *heavy* if it has degree greater than L in R_1 and *light* otherwise. Let B^H, B^L be the set of heavy and light values in B respectively. We further divide B^L into $k = O(\frac{N_1}{\tau})$ disjoint groups $B_1, B_2, \cdots, B_k$ such that values in each group have total degree $\Theta(\tau)$ in relation R_1. The original join is decomposed into following subjoins:

$$\mathcal{Q}_1 = \bigcup_{b\in B^H} (R_1(A,b) \times R_2(b,C) \bowtie R_3(C,D))$$

$$\mathcal{Q}_2 = \bigcup_i (R_1(A,B_i) \bowtie R_2(B_i,C) \bowtie R_3(C,D))$$

For a subjoin query induced by heavy value b in $\mathcal{Q}_1$, we compute the Cartesian product between tuples in $R_1(A,b)$ and results of $R_2(b,C) \bowtie R_3(C,D)$. To compute these subjoins in parallel, we allocate servers proportional to the degree of b in relation R_1. For a subjoin query induced by group B_i in $\mathcal{Q}_2$, we allocate $\sqrt{p}$ servers and compute $R_1(A,B_i) \bowtie R_2(B_i,C) \bowtie R_3(C,D)$ by broadcasting tuples in $R_1(A,B_i)$ and invoking the optimal binary-join algorithm for $R_2(B_i,C) \bowtie R_3(C,D)$.

4.3 Free-Connex Join-Aggregate Queries

We now present a primitive through which any free-connex join-aggregate query can be transformed into a full join, running in $O(1)$ rounds with linear load.

In the preprocessing step, we remove the dangling tuples and reduce the query. We find a free-connex width-1 GHD $\mathcal{T}$ of $\mathcal{Q}$ [6, 5]. Note that the nodes of $\mathcal{T}$ also define a hypergraph, and can be regarded as another join-aggregate query, but with the property that it has a free-connex subset $\mathcal{T}'$ such that $\mathbf{y} = \bigcup_{u\in\mathcal{T}'} u$. We construct an instance $\mathcal{R}_\mathcal{T} = \{R(u) : u \in \mathcal{T}\}$ such that $\mathcal{Q}_\mathbf{y}(\mathcal{R}) = \mathcal{T}(\mathcal{R}_\mathcal{T})$, where $\mathcal{T}(\mathcal{R}_\mathcal{T})$ denotes the result of running the query defined by $\pi_\mathbf{y}\mathcal{T}$ on $\mathcal{R}_\mathcal{T}$. Observe that on a reduced $\mathcal{Q}$, the condition $e \subseteq u$ in property (2) of a width-1 GHD can be replaced by $e = u$, since if $e \subset u$ and $u \subseteq e'$ for some other $e' \in \mathcal{E}$ due to property (3), we would find $e \subset e'$. This implies that $\mathcal{T}$ has only two types of nodes: (1) all hyperedges in $\mathcal{E}$, and (2) nodes that are a proper subset of some $e \in \mathcal{E}$. Then we construct $\mathcal{R}_\mathcal{T}$ as follows. For each $u \in \mathcal{T}$ of type (1), we set $R(u) := R(e)$ where $e = u$; for each $u \in \mathcal{T}$ of type (2), we set $R(u) := R(e)$ for any $e \in \mathcal{E}, u \subset e$, but the annotations of all tuples in $R(u)$ are set to 1 (the $\otimes$-identity). Then, we only focus on computing $\mathcal{T}(\mathcal{R}_\mathcal{T})$.

LEMMA 1. *Given any free-connex width-1 GHD $\mathcal{T}$ and an instance $\mathcal{R}_\mathcal{T}$, an instance $\mathcal{R}_{\mathcal{T}'}$ can be returned in $O(1)$ rounds with linear load such that $\mathcal{T}(\mathcal{R}_\mathcal{T}) = \mathcal{T}'(\mathcal{R}_{\mathcal{T}'})$, where $\mathcal{T}'$ is the free-connex subset of $\mathcal{T}$.*

By plugging to the optimal two-way join algorithm in [8, 22], the aggregated Yannakakis algorithm [23] can aggregate over all the non-output attributes, returning a modified query $\mathcal{T}'(\mathcal{R}_{\mathcal{T}'})$ that only has the output attributes. Because $\mathcal{T}'$ is an acyclic join, thus any results in Section 4.2 and Section 4.1 can be applied to $\mathcal{T}'$.

