Instance and Output Optimal Parallel Algorithms for Acyclic Joins[*]

Xiao Hu
HKUST
xhuam@cse.ust.hk

Ke Yi
HKUST
yike@cse.ust.hk

ABSTRACT

Massively parallel join algorithms have received much attention in recent years, while most prior work has focused on worst-optimal algorithms. However, the worst-case optimality of these join algorithms relies on hard instances having very large output sizes, which rarely appear in practice. A stronger notion of optimality is *output-optimal*, which requires an algorithm to be optimal within the class of all instances sharing the same input and output size. An even stronger optimality is *instance-optimal*, i.e., the algorithm is optimal on every single instance, but this may not always be achievable.

In the traditional RAM model of computation, the classical Yannakakis algorithm is instance-optimal on any acyclic join. But in the massively parallel computation (MPC) model, the situation becomes much more complicated. We first show that for the class of r-hierarchical joins, instance-optimality can still be achieved in the MPC model. Then, we give a new MPC algorithm for an arbitrary acyclic join with load $O(\frac{\text{IN}}{p} + \frac{\sqrt{\text{IN} \cdot \text{OUT}}}{p})$, where IN, OUT are the input and output sizes of the join, and p is the number of servers in the MPC model. This improves the MPC version of the Yannakakis algorithm by an $O(\sqrt{\frac{\text{OUT}}{\text{IN}}})$ factor. Furthermore, we show that this is output-optimal when $\text{OUT} = O(p \cdot \text{IN})$, for every acyclic but non-r-hierarchical join. Finally, we give the first output-sensitive lower bound for the triangle join in the MPC model, showing that it is inherently more difficult than acyclic joins.

[*]This work was supported by Hong Kong RGC under grants 16200415, 16202317, and 16201318, and by grants from Microsoft and Alibaba.

PODS'19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6227-6/19/06…$15.00
https://doi.org/10.1145/3294052.3319698

CCS CONCEPTS

• **Theory of computation → Database query processing and optimization (theory)**.

KEYWORDS

MPC algorithms; acyclic joins

ACM Reference Format:
Xiao Hu and Ke Yi. 2019. Instance and Output Optimal Parallel Algorithms for Acyclic Joins. In *38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3294052.3319698

1 INTRODUCTION

A (natural) *join* is defined 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*, where each tuple is an assignment that assigns a value from $\text{dom}(x)$ to x for every $x \in e$. We use $\text{IN} = \sum_{e \in \mathcal{E}} |R(e)|$ to denote the size of $\mathcal{R}$. The *join results* of $\mathcal{Q}$ on $\mathcal{R}$, denoted as $\mathcal{Q}(\mathcal{R})$, consist of all combinations of tuples, one from each $R(e)$, such that they share common values on their common attributes. Let $\text{OUT} = |\mathcal{Q}(\mathcal{R})|$ be the output size. We study the *data complexity* of join algorithms, i.e., we assume that the query size, namely n and m, are constants. In this paper, we focus on *acyclic joins*, i.e., when the hypergraph $\mathcal{Q}$ is acyclic (formal definition given later).

1.1 The model of computation

The problem gets much more interesting in the parallel setting. In this paper, we consider the *massively parallel computation* (MPC) model [1, 2, 6, 7, 21, 23, 25], which has become the standard model of computation for studying massively parallel algorithms, especially for join algorithms.

In the MPC model, data is initially distributed evenly over p servers with each server holding IN/p tuples. Computation proceeds in rounds. In each round, each server first sends messages to other servers, receives messages from other servers, and then does some local computation. 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. A *linear load* $L = O(\frac{\text{IN}}{p})$ is the ideal case (since the initial load is already $\frac{\text{IN}}{p}$), while if $L = O(\text{IN})$, all problems can be solved trivially in one round by simply sending all data to one server. Initial efforts were mostly spent on what can be done in a single round of computation [2, 6, 7, 23, 25, 25], but recently, more interest has been given to multi-round (but still a constant) algorithms [1, 21, 23], since new main memory based systems, such as Spark and Flink, have much lower overhead per round than previous generations like Hadoop.

The MPC model can be considered as a simplified version of the BSP model [31], 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 with 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.

All our algorithms work under 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 IN bits, in $O(1)$ rounds. Our lower bounds hold under $\text{IN} \geq p^c$ for some constant c, which may depend on the particular lower bound construction.

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 Q on instance $\mathcal{R}$ if the following is achieved: For any join result $(t_1, \ldots, t_m) \in Q(\mathcal{R})$ where $t_i \in R(e_i)$, $i = 1, \ldots, m$, these m tuples (or their copies) must all be present on the same server at some point. Then the server will call a zero-cost function $emit(t_1, \ldots, t_m)$ to report the join result. Note that since we only consider constant-round algorithms, whether a server is allowed to keep the tuples it has received from previous rounds is irrelevant: if not, it can just keep sending all these tuples to itself over the rounds, increasing the load by a constant factor. All known join algorithms in the MPC model are tuple-based and obey these requirements. Our lower bounds are combinatorial in nature: we only count the number of tuples that must be communicated in order to emit all join results, while all other information can be communicated for free. The upper bounds include all messages, with a tuple and an integer of $O(\log \text{IN})$ bits both counted as 1 unit of communication.

1.2 Instance and output optimality

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. For many important computational problems, this is too coarse-grained and cannot accurately characterize the performance of the algorithm. For the join problem, no algorithm can do better than $O(\text{IN}^{1/\rho})$ time in the worst case, where ρ is the fractional edge cover number of the hypergraph Q [28, 32]. This bound drastically overestimates the running time on most typical instances.

A more refined approach is *parameterized analysis*, which further subdivides the instance space into smaller classes by introducing more parameters that supposedly 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

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

where $L_{\mathcal{A}}(\mathcal{R})$ denotes the load of $\mathcal{A}$ on $\mathcal{R}$. 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 extreme case when each class contains just one instance, we obtain instance-optimal algorithms. More precisely, an 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 A'. Note that by definition, an instance-optimal algorithm must be output-optimal, and an output-optimal algorithm must be worst-case optimal, but the reserve direction may not be true.

In the traditional RAM model of computation, the classical Yannakakis algorithm [33] 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 all the inputs[1] and $\Omega(\text{OUT})$ time to enumerate the outputs. Thus, the two

[1]To formally prove this claim, one will have to be more careful with the family of algorithms under consideration. In particular, if OUT = 0, then the algorithm may not have to do anything. One possible approach is to ask the algorithm to produce a *certificate* in addition to the join results [27]. We

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 $\Re(\text{IN}, \text{OUT})$ have exactly the same complexity $O(\text{IN} + \text{OUT})$.

1.3 Join algorithms in the MPC model

The situation becomes much more interesting in the MPC model. First, it has been observed that the Yannakakis algorithm can be easily implemented in the MPC model with a load of $O(\frac{\text{IN}}{p} + \frac{\text{OUT}}{p})$ [1][2], but this is not optimal. In particular, it is known that the binary join $R_1(A, B) \bowtie R_2(B, C)$ can be computed with load $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$ [7, 16]. This is optimal by the following simple lower bound argument: Each server can only produce $O(L^2)$ join results in a constant number of rounds with the load limited to L, so all the p servers can produce at most $O(p \cdot L^2)$ join results. Thus, producing OUT join results needs at least a load of $L = \Omega(\sqrt{\frac{\text{OUT}}{p}})$. Meanwhile, since $L \geq \text{IN}/p$ by definition, the $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$ bound is optimal. Note that this argument can be applied on a per-instance basis, which means that the load complexity of any instance is still precisely captured by IN and OUT, and $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$ is both an instance-optimal and output-optimal bound.

However, when the join involves three relations, the situation becomes subtler, and we start to see a separation between the two notions of optimality, meaning that the load complexity of an instance may not depend only on IN and OUT. Let us start with the simplest 3-relation join $R_1(A) \bowtie R_2(B) \bowtie R_3(C)$, i.e., computing the Cartesian product of 3 sets of tuples. Consider a particular class $\Re(\text{IN}, \text{OUT})$ when $\text{OUT} = \text{IN}^2$. Suppose the 3 relations have sizes N_1, N_2, N_3, respectively. Then $\Re(\text{IN}, \text{OUT})$ consists of all instances with $N_1 + N_2 + N_3 = \text{IN}$ and $N_1 N_2 N_3 = \text{OUT} = \text{IN}^2$. Consider the following two instances: (1) $N_1 = N_2 = \Theta(\sqrt{\text{IN}})$, $N_3 = \Theta(\text{IN})$, applying the same argument above except that each server now can produce $O(L^3)$ join results, i.e., $p \cdot L^3 = \Omega(\text{OUT})$, we have $L = \Omega((\frac{\text{OUT}}{p})^{1/3})$; (2) if $N_1 = 1, N_2 = N_3 = \Theta(\text{IN})$, then the problem boils down to computing the Cartesian product of two sets, which has a lower bound of $L = \Omega((\frac{\text{OUT}}{p})^{1/2})$. The reason why instance (2) has a higher lower bound than instance (1) is that it has a higher skew, which causes more difficulty for the MPC model. Note that this phenomenon does not exist in the RAM model, in which both instances (in fact all instances in $\Re(\text{IN}, \text{OUT})$) have the same complexity of $O(\text{IN} + \text{OUT})$. Fundamentally, this is because the MPC model is all about *locality*: An MPC algorithm should strive to bring all related tuples to one server so as to produce as many join results as possible, while a higher skew reduces locality.

We can extend this argument to computing the Cartesian product of m sets of sizes $N_1, \ldots, N_m$. Any algorithm computing the full Cartesian product obviously must also compute the Cartesian product of any subset of the n sets, thus the load must be at least

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

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 $\mathcal{R}$ [7]. Thus, it can be considered as an instance-optimal algorithm for computing Cartesian products, with an optimality ratio of $\log^{O(1)} p$.

The binary join and Cartesian products are the simplest joins. Then the obvious question is, do instance-optimal algorithms exist for larger classes of joins? If not, how about output-optimal algorithms? These are the main questions we wish to address in this paper.

1.4 Classification of acyclic joins

Before describing our results, we first define some sub-classes of acyclic joins.

Acyclic joins [8]. We use the common notion of acyclicity, which is also known as α-acyclicity. A join $Q = (\mathcal{V}, \mathcal{E})$ is *acyclic* if there exists an undirected tree $\mathcal{T}$ whose nodes are in one-to-one correspondence with the edges in $\mathcal{E}$ such that for any vertex $v \in \mathcal{V}$, all nodes containing v form a connected subtree. Such a tree $\mathcal{T}$ is called the *join tree* of Q. Note that the join tree may not be unique for a given Q.

Hierarchical joins [11]. A join $Q = (\mathcal{V}, \mathcal{E})$ 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. Thus, all attributes can be organized into a forest, such that x is a descendant of y iff $\mathcal{E}_x \subseteq \mathcal{E}_y$. Hierarchical joins have been enjoyed nice properties in probabilistic databases [11, 12] and query answering under updates [9], but their role in the MPC model has not been studied so far.

r-hierarchical joins. 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*

will not digress to this direction since this paper is only concerned about MPC algorithms.

[2]The bound stated in [1] is actually $O(\frac{(\text{IN}+\text{OUT})^2}{p})$, because they used a sub-optimal binary join algorithm as the subroutine. Replacing it with the optimal binary join algorithm in [7, 16] yields the claimed bound, as observed in [24].

Joins	Instance-optimal		Output-optimal	
	one-round	multi-round	one-round	multi-round
tall-flat	$(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R})) \log^{O(1)} p$ [7]	$\Theta\left(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R})\right)$	-	
r-hierarchical w/o dangling tuples				
r-hierarchical w/ dangling tuples	$\omega\left(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R})\right)$		$\omega\left(\frac{\text{IN}}{p} + \frac{\text{OUT}}{p}\right)$ [25]	-
acyclic	$\omega\left(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R})\right)$			$\Theta\left(\frac{\text{IN}}{p} + \frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p}\right)$ LB holds for OUT $\leq O(p \cdot \text{IN})$
triangle				$\tilde{\Omega}\left(\min\left\{\frac{\text{IN}}{p} + \frac{\text{OUT}}{p}, \frac{\text{IN}}{p^{2/3}}\right\}\right)$

Table 1: Summary of results.

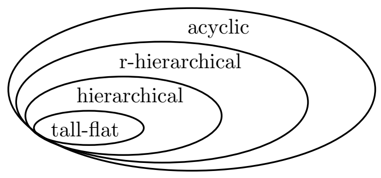

Figure 1: Relationships of joins.

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.

Tall-flat joins [25]. A join $Q = (\mathcal{V}, \mathcal{E})$ 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 must be hierarchical.

The relationships of these joins are illustrated in Figure 1.

1.5 Our results

This paper gives an almost complete characterization of acyclic joins with respect to instance-optimality and output-optimality in the MPC model. Our results are summarized in Table 1, and we explain them below in more detail.