Moreover, the join size of a (non-aggregate) join is a special join-aggregate query with $\mathbf{y} = \emptyset$, without any circular dependency here, which must be free-connex. Thus, for any acyclic join $\mathcal{Q}$ and any instance $\mathcal{R}$, $|\mathcal{Q}(\mathcal{R})|$ can be computed in $O(1)$ rounds with linear load.

5. SPARSE MATRIX MULTIPLICATION

In this section, we review the output-optimal algorithm [21] for sparse matrix multiplication problem, i.e., $\sum_B R_1(A,B) \bowtie R_2(B,C)$, which is the simplest non-free-connex query. Let N_1, N_2 be the sizes of R_1, R_2 respectively.

First, if $N_1 = 1$ (resp. $N_2 = 1$), the problem can be trivially solved by simply broadcasting the only tuple in R_1 (resp. R_2) with $O(1)$ load. In general, for any $N_1, N_2 \geq 2$, it can be solved in $O(1)$ rounds with

$$\widetilde{O}\left(\frac{N_1+N_2}{p} + \min\{\sqrt{\frac{N_1 N_2}{p}}, \frac{N_1^{1/3} \cdot N_2^{1/3} \cdot \mathrm{OUT}^{1/3}}{p^{2/3}}\}\right)$$

load, with probability at least $1 - 1/N^{O(1)}$.

One can verify that this presents an asymptotic improvement over the Yannakakis algorithm for $\mathrm{OUT} = \omega(1)$. In fact, our algorithm performs the same amount of computation as the Yannakakis algorithm and computes all $O(\mathrm{IN}\sqrt{\mathrm{OUT}})$ elementary products, which is unavoidable in the semiring model. The key to the reduction in load is *locality*, namely, we arrange these elementary products to be computed on the servers in such a way that most of them can be aggregated locally. The standard Yannakakis algorithm has no locality at all, and all the elementary products are shuffled around.

Observation 1. *If $N_1 > pN_2$ or $N_2 > pN_1$, matrix multiplication can be computed with linear load.*

We start with Observation 1, in which two cases can be tackled just by sorting (an MPC primitive). Below, we assume $1/p < N_1/N_2 < p$.

5.1 Worst-case optimal algorithm

We first describe an algorithm with load $O(\sqrt{\frac{N_1 N_2}{p}})$. This is actually worst-case optimal because when there is a single value in the domain of attribute B, there are $N_1 N_2$ elementary products. A server with load L can compute $O(L^2)$ of them in a constant number of rounds, so we have $pL^2 = \Omega(N_1 N_2)$, i.e., $L = \Omega(\sqrt{\frac{N_1 N_2}{p}})$.

Set $L = \sqrt{\frac{N_1 N_2}{p}}$. We first compute all degrees for values in attributes A and C. A value $a \in \mathrm{dom}(A)$ (resp. $c \in \mathrm{dom}(C)$) is *heavy* if it has degree greater than L in R_1 (resp. R_2), and *light* otherwise. The set of heavy and light values in A (resp. C) is denoted as A^H and A^L (resp. C^H and C^L). Then, the original query can be decomposed into four subqueries:

$$\sum_B R_1(A^?, B) \bowtie R_2(B, C^!),$$

where ?, ! can be either H or L. Note that the results produced by these subqueries are disjoint and the final aggregated result is just their union. We handle each subquery separately.

Case (1): At least one of A, C is heavy. W.l.o.g., assume A is heavy. We use the aggregated Yannakakis to compute $\sum_B R_1(A^H, B) \bowtie R_2(B, C)$ with load $O(\frac{J}{p})$, where $J = |R_1(A^H, B) \bowtie R_2(B, C)| = O\left(\frac{N_1 N_2}{\tau}\right)$ since each tuple in R_2 can join with at most $\frac{N_1}{\tau}$ values in A^H.