Instance-optimality. First, we extend the Cartesian product lower bound (1) to a general join $Q = (\mathcal{V}, \mathcal{E})$. For any subset of relations $S \subseteq \mathcal{E}$, define

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

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

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

The BinHC algorithm [7] 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 (more detail given in Section 3). Beame et al. [7] show that BinHC is optimal (up to polylog factors) within the class of instances sharing the same degrees, among all one-round MPC algorithms. In this paper, we strengthen this result by giving a new analysis of the BinHC algorithm, showing 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* (a dangling tuple is one that does not appear in the join results). Furthermore, because 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 $O(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R}))$ load, but we can remove the dangling tuples in $O(1)$ rounds with $O(\frac{\text{IN}}{p})$ load [33], and then run then BinHC algorithm. This gives a multi-round, $(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R})) \log^{O(1)} p$-load algorithm, where the $O(1)$ exponent depends on the query size, and is at least m, the number of relations. Then we give a new multi-round algorithm for r-hierarchical joins with load $O(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R}))$, i.e., improving the instance-optimality ratio from $\log^{O(1)} p$ to $O(1)$.

The instance-optimal load $O(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R}))$ is not achievable beyond r-hierarchical joins[3]. More precisely, we show that 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

[3]But instance-optimal algorithms are still possible, if some higher per-instance lower bound can be derived.

multi-round algorithm must incur a load of[4] $\tilde{\Omega}(\frac{\mathrm{IN}}{p^{1/2}})$ on $\mathcal{R}$. This is actually a corollary following our output-sensitive lower bound, which is described next.

Output-optimality. One-round algorithms have severe limitations with respect to OUT: As shown in [25], any non-tall-flat joins must incur load $\omega(\frac{\mathrm{IN}}{p} + \frac{\mathrm{OUT}}{p})$ if only one round is allowed. On the other hand, as mentioned, the classical Yannakakis algorithm is a multi-round MPC algorithm that works for all acyclic joins and has a load of $O(\frac{\mathrm{IN}}{p} + \frac{\mathrm{OUT}}{p})$ [1, 24]. Thus, our focus will be on multi-round algorithms and see if this result can be improved. An instance-optimal algorithm must also be output-optimal, so we have automatically obtained output-optimal algorithms for r-hierarchical joins. In fact, we show that $L_{\mathrm{instance}}(p, \mathcal{R}) = O(\frac{\mathrm{IN}}{p} + \sqrt{\frac{\mathrm{OUT}}{p}})$ for all r-hierarchical joins, so this is already an asymptotic improvement over the Yannakakis algorithm. But the more important question is, how about acyclic joins that are not r-hierarchical?

Our main output-optimal result is a new MPC algorithm for acyclic joins achieving a load of $O(\frac{\mathrm{IN}}{p} + \frac{\sqrt{\mathrm{IN}\cdot\mathrm{OUT}}}{p})$, which is an $O(\sqrt{\frac{\mathrm{OUT}}{\mathrm{IN}}})$-factor improvement from the Yannakakis algorithm. Interestingly enough, we observe that while the join order does not change the running time of the Yannakakis algorithm by more than a constant factor in the RAM model, it does have asymptotic consequences in the MPC model. However, there are instances on which no join order is good, in which case we recursively decompose the join into multiple parts, and choose a good join order for each part. The number of parts is exponential in the query size but constant in terms of data size. To achieve this result, we first give a simple algorithm on the line-3 join $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$ (Section 4), and then extend it to arbitrary acyclic joins (Section 5).

We also give a matching lower bound (up to a log factor), thereby establishing the output-optimality of the algorithm. However, the lower bound only holds when $\mathrm{OUT} = O(p \cdot \mathrm{IN})$. This restriction on OUT is actually inherent, because the $O(\frac{\mathrm{IN}}{p} + \frac{\sqrt{\mathrm{IN}\cdot\mathrm{OUT}}}{p})$ bound cannot be optimal for all values of OUT. When OUT is large enough, a worst-case optimal algorithm will take over. For example, on the line-3 join, the worst-case optimal algorithm, which has load $O(\frac{\mathrm{IN}}{\sqrt{p}})$ [17, 23], becomes better when $\mathrm{OUT} > p\cdot\mathrm{IN}$. Our lower bound actually indicates that the $O(\frac{\mathrm{IN}}{\sqrt{p}})$ bound is output-optimal (though it does not depend on OUT) for all $\mathrm{OUT} > p \cdot \mathrm{IN}$. Thus, we now have a complete understanding of the line-3 join with respect to output-optimality. For more complicated joins, their worst-case optimal algorithms have a higher load, and the output-optimality for OUT values in the middle is still unclear.

Next, we extend these results to *join-aggregate* (including *join-project*) queries that are *free-connex* (formal definition given in Section 6). More precisely, we give an MPC algorithm with linear load that removes all the non-output attributes of the query, converting it into an acyclic join. Then we apply our instance-optimal or output-optimal algorithm on the resulting acyclic join.

Finally in Section 7, we turn to the triangle join $R_1(B, C) \bowtie R_2(A, C) \bowtie R_3(A, B)$, which is the simplest cyclic join, and give the first output-sensitive lower bound $\tilde{\Omega}(\min\{\frac{\mathrm{IN}}{p} + \frac{\mathrm{OUT}}{p}, \frac{\mathrm{IN}}{p^{2/3}}\})$ in the MPC model. Previously, only a worst-case bound of $\Omega(\frac{\mathrm{IN}}{p^{2/3}})$ is known [23, 29] and that construction uses an instance with the maximum possible output size $\mathrm{OUT} = \mathrm{IN}^{3/2}$. Note that the second term in the lower bound is smaller as long as $\mathrm{OUT} = \Omega(\mathrm{IN} \cdot p^{1/3})$, which means that under this parameter range, the $\tilde{O}(\frac{\mathrm{IN}}{p^{2/3}})$-load algorithm [23] is not only worst-case optimal but also output-optimal. For $\mathrm{OUT} = o(\mathrm{IN} \cdot p^{1/3})$, the lower bound becomes $\tilde{\Omega}(\frac{\mathrm{IN}}{p} + \frac{\mathrm{OUT}}{p})$ while we do not have a matching upper bound yet (some explanation on why this is difficult is given below). But at least, this shows a separation from acyclic joins, i.e., cyclic joins are harder than acyclic ones by at least a factor of $\tilde{\Omega}(\sqrt{\frac{\mathrm{OUT}}{\mathrm{IN}}})$.

1.6 Other related results

Most existing work on join algorithms in the MPC model has focused on the worst case. Here, the goal is to achieve a load of $O(\frac{\mathrm{IN}}{p^{1/\rho}})$, where ρ is the fractional edge cover number of the hypergraph Q. So far, this bound has been achieved on Berge-acyclic joins[5] [17], joins where each relation has two attributes (i.e., Q is an ordinary graph) [21], and LW joins [23][6]. Whether this bound can be achieved for arbitrary joins, or even just α-acyclic joins, is still open. Assuming this is achievable, our output-sensitive algorithm is still better when $\mathrm{OUT} = O(p^{2-2/\rho} \cdot \mathrm{IN})$.