Case (2): Both A, C are light. We divide A^L into $k = O(\frac{N_1}{L})$ disjoint groups $A_1, A_2, \cdots, A_k$ such that each group has total degree $O(L)$ in $R_1(A^L, B)$ (an MPC primitive) as well as $l = O(\frac{N_2}{L})$ disjoint groups $C_1, C_2, \cdots, C_l$ for C^L such that each group has total degree $O(L)$ in $R_2(B, C^L)$. Then we arrange all servers into a $\lceil \frac{N_1}{L} \rceil \times \lceil \frac{N_2}{L} \rceil$ grid, where each one is associated with (i, j) for $i \in [\lceil \frac{N_1}{L} \rceil], j \in [\lceil \frac{N_2}{L} \rceil]$. The server (i, j) receives all tuples in $R_1(A_i, B), R_2(B, C_j)$ and then compute the subquery $\sum_B R_1(A_i, B) \bowtie R_2(B, C_j)$ locally.

5.2 Output-sensitive algorithm

We first compute a constant-factor approximation of OUT, which should be known by the algorithm in advance. Another important observation on OUT is that any matrix multiplication can be computed with a load $O(\mathrm{OUT} + \frac{\mathrm{IN}}{p})$, through sorting and reduce-by-key primitives. Below, we consider the case that $\mathrm{OUT} > \frac{N_1+N_2}{p}$.

Observation 2. *Matrix multiplication can be computed with load $\widetilde{O}(\mathrm{OUT} + \frac{N_1+N_2}{p})$.*

We next show an algorithm with load $O(L)$, where

$$L = \frac{N_1^{1/3} \cdot N_2^{1/3} \cdot \mathrm{OUT}^{1/3}}{p^{2/3}} + \frac{N_1 + N_2}{p}.$$

Slightly different from the previous algorithm, value $a \in \mathrm{dom}(A)$ is *heavy* if it participates in more than $\tau = \sqrt{\frac{N_2 \cdot \mathrm{OUT} \cdot L}{N_1}}$ final aggregate results, and *light* otherwise. Note that there are at most $\frac{\mathrm{OUT}}{\tau}$ values in A^H since aggregate results by different a's are disjoint.

Case (1): A is heavy. We use the aggregated Yannakakis to compute $\sum_B R_1(A^H, B) \bowtie R_2(B, C)$ with load $O(\frac{J}{p})$, where $J = |R_1(A^H, B) \bowtie R_2(B, C)| = O\left(\frac{N_2 \cdot \mathrm{OUT}}{\tau}\right)$, since each tuple in R_2 can join with at most $\frac{\mathrm{OUT}}{\tau}$ values in A^H.

Case (2): A is light. We divide A^L into $k_1 = O(\frac{\mathrm{OUT}}{\tau})$ disjoint groups $A_1, A_2, \cdots, A_{k_1}$ such that values in each group appear in $O(\tau)$ final results. On group A_i, value $c \in \mathrm{dom}(C)$ is *heavy* if it appears in more than L results of the subquery $\sum_B \sigma_{A \in A_i} R_1(A, B) \bowtie R_2(B, C)$, and *light* otherwise. The set of heavy and light values in $\mathrm{dom}(C)$, with respect to A_i, is denoted as C_i^H and C_i^L. Observe that $|C_i^H| \leq \frac{\tau}{L}$.

Case (2.1): C is heavy. For each group A_i, we use the aggregated Yannakakis algorithm to compute $\sum_B R_1(A_i, B) \bowtie R_2(B, C_i^H)$ with $J_i = |R_1(A_i, B) \bowtie R_2(B, C_i^H)| = O(|R_1(A_i, B)| \cdot \frac{\tau}{L})$, since each tuple in $R_1(A_i, B)$ can join with at most $\frac{\tau}{L}$ values in C_i^H. To compute all groups in parallel, we allocate servers proportional to the input sizes of each group and achieve a uniform load of $O(L)$ for all groups.

Case (2.2): C is light. For each group A_i, we divide C_i^L into $k_2 = O(\sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_2}{N_1}})$ disjoint groups C_1^i, C_2^i, $\cdots$, $C_{k_2}^i$ such that values in each group appear together in $O(L)$ results of the subquery $\sum_B R_1(A_i, B) \bowtie R_2(B, C)$. Note that each pair of (A_i, C_j^i) further defines a subquery as $\sum_B R_1(A_i, B) \bowtie R_2(B, C_j^i)$, which has output size smaller than L. Thus, by allocating servers proportional to the input sizes, we can reduce it to the case in Observation 2, achieving a uniform load of $O(L)$ for all subqueries.

5.3 Lower Bound

We mention the following two lower bounds, which together show that the upper bound achieved is optimal when $N_1, N_2 \geq 2$ and $\max\{N_1, N_2\} \leq \text{OUT} \leq N_1 N_2$.

THEOREM 2. *For any $N_1, N_2 \geq 2$, there exists an instance $\mathcal{R}$ for matrix multiplication with input sizes N_1, N_2 such that any algorithm computing it must incur a load of $\Omega\left(\frac{N_1+N_2}{p}\right)$ in the semiring MPC model.*

THEOREM 3. *For any $1/p \leq N_1/N_2 \leq p$ and $1 \leq \text{OUT} \leq N_1 N_2$, there exists an instance $\mathcal{R}$ for sparse matrix multiplication with input sizes N_1, N_2 and output size OUT, such that any algorithm computing it in $O(1)$ rounds in the semiring MPC model must incur a load of $\Omega\left(\min\left\{\sqrt{\frac{N_1 N_2}{p}}, \frac{N_1^{1/3} \cdot N_2^{1/3} \cdot \text{OUT}^{1/3}}{p^{2/3}}\right\}\right)$.*

5.4 General Tree Queries

In [21], an output-sensitive algorithm based on matrix multiplication is also proposed for general tree queries. However, an inherent difficulty for tree query is that it is not known how to compute a constant-factor approximation of OUT without actually computing all the query results. One standard technique is to repeatedly double a guess of OUT and try to run the algorithm, until the guess is correct (i.e., within a constant factor of the true value). This would work in a sequential model, since the running times of the successive guesses will form a geometric series, increasing the total running time by only a constant factor. In the parallel model like the MPC, although the total load can still be bounded, but the repeated guesses would lead to $O(\log N)$ rounds of computation. The idea to get around this is to make the algorithm *oblivious* to the value of OUT, i.e., the value of OUT is not needed by the algorithm but only used in the analysis. All details, including how to reduce the orignal query into a matrix multiplication problem, and how to bound the sizes of intermediate query results with OUT, are referred to [21].

6. GRAPH JOIN

Graph join, as a special class of cyclic joins, enjoy very nice properties as follows: (1) $\tau^* \leq \rho^*$; (2) $\tau^* + \rho^* = |\mathcal{V}|$; (3) τ^* and ρ^* admit half-integral solutions. In plain language, a graph join always has its optimal fractional edge packing number smaller than edge covering number; moreover, every edge takes a value in $\{0, \frac{1}{2}, 1\}$ in the optimal solution for edge cover and packing. Those three properties are taken full use by algorithm design.

In [24], Ketsman and Suciu gave an algorithm in the MPC model for computing the graph join with load $O(\frac{\text{IN}}{p^{1/\rho^*}})$. Later, this complicated algorithm was simplified by Tao [31], and the number of rounds required was decreased from 7 to 3. These two algorithms are based on the HYPERCUBE algorithm [2, 8], which arranges servers into a hypercube where each dimension corresponds to one attribute. We first mention one important property of this algorithm on graph joins, resilient to the data skew. Let $\mathbf{p}$ be a function mapping each attribute x to a positive integer $\mathbf{p}_x$. Instance $\mathcal{R}$ is *skew-free* with respect to $\mathbf{p}$ if for each relation $R(e)$ with $e = \{x, y\}$, each value of $\text{dom}(x)$ has degree at most $\frac{|R(e)|}{\mathbf{p}_x}$ and each value of $\text{dom}(y)$ has degree at most $\frac{|R(e)|}{\mathbf{p}_y}$. For a graph join $\mathcal{Q}$ and a skew-free instance $\mathcal{R}$ under $\mathbf{p}$, the join result $\mathcal{Q}(\mathcal{R})$ can be computed using $\prod_{x \in \mathcal{V}} \mathbf{p}_x$ servers in a single round with load complexity $O(\text{IN}/\min_{e \in \mathcal{E}} \prod_{v \in e} \mathbf{p}_v)$.