Joglekar et al. [19] described a multi-round MPC algorithm for arbitrary joins, whose load complexity depends on IN, OUT, as well as the degrees of the values. However, the load of their algorithm is at least $\Omega(\frac{\mathrm{IN}}{p} + \frac{\mathrm{OUT}}{p})$, i.e., no better than the Yannakakis algorithm on acyclic joins.

In the RAM model, output-sensitive join algorithms have been extensively studied. The running time of most algorithms is in form of $O(\mathrm{IN}^w + \mathrm{OUT})$, where w is certain notion of *width* of the hypergraph Q [13, 15, 22, 26]. However, it is not clear if this is optimal. Even for the triangle join,

[4]The $\tilde{O}$ and $\tilde{\Omega}$ notation suppresses polylog factors.

[5]A sub-class of α-acyclic joins.

[6]The LW join algorithm presented in [23] has a mistake, but it can be fixed, although non-trivially.

it is not known what the output-optimal bound is. For the triangle join, any notion of width has $w \geq 1.5$, thus these algorithms are no better than the worst-case optimal algorithm, which has running time $O(\text{IN}^{1.5})$. Recently, an improved triangle algorithm has been developed with a running time of $O(\text{IN}^{1.408} + \text{IN}^{1.222}\text{OUT}^{0.186})$ [10], which is better than the worst-case optimal algorithm when $\text{OUT} < \text{IN}^{1.495}$. On the lower bound side, it is known that when $\text{OUT} \geq \text{IN}$, at least $\Omega(\text{IN}^{4/3-o(1)})$ time is needed, assuming the 3SUM conjecture [30]. Thus, output-optimal algorithms for cyclic joins still remain a wide open problem.

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.

Multi-numbering [16]: Given IN (key, value) pairs, for each key, assigns consecutive numbers $1, 2, 3, \dots$ to all the pairs with the same key.

Sum-by-key [16]: Given IN (key, value) pairs, compute the sum of values for each key, where the sum is defined by any associative operator.

Multi-search [16]: 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 semijoin.

Note that we can remove all dangling tuples in an acyclic-join [33] by a constant number of semi-joins, so it can be done in $O(1)$ rounds with linear load.

Parallel-packing: 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 $m \leq 1 + 2\sum_i x_i$.

We are not aware of an explicit reference on this primitive, but it can be solved quite easily, as shown in the full version of the paper [18].

Computing the output size OUT **of an acyclic join:** This primitive is a special case of our join-aggregate algorithm, which will be described in Section 6.

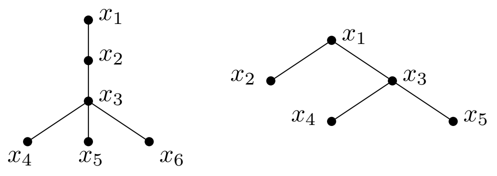

Figure 2: Examples of tall-flat and hierarchical join.

3 R-HIERARCHICAL JOINS

Recall that in a hierarchical join, all attributes can be organized into a forest, such that x is a descendant of y if and only if $\mathcal{E}_x \subseteq \mathcal{E}_y$. Each $e \in \mathcal{E}$ corresponds to a node x in the forest, such that e contains precisely x and all its ancestors. A subclass of hierarchical joins are tall-flat joins. 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. For example, $Q_1 = R_1(x_1) \bowtie R_2(x_1, x_2) \bowtie R_3(x_1, x_2, x_3) \bowtie R_4(x_1, x_2, x_3, x_4) \bowtie R_5(x_1, x_2, x_3, x_5) \bowtie R_6(x_1, x_2, x_3, x_6)$ is a tall-flat join; $Q_2 = R_1(x_1, x_2) \bowtie R_2(x_1, x_3, x_4) \bowtie R_3(x_1, x_3, x_5)$ is a hierarchical join (but not tall-flat). Their attribute forests (actually, trees for these two cases) are shown in Figure 2.

In this section, we study r-hierarchical joins. A join is r-hierarchical if its reduced join is hierarchical. For example, $Q_3 = Q_2 \bowtie R_4(x_3, x_5) \bowtie R_5(x_5)$ is an r-hierarchical join (but not hierarchical). After an r-hierarchical join is reduced, its hyperedges correspond to the leaves of the attribute forest.

3.1 BinHC algorithm revisited

We mentioned above that the HyperCube algorithm [2] is an instance-optimal algorithm for computing Cartesian products. The BinHC algorithm [7] is a generalization of the HyperCube algorithm to general joins. For a join Q, denote the residual query by removing attributes $\mathbf{x} \subseteq \mathcal{V}$ as $Q_{\mathbf{x}}$. Let $\mathbf{u}$ be any fractional edge packing of $Q_{\mathbf{x}}$ that saturates the attributes $\mathbf{x}$, i.e., $\sum_{e:x\in e} \mathbf{u}(e) \geq 1$ for every $x \in \mathbf{x}$, and $\sum_{e:x\in e} \mathbf{u}(e) \leq 1$ for every $x \in \mathcal{V} - \mathbf{x}$. Assuming knowing all degree information in advance, this algorithm computes Q on instance $\mathcal{R}$ in a single round with a load of $\widetilde{O}(\frac{\text{IN}}{p} + L_{\text{BinHC}}(p, \mathcal{R}))$, where

$$L_{\text{BinHC}}(p, \mathcal{R}) := \max_{\mathbf{x},\mathbf{u}} \left(\frac{\sum_{\mathbf{a}\in\text{dom}(\mathbf{x})} \prod_{e\in\mathcal{E}} |\sigma_{\mathbf{x}=\mathbf{a}} R(e)|^{\mathbf{u}(e)}}{p} \right)^{\frac{1}{\sum_{e\in\mathcal{E}} \mathbf{u}(e)}}$$

Here we define $0^0 = 0$. Note that for any $e \subseteq \mathbf{x}$, $|\sigma_{\mathbf{x}=\mathbf{a}} R(e)|$ is either 0 or 1, so we can just set $\mathbf{u}(e) = 0$ for each such e in the definition above.

We prove the following results and the proofs can be found in the full version of the paper [18].

Theorem 3.1. *On any tall-flat join and any instance $\mathcal{R}$, $L_{BinHC}(p, \mathcal{R}) = O(L_{instance}(p, \mathcal{R}))$.*

THEOREM 3.2. *On any r-hierarchical join Q and instance $\mathcal{R}$ without dangling tuples, $L_{BinHC}(p, \mathcal{R}) = O(L_{instance}(p, \mathcal{R}))$.*

Note that since $L_{\text{instance}}(p, \mathcal{R})$ is a per-instance lower bound even for multi-round algorithms, this means that the BinHC algorithm is instance-optimal even among all multi-round algorithms, up to polylogarithmic factors. This result also incorporates the instance-optimality of the HyperCube algorithm on Cartesian products, which are special r-hierarchical joins without dangling tuples.

Remark. Koutris and Suciu [25] show that non-tall-flat joins cannot be done with load $\tilde{O}(\frac{\text{IN}}{p} + \frac{\text{OUT}}{p})$ by one-round algorithms. This does not contradict Theorem 3.2 since their lower bound construction uses dangling tuples. Our result implies that the key barrier for one-round algorithms is actually the dangling tuples. If they do not exist, one-round algorithms can go beyond tall-flat joins and solve r-hierarchical joins instance-optimally, up to polylog factors. On the other hand, once $O(1)$ rounds are allowed, dangling tuples become irrelevant, since they can be removed with linear load and $O(1)$ rounds.

3.2 An instance-optimal algorithm

We have shown that the BinHC algorithm is an instance-optimal algorithm for r-hierarchical joins, but it has an instance-optimality ratio of $\log^{O(1)} p$, where the $O(1)$ exponent depends on the query size, and is at least m, the number of relations. In this section, we improve the optimality ratio to $O(1)$, i.e., achieving a load of $O(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R}))$. Our algorithm uses $O(1)$ rounds, but note that BinHC also needs $O(1)$ rounds to remove the dangling tuples if they exist. Furthermore, our algorithm is deterministic while BinHC is randomized.

As a preprocessing step, we remove all dangling tuples. Then we reduce the join hypergraph, since if $e \subseteq e'$, $R(e)$ will not affect the final join results after dangling tuples are removed[7]. Thus, we are left with a hierarchical join Q on an instance $\mathcal{R}$ with no dangling tuples.

Let $\mathcal{T}$ be the attribute forest of Q. Recall that after the join is reduced, each relation corresponds to a leaf of $\mathcal{T}$, whose attributes are precisely the leaf's ancestors in $\mathcal{T}$. Our algorithm is recursive. We will show that the load of this algorithm is $O(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R}))$ for any hierarchical join Q on any instance $\mathcal{R}$. To simplify notation, we will not derive the exact constant in the big-Oh, which depends (exponentially) on the recursion depth. Since the recursion depth is proportional to (actually, twice) the height of $\mathcal{T}$, which is a constant, this is not a concern. Similarly, the number of servers employed by the algorithm will be $O(p)$, where the hidden constant may also depend on the recursion depth.

[7]Strictly speaking, this violates the tuple-based requirement that when emitting a join result, all the participating tuples must be present. This can be easily fixed. Before removing $R(e)$, we attach each tuple $t \in R(e)$ to all tuples in $R(e')$ that join with t. This can be done by the multi-search primitive with linear load.

The base case is when Q has just one relation. In this case the algorithm just emits all tuples in the relation, achieving the bound $O(\frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R}))$ trivially.

For a general hierarchical join Q and an instance $\mathcal{R}$, we proceed as follows. We first compute $L_{\text{instance}}(p, \mathcal{R})$: We use p servers to compute $| \bowtie_{e \in S} R(e)|$ for each $S \subseteq \mathcal{E}$ (recall that computing the output size of an acyclic join is an MPC primitive). This requires $O(p)$ servers with load $O(\frac{\text{IN}}{p})$. Note that $Q(\mathcal{R}, S) = \bowtie_{e \in S} R(e)$ when there is no dangling tuples in $\mathcal{R}$, so we can compute $L_{\text{instance}}(p, \mathcal{R})$ as defined in (2). Setting $L = \frac{\text{IN}}{p} + L_{\text{instance}}(p, \mathcal{R})$, we will show below how to compute the join with $O(p)$ servers and load $O(L)$.

Let k be the number of trees in $\mathcal{T}$. We handle the following two cases using different recursive strategies:

Case (1): $k = 1$. In this case, $\mathcal{T}$ is a tree. Suppose the root attribute of $\mathcal{T}$ is x, which is included in all the relations. Consider every $a \in \text{dom}(x)$, and let $\mathcal{R}_a = \{\sigma_{x=a} R(e) : e \in \mathcal{E}\}$. It suffices to compute the residual query Q_x on each $\mathcal{R}_a$, but all the $Q_x(\mathcal{R}_a)$'s have to be computed in parallel, using $O(p)$ servers in total. Thus, the key is to allocate servers to these residual queries appropriately so as to ensure a uniform load of $O(L)$. To do so, we first compute IN_a, the input size of $\mathcal{R}_a$, for all $a \in \text{dom}(x)$. Since $\text{IN}_a = \sum_{e \in \mathcal{E}} |\sigma_{x=a} R(e)|$, and each tuple belongs to exactly one $\mathcal{R}_a$, this is a sum-by-key problem, i.e., each tuple t with $\pi_x t = a$ has key a and weight 1. Note that $\text{IN} = \sum_a \text{IN}_a$.

An instance $\mathcal{R}_a$ is *heavy* if $\text{IN}_a > L$ and *light* otherwise. We handle heavy and light instances in different ways.

Case (1.1): Light instances. We use the parallel-packing primitive to put the light instances into $O(\frac{\text{IN}}{L}) = O(p)$ groups with each group having total input size $O(L)$. Then we simply use one server to solve the instances in each group. The load of each server is $O(L)$.

Case (1.2): Heavy instances. By definition, there are at most $\frac{\text{IN}}{L} = O(p)$ heavy instances. For each heavy instance $\mathcal{R}_a$, we allocate $p_a = \lceil p \cdot \frac{\text{IN}_a}{\text{IN}} \rceil$ servers to compute in parallel the join size $|Q_x(\mathcal{R}_a, S)|$ for all $a \in \text{dom}(x)$ and all $S \subseteq \mathcal{E}$. This uses $O(p)$ servers, and the load is $O(\max_a \frac{\text{IN}_a}{p_a}) = O(\frac{\text{IN}}{p})$. Next, for each heavy instance $\mathcal{R}_a$, we allocate

$$p_a = \max_{S \subseteq \mathcal{E}} \frac{|Q_x(\mathcal{R}_a, S)|}{L^{|S|}}$$

servers and compute $Q_x(\mathcal{R}_a)$ recursively in parallel. The number of servers used is

$$\sum_a p_a \le \sum_a \sum_{S \subseteq \mathcal{E}} \frac{|Q_x(\mathcal{R}_a, S)|}{L^{|S|}} = \sum_{S \subseteq \mathcal{E}} \frac{|Q(\mathcal{R}, S)|}{L^{|S|}} = O(p).$$

By the induction hypothesis, computing $Q_x(\mathcal{R}_a)$ with p_a servers has a load of (the big-Oh of)

$$\frac{\mathrm{IN}_a}{p_a} + L_{\text{instance}}(p_a, \mathcal{R}_a) = \frac{\mathrm{IN}_a}{p_a} + \max_{S \subseteq \mathcal{E}} \left(\frac{|Q_x(\mathcal{R}_a, S)|}{p_a} \right)^{\frac{1}{|S|}}. \quad (3)$$