For a graph join $\mathcal{Q}$ and an instance $\mathcal{R}$, if each value of any attribute has degree smaller than $\frac{\text{IN}}{p^{1/2\rho^*}}$, then this is a skew-free instance w.r.t. $\mathbf{p}_x = p^{1/2\rho^*}$ for every attribute $x \in \mathcal{V}$. Implied by the result above, such an instance can be computed using $p^{|\mathcal{V}|/2\rho^*} \leq p$ servers in a single round with load $O(\frac{\text{IN}}{p^{1/\rho^*}})$. The challenge comes when skew exists. The high-level idea is to decompose the original join into a set of skew-free subjoins and allocate servers appropriately for computing all subjoins in parallel while achieving a uniform load of $O(\frac{\text{IN}}{p^{1/\rho^*}})$.

Set $\tau = \frac{\text{IN}}{\lambda}$. For each attribute x, it divides values in $\text{dom}(x)$ into *heavy* and *light*. More specifically, value $a \in \text{dom}(x)$ is *heavy* if there exists a relation e for $x \in e$ such that a has degree more than τ in $R(e)$, and *light* otherwise. Then, the join results can be distinguished into $O(2^{|\mathcal{V}|})$ cases, such that each case corresponds to one subset of attributes $S \subseteq \mathcal{V}$ and each join results fall into this case has heavy values in S and light values in $\mathcal{V} - S$. Fixing one subset of attributes S, there are $O(\lambda^S)$ different combinations of heavy values over S, and each one is noted as *configuration*. In this way, the original join is divided into a set of subjoins, each one corresponds to a residual query by fixing a configuration. Consider a subjoin defined by a configuration over attributes S. Observe that all values in $\text{dom}(x)$ for $x \in \mathcal{V} - S$ are light,thus this is a skew-free instance. The HYPERCUBE algorithm is then applied for each subjoin independently. More algorithmic details, including how to allocate servers for each subjoin, and semi-join reduction, can be found in [31].

7. LOWER BOUNDS FOR CYCLIC JOINS

In this section, we review two lower bounds for cyclic joins, one is an output-sensitive lower bound for the triangle join $\mathcal{Q}_\triangle = R_1(B, C) \bowtie R_2(A, C) \bowtie R_3(A, B)$ [20], and the other is an worst-case optimal lower bound for the box-minus join $\mathcal{Q}_\boxminus = R_1(A, B, C) \bowtie R_2(D, E, F) \bowtie R_3(A, D) \bowtie R_4(B, E) \bowtie R_5(C, F)$ [18], both of which verify that cyclic joins are inherently more difficult than acyclic joins. The high-level idea of these lower bound proofs is still resorted to the counting argument. It first show how to construct an probabilistic instance such that it will have a bounded $J(L)$, the maximum number of join results a server can produce, if it loads at most

L tuples from each relation. Then setting $p \cdot J(L) = \Omega(|\mathcal{Q}(\mathcal{R})|)$ yields a lower bound on L. Any attempts in lowering this bound further would break the counting argument.