We bound each term of (3): For a heavy instance $\mathcal{R}_a$, there must exist at least one $e \in \mathcal{E}$ such that $|\sigma_{x=a}R(e)| \geq \frac{1}{|\mathcal{E}|} \cdot \mathrm{IN}_a$. Furthermore, since there are no dangling tuples, every tuple in $\sigma_{x=a}R(e)$ must be part of a join result of $Q_x(\mathcal{R}_a)$, so $|\sigma_{x=a}R(e)| = |Q_x(\mathcal{R}_a, \{e\})|$. Taking $S = \{e\}$, we have

$$p_a \geq \frac{1}{L} \cdot |Q_x(\mathcal{R}_a, \{e\})| = \frac{1}{L} \cdot |\sigma_{x=a}R(e)| \geq \frac{\mathrm{IN}_a}{|\mathcal{E}| \cdot L},$$

so $\frac{\mathrm{IN}_a}{p_a} = O(L)$. The second term of (3) is also bounded by L simply by the definition of p_a.

Case (2): $k > 1$. In this case, the join becomes a Cartesian product $Q_1(\mathcal{R}_1) \times \cdots \times Q_k(\mathcal{R}_k)$, where each $Q_i(\mathcal{R}_i)$ is a join under Case (1). One would attempt to first compute each $Q_i(\mathcal{R}_i)$ recursively, and then compute the Cartesian product, but this would not yield instance-optimality. Just consider an instance with $|Q_1(\mathcal{R}_1)| = 1$ and $|Q_2(\mathcal{R}_2)| = p \cdot \mathrm{IN}$, where $Q_2(\mathcal{R}_2) = R_1(A, B) \bowtie R_2(B, C)$ with $|\mathrm{dom}(B)| = 1, |R_1| = \mathrm{IN}, |R_2| = p$. On this instance, we have $L_{\text{instance}}(p, \mathcal{R}) = \max(\frac{\mathrm{IN}}{p}, \sqrt{\mathrm{IN}})$, but if we took a two-step approach, merely storing the intermediate result $Q_2(\mathcal{R}_2)$ would incur a load of $\Omega(\mathrm{IN})$. This means that we have to interleave the two steps so as to avoid storing the intermediate results $Q_i(R_i)$ explicitly.

We arrange servers into a $p_1 \times p_2 \times \cdots \times p_k$ hypercube, where the dimensions $p_1, p_2, \cdots, p_k$ will be determined later. We identify each server with coordinates $(c_1, c_2, \cdots, c_k)$, where $c_i \in [p_i]$. 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 $Q_i(\mathcal{R}_i)$ (using the algorithm under Case (1)). Yes, each $Q_i(\mathcal{R}_i)$ 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 $Q_i(\mathcal{R}_i), i = 1, \ldots, k$. For each $Q_i(\mathcal{R}_i)$, it emits a subset of its join results, denoted $Q_i(\mathcal{R}_i, c_1 \ldots, c_k)$. Then the server emits the Cartesian product $Q_1(\mathcal{R}_1, c_1 \ldots, c_k) \times \cdots \times Q_k(\mathcal{R}_k, c_1 \ldots, c_k)$. Note that for each group of servers computing $Q_i(\mathcal{R}_i)$, the p_i servers in the group emit $Q_i(R_i)$ with no redundancy, so there is no redundancy in emitting the Cartesian product.

It remains to show how to set $p_1, \ldots, p_k$ so that $p_1 \cdots p_k = O(p)$ and each server has a load of $O(L)$. To do so, we first compute IN_i, the input size of $\mathcal{R}_i$, in the same way as in Case (1). An instance $\mathcal{R}_i$ is *heavy* if $\mathrm{IN}_i > L$ and *light* otherwise. For each heavy instance $\mathcal{R}_i$, we use p servers to compute $|\bowtie_{e \in S} \mathcal{R}_i(e)| = |Q_i(R_i, S)|$ for all $S \subseteq \mathcal{E}_i$, where $\mathcal{E}_i$ is the set of edges in Q_i. This requires $O(p)$ servers with load $O(\frac{\mathrm{IN}}{p})$. Then if $\mathcal{R}_i$ is light, we set $p_i = 1$; otherwise set

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

Let $I = \{i \mid \mathcal{R}_i \text{ is heavy}\}$. The number of servers used is

$$\prod_{i \in I} p_i \leq \prod_{i \in I} \sum_{S \subseteq \mathcal{E}_i} \left(\frac{|Q_i(\mathcal{R}_i, S_i)|}{L^{|S|}} + 1 \right) \leq \sum_{S \subseteq \bigcup_{i \in I} \mathcal{E}_i} \frac{|Q(\mathcal{R}, S)|}{L^{|S|}},$$

which is bounded by $O(p)$.

Finally, consider the load of each server, which serves to compute each $Q_i(\mathcal{R}_i)$ with a group of p_i servers. For a light $\mathcal{R}_i$, $p_i = 1$ and it imposes a load of $O(L)$. For a heavy $\mathcal{R}_i$, by the induction hypothesis, the load is (the big-Oh of)

$$\frac{\mathrm{IN}_i}{p_i} + L_{\text{instance}}(p_i, \mathcal{R}_i) = \frac{\mathrm{IN}_i}{p_i} + \max_{S \subseteq \mathcal{E}_i} \left(\frac{|Q_i(\mathcal{R}_i, S)|}{p_i} \right)^{\frac{1}{|S|}}.$$

This can be bounded by $O(L)$ using the same argument as Case (1.2). Summing over all $i = 1, \ldots, k$ increases the load by just a $k = O(1)$ factor.

The induction proof thus completes and we obtain the following result.

THEOREM 3.3. *On any r-hierarchical join query Q and any instance $\mathcal{R}$, there is an algorithm computing $Q(\mathcal{R})$ in $O(1)$ rounds with load $O(\frac{\mathrm{IN}}{p} + L_{instance}(p, \mathcal{R}))$.*

Since an instance-optimal algorithm is also output-optimal, we also obtain an output-optimal algorithm for r-hierarchical joins. In fact, we derive a closed-form formula of the output-optimal bound, described in Theorem 3.4. Moreover, we give a cleaner but not tight output-sensitive bound in Corollary 3.5. In particular, it will be used in the analysis of the output-sensitive algorithm for arbitrary acyclic joins in Section 5.1. Their proofs are shown in the full version of the paper [18].

THEOREM 3.4. *There is an algorithm that computes any r-hierarchical join with load $O\left(\frac{\mathrm{IN}}{p^{1/\max\{1, k^*-1\}}} + (\frac{\mathrm{OUT}}{p})^{\frac{1}{k^*}}\right)$ in $O(1)$ rounds, where $k^* = \lceil \log_{\mathrm{IN}} \mathrm{OUT} \rceil$. This bound is output-optimal.*

COROLLARY 3.5. *There is an algorithm that computes any r-hierarchical join in $O(1)$ rounds with load $O(\frac{\mathrm{IN}}{p} + \sqrt{\frac{\mathrm{OUT}}{p}})$.*

4 LINE-3 JOIN

The simplest acyclic but not r-hierarchical join is the line-3 join $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$. In this section, we give an output-optimal MPC algorithm with load $O(\frac{\mathrm{IN}}{p} + \frac{\sqrt{\mathrm{IN} \cdot \mathrm{OUT}}}{p})$, together with a matching lower bound. In particular, the lower bound implies that instance-optimal algorithms are not possible for the line-3 join. In Section 5, we extend these results to arbitrary acyclic joins.

4.1 The Yannakakis algorithm revisited

The Yannakakis algorithm first removes all the dangling tuples, which is just a series of semi-joins and can be done with load $O(\frac{\text{IN}}{p})$. Then the algorithm performs pairwise joins in some arbitrary order. In the RAM model, 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. In fact, this argument applies on a per-instance basis, and the Yannakakis algorithm is instance-optimal on any instance with any join order.

Interestingly, the join order does matter in the MPC model. Consider the following instance of the line-3 join (see the top half of Figure 3). Attributes A, B, C, D have domain sizes $\frac{\text{OUT}}{N}, \frac{N^2}{\text{OUT}}, N, 1$, respectively. Set $R_1(A, B) = \text{dom}(A) \times \text{dom}(B)$, $R_2(B, C)$ is a one-to-many relation from $\text{dom}(B)$ to $\text{dom}(C)$, and $R_3(C, D) = \text{dom}(C) \times \text{dom}(D)$. Note that this instance has $\text{IN} = \Theta(N)$ and the output size is exactly OUT. Consider first the join plan $(R_1 \bowtie R_2) \bowtie R_3$, and note that $|R_1 \bowtie R_2| = |R_1 \bowtie R_2 \bowtie R_3| = \text{OUT}$. Using the $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$-load algorithm [7, 16] for binary joins, the load of computing $R_1 \bowtie R_2$ is $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$. However, since the output of the first join is the input of the second join, the input size for the second join is OUT, so the load of the second join is $O(\frac{\text{OUT}}{p} + \sqrt{\frac{\text{OUT}}{p}}) = O(\frac{\text{OUT}}{p})$. In general, the intermediate join result can be as large as $O(\text{OUT})$, which is why the Yannakakis algorithm incurs a load of $O(\frac{\text{OUT}}{p})$ (after dangling tuples are removed) on an acyclic join, as observed in [1, 24].

Now consider the alternative plan $R_1 \bowtie (R_2 \bowtie R_3)$. Note that $|R_2 \bowtie R_3| = O(\text{IN})$, so the load of computing $R_2 \bowtie R_3$ is $O(\frac{\text{IN}}{p})$, while the load of computing the second join is $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$. Crucially, the reason why the second plan is better is that it has a smaller intermediate join size. Note that a smaller intermediate join size does not matter in the RAM model, where the total cost is always dominated by the last join. But it does matter in the MPC model, because of the $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$ load complexity of a binary join, which has a linear dependency on the input size but sublinear in the output size. Fundamentally, this is because the MPC model is all about *locality*: algorithms strive to send all "related" tuples to the same server so as to maximize the number of join results that can be found by the server locally.

Now, the key question is if there is always a join plan with an intermediate join size asymptotically smaller than $O(\text{OUT})$. Unfortunately, the answer is no. A bad example can be easily constructed, by just putting two of the above instances together, but in opposite directions (see Figure 3).

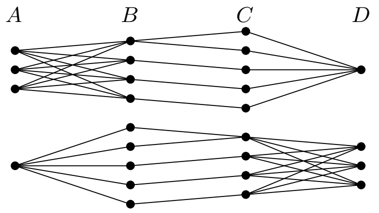

Figure 3: A hard instance for the Yannakakis algorithm.

Nevertheless, this bad example precisely points us to the right direction: Although a global best join order may not exist, but if we decompose the join into multiple pieces, it is possible to find a provably good join order for each. This is exactly the basic idea of our algorithm, presented next.

4.2 A new algorithm for the line-3 join

We first compute OUT (an MPC primitive). Then we proceed in two steps:

Step (1): Computing degrees. For a value in attribute B, it is *heavy* if its *degree* in relation R_1, i.e., $|\sigma_{B=b} R_1|$, is greater than τ (value to be determined later), otherwise *light*. We first use the sum-by-key primitive to compute the degrees of all b's for $b \in \text{dom}(B)$. After classifying the values in $\text{dom}(B)$ as heavy and light, we divide tuples in R_1 and R_2 also into heavy tuples and light tuples, depending on their B value. More precisely, a tuple in R_1 or R_2 is heavy if its B value is heavy, and light otherwise. This can be done by the multi-search primitive. We denote the heavy (resp. light) tuples in R_i as R_i^H (resp. R_i^L), for $i = 1, 2$.

Step (2): Decomposing the join. We decompose the join into the following two parts, and compute them using different join orders:

$$Q_1 = R_1^H \bowtie (R_2^H \bowtie R_3),$$
$$Q_2 = (R_1^L \bowtie R_2^L) \bowtie R_3.$$

Note that since R_1 and R_2 are both divided according to the B attribute, R_1^H do not join with R_2^L, R_1^L do not join with R_2^H.

Now we analyze the load. For Q_1, the intermediate join $R_{23} = R_2^H \bowtie R_3$ has size bounded by $\frac{\text{OUT}}{\tau}$, since each intermediate join result from R_{23} has a heavy B value, so it joins with at least τ tuples in R_1. Thus, the load of computing Q_1 is $O(\frac{\text{IN}}{p} + \frac{\text{OUT}}{p\tau} + \sqrt{\frac{\text{OUT}}{p}})$.

For Q_2, the intermediate join $R_{12} = R_1^L \bowtie R_2^L$ has size bounded by $\text{IN} \cdot \tau$, since each light tuple from R_2 can join with at most τ tuples from R_1. Thus, the load of computing Q_2 is $O(\frac{\text{IN}}{p} + \frac{\text{IN} \cdot \tau}{p} + \sqrt{\frac{\text{OUT}}{p}})$.

Setting $\tau = \sqrt{\frac{\text{OUT}}{\text{IN}}}$ balances the second term in both cases, and we obtain the claimed result (note that $\sqrt{\frac{\text{OUT}}{p}} \le \frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p}$ for $\text{IN} \ge p$):

Theorem 4.1. *There is an algorithm computing the line-3 join in* $O(1)$ *rounds with load* $O\left(\frac{\text{IN}}{p} + \frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p}\right)$.