7.1 A lower bound for Triangle Join

For $\mathcal{Q}_\triangle$, a worst-case lower bound of $\Omega(\frac{\text{IN}}{p^{2/3}})$ is known, by the counting argument. However, if OUT is also used as a parameter, this argument only leads to a lower bound of $\Omega((\frac{\text{OUT}}{p})^{\frac{3}{2}})$. This lower bound has been improved to $\Omega(\min\{\frac{\text{IN}}{p} + \frac{\text{OUT}}{p \log N}, \frac{\text{IN}}{p^{2/3}}\})$. The proof given in [20], is quite technical, but the intuition is simple: When $\text{OUT} = \Theta(\text{IN}^{\frac{3}{2}})$, the triangles are "dense" enough, so a server can achieve the maximum efficiency and emit $\Theta(L^{\frac{3}{2}})$ triangles. However, for small OUT, we can construct an instance in which the triangles are "sparse" so that a server cannot be as efficient. In fact, an instance constructed randomly (in a certain way) would have this property with high probability. This lower bound has the following consequences:

When $\text{OUT} \geq \text{IN} \cdot p^{1/3}$ for some constant c, the lower bound becomes $\tilde{\Omega}(\frac{\text{IN}}{p^{2/3}})$, which means that the worst-case optimal algorithm of [26] is actually also output-optimal in this parameter range. Finding $\tilde{\Omega}(\text{IN} \cdot p^{1/3})$ triangles is as difficult as finding $\Theta(\text{IN}^{3/2})$ triangles.

When $\text{IN} \leq \text{OUT} \leq \text{IN} \cdot p^{1/3}$, the lower bound becomes $\tilde{\Omega}(\frac{\text{OUT}}{p})$ while we do not have a matching upper bound yet. Nevertheless, this already exhibits a separation from acyclic joins, which can be done with load $O(\frac{\sqrt{\text{IN} \cdot \text{OUT}}}{p})$, with a gap being at least $\tilde{\Omega}(\sqrt{\frac{\text{OUT}}{\text{IN}}})$.

7.2 A lower bound for Box-minus Join

When studying the worst-case optimal algorithms for cyclic joins, it is surprisingly observed that $O(\frac{\text{IN}}{p^{1/\rho^*}})$ is not necessarily a correct target for multi-round worst-case optimal join algorithms [18]. An open question posed in [29, 24] was answered: For $\mathcal{Q}_\boxminus$, whether there exists a better upper bound than $\widetilde{O}(\frac{\text{IN}}{p^{1/3}})$, or a better lower bound than $\Omega(\frac{\text{IN}}{p^{1/2}})$? Note that $\mathcal{Q}_\boxminus$ has optimal fractional edge covering number $\rho^* = 2$ and optimal fractional edge packing number $\tau^* = 3$. [18] proves a higher lower bound $\Omega(\frac{\text{IN}}{p^{1/3}})$ for $\mathcal{Q}_\boxminus$.

The intuition for proving this lower bound is that a probabilistic instance could be shown when there are $\Theta(\text{IN}^2)$ join results, a server cannot be as efficient since the input instance is "sparse" enough. For the remaining cyclic joins (except LW join and graph join), there are no other lower bounds. Meanwhile, the existing algorithm [26] can compute it in a single round with load $\widetilde{O}(\frac{\text{IN}}{p^{1/3}})$[4], which is already worst-case optimal implied by this new lower bound.

8. CONCLUSION

[4]$\mathcal{Q}_\boxminus$ has optimal fractional edge quasi-packing number as 3.

In this article, we have surveyed recent results on computing join and join-aggregate queries in the MPC model. While most of them are theoretical in nature, experimental results have been presented in [12, 22], showing that some of the algorithmic ideas can lead to practical performance gains with certain engineering efforts. We conclude by summarizing several key results and posing some open questions:

- The instance-optimality can be achieved if and only if the join query is r-hierarchical.
- Beyond r-hierarchical joins, the output-optimality has only been achieved on acyclic joins if the output size is small, by an output-sensitive algorithm. Note that this algorithm is not always optimal, at least when the output size is large. However, there is no result on the output-optimal upper bound for cyclic joins in the MPC model, even for the triangle join. On the other hand, a lower bound for the triangle join indicates an inherent gap between the acyclic joins and cyclic joins in terms of the dependence on OUT.
- Worst-case optimal algorithms with load $O(\frac{\text{IN}}{p^{1/\rho^*}})$ have been discovered for acyclic joins and some specific class of cyclic joins (graph join and LW join). On the other hand, a recent edge-packing lower bound for the box-minus join $\mathcal{Q}_\boxminus$ shows that $O(\frac{\text{IN}}{p^{1/\rho^*}})$ is not achievable for all queries. It is thus an intriguing question to determine the worst-case complexity for the remaining cyclic joins. Would it be $\Omega(\text{IN}/p^{1/\max\{\rho^*,\tau^*\}})$?
- For join-aggregregate query, if it is free-connex, it can be transformed into a full join query through primitive operations and then enjoy all the results for full-join queries. For non-free-connex queries, the output-optimal algorithm has been achieved only for the matrix multiplication query. For other non-free connex queries, some output-sensitive algorithms have been developed, but without any non-trivial lower bound.