4.3 Lower bound

In the full version [18], we prove the following lower bound on any tuple-based algorithm for computing the line-3 join.

Theorem 4.2. *For any* $\text{OUT} \ge \text{IN}$, *there exists an instance* $\mathcal{R}$ *for the line-3 join with input size* $\Theta(\text{IN})$ *and output size* $\Theta(\text{OUT})$, *such that any tuple-based algorithm computing the join in* $O(1)$ *rounds must have a load of* $\Omega\left(\min\left\{\frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p\cdot\log\text{IN}}, \frac{\text{IN}}{\sqrt{p}}\right\}\right)$.

Ignoring logarithmic factors, this lower bound completes our understanding of the line-3 join in terms of output-optimality: (1) When $\text{OUT} \le \text{IN}$, the Yannakakis algorithm has linear load $O\left(\frac{\text{IN}}{p}\right)$. (2) When $\text{IN} < \text{OUT} \le p \cdot \text{IN}$, the lower bound becomes $\tilde{\Omega}\left(\frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p}\right)$, which is matched by our new algorithm. (3) When $\text{OUT} \ge p \cdot \text{IN}$, the lower bound is $\Omega\left(\frac{\text{IN}}{\sqrt{p}}\right)$, which is matched by the worst-case optimal algorithm in [17, 23]. 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 any acyclic join always grows linearly with OUT.

An easy corollary (proof in the full version [18]) is the following result, which shows that instance-optimality is not achievable for the line-3 join.

Corollary 4.3. *For any* $\text{IN} \ge p^{3/2}$, *there is an instance* $\mathcal{R}$ *with input size* $\Theta(\text{IN})$ *for the line-3 join, such that any tuple-based algorithm computing the join in* $O(1)$ *rounds must have a load of* $\Omega(\frac{\text{IN}}{p^{1/2}\log\text{IN}})$, *while* $L_{instance}(p, \mathcal{R}) = O(\frac{\text{IN}}{p})$.

5 ACYCLIC JOINS

In this section, we first extend the results from the previous section to arbitrary acyclic joins. Specifically, the algorithm is a (nontrivial) generalization of the line-3 algorithm, but it is self-contained; the lower bound builds on top of the hard instance of the line-3 join.

5.1 Algorithm

As preprocessing, we remove all dangling tuples. Assume the output size OUT has been computed (an MPC primitive).

Recall that in an acyclic join $Q = (\mathcal{V}, \mathcal{E})$, the hyperedges $\mathcal{E}$ can be organized into a join tree $\mathcal{T}$, such that for each attribute $x \in \mathcal{V}$, the nodes corresponding to $\mathcal{E}_x$ are connected

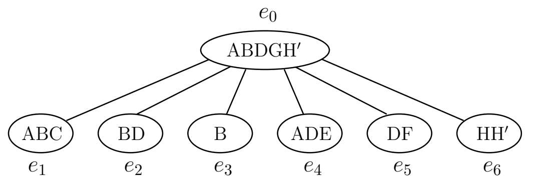

Figure 4: A node e_0 in the join tree $\mathcal{T}$ and its leaf children $e_1, e_2, e_3, e_4, e_5, e_6$.

in $\mathcal{T}$. Given such a join tree $\mathcal{T}$, our algorithm recursively decomposes the join into multiple pieces, and apply a different join strategy for each.

We start from an internal node of $\mathcal{T}$ whose children are all leaves. Let this node be e_0, which has k leaf children $e_1, \cdots, e_k$ (see Figure 4 for an example). Let $s_i = e_0 \cap e_i$ be the set of join attributes between e_0 and e_i. We will assume $s_i \ne \emptyset$; otherwise we can add a dummy attribute to both e_0 and e_i and all tuples in $R(e_0)$ and $R(e_i)$ share the same value on this dummy attribute (e.g., we add a dummy attribute H' to both e_0 and e_6 in Figure 4). Note that the join tree ensures the property that if $x \in e_i \cap e_j$ for $i \ne j$, then $x \in e_0$.

Let $N_\alpha = \sum_{i=1}^{k} |R(e_i)|$ and $N_\beta = \text{IN} - N_\alpha$. We will actually prove a slightly tighter bound, that the load of our algorithm is bounded by $O(\frac{\text{IN}}{p} + \frac{\sqrt{N_\beta\cdot\text{OUT}}}{p} + \sqrt{\frac{\text{OUT}}{p}})$.

Set $\tau = \sqrt{\frac{\text{OUT}}{N_\beta}}$. Our algorithm proceeds in three steps.

Step (1): Computing data statistics. In each relation $R(e_i)$, $i = 1, \ldots, k$, let v be an assignment of values for attributes s_i. The set of *heavy* assignments in $R(e_i)$ is

$$H(s_i, e_i) = \{v \in \pi_{s_i} R(e_i) : |\sigma_{s_i=v} R(e_i)| \ge \tau\}.$$

Tuples in $R(e_i)$ can also be identified as *heavy* or *light*, depending on their projection on attributes s_i. More precisely, a tuple $t \in R(e_i)$ is heavy if $\pi_{s_i} t \in H(s_i, e_i)$. The set of heavy tuples and light tuples in $R(e_i)$ are denoted as $R_H(e_i)$ and $R_L(e_i)$, respectively. All the statistics can be computed in by the sum-by-key and multi-search primitives with linear load.

Let $\bar{\mathcal{E}} = \mathcal{E} - \{e_0, e_1, \cdots, e_k\}$. We decompose the join into the following sub-joins:

$$R(e_0) \bowtie R_?(e_1) \bowtie \cdots \bowtie R_?(e_k) \bowtie \left(\bowtie_{e\in\bar{\mathcal{E}}} R(e)\right),$$

where each ? can be either H or L. Note that there are 2^k, which is a constant, sub-joins, so we can afford to use p servers for each sub-join. If a sub-join involves at least one $R_H(e_i)$, we apply the procedure in step (2) to it. In step (3), we handle the case where all ? are L.

Step (2): Sub-joins with at least one $R_H(e_i)$. Without loss of generality, suppose $R_H(e_1)$ is in the sub-join, i.e., we need to compute the sub-join

$$R(e_0) \bowtie R_H(e_1) \bowtie R_?(e_2) \bowtie \cdots \bowtie R_?(e_k) \bowtie \left(\bowtie_{e\in\bar{\mathcal{E}}} R(e)\right),$$

where each ? can be either H or L. The algorithm consists of three steps:

(2.1) Compute $R'(e_0) = R(e_0) \ltimes R_H(e_1)$.
(2.2) Compute $R' = R'(e_0) \