9. REFERENCES

[1] F. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A multiround join algorithm in MapReduce. In *Proc. International Conference on Database Theory*, 2017.

[2] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, 2011.

[3] R. R. Amossen and R. Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*, pages 121–126. ACM, 2009.

[4] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[5] G. Bagan. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques*. PhD thesis, Université de Caen, 2009.

[6] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

[7] N. Bakibayev, T. Kocisky, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorised databases. In *Proc. International Conference on Very Large Data Bases*, 2013.

[8] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *Proc. ACM Symposium on Principles of Database Systems*, 2014.

[9] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. *Journal of the ACM (JACM)*, 64(6):40, 2017.

[10] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM (JACM)*, 30(3):479–513, 1983.

[11] A. Björklund, R. Pagh, V. V. Williams, and U. Zwick. Listing triangles. In *International Colloquium on Automata, Languages, and Programming*, pages 223–234. Springer, 2014.

[12] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2015.

[13] E. Cohen. Estimating the size of the transitive closure in linear time. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 190–200. IEEE, 1994.

[14] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.

[15] M. T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999.

[16] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching and simulation in the mapreduce framework. In *Proc. International Symposium on Algorithms and Computation*, 2011.

[17] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc. ACM Symposium on Principles of Database Systems*, 2007.

[18] X. Hu. Cover or pack: New upper and lower bounds for massively parallel joins, `https://users.cs.duke.edu/~xh102/cover.pdf`. Technical report, Duke University, 2020.

[19] X. Hu and K. Yi. Towards a worst-case I/O-optimal algorithm for acyclic joins. In *Proc. ACM Symposium on Principles of Database Systems*, 2016.

[20] X. Hu and K. Yi. Instance and output optimal parallel algorithms for acyclic joins. In *Proc. ACM Symposium on Principles of Database Systems*, 2019.

[21] X. Hu and K. Yi. Parallel algorithms for sparse matrix multiplication and join-aggregate queries. In *Proc. ACM Symposium on Principles of Database Systems*, 2020.

[22] X. Hu, K. Yi, and Y. Tao. Output-optimal massively parallel algorithms for similarity joins. *ACM Transactions on Database Systems*, 44(2):6, 2019.

[23] M. R. Joglekar, R. Puttagunta, and C. Ré. AJAR: Aggregations and joins over annotated relations. In *Proc. ACM Symposium on Principles of Database Systems*, 2016.

[24] B. Ketsman and D. Suciu. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries. In *Proc. ACM Symposium on Principles of Database Systems*, 2017.

[25] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions asked frequently. In *Proc. ACM Symposium on Principles of Database Systems*, 2016.

[26] P. Koutris, P. Beame, and D. Suciu. Worst-case optimal algorithms for parallel query processing. In *Proc. International Conference on Database Theory*, 2016.

[27] P. Koutris, S. Salihoglu, and D. Suciu. *Algorithmic Aspects of Parallel Data Processing*. Now Publishers, 2018.

[28] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *Proc. ACM Symposium on Principles of Database Systems*, 2011.

[29] P. Koutris and D. Suciu. A guide to formal analysis of join processing in massively parallel systems. *ACM SIGMOD Record*, 45(4):18–27, 2017.

[30] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proc. ACM Symposium on Principles of Database Systems*, pages 37–48, 2012.

[31] Y. Tao. A simple parallel algorithm for natural joins on binary relations. In *23rd International Conference on Database Theory*, 2020.

[32] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[33] T. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.

[34] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. International Conference on Very Large Data Bases*, pages 82–94, 1981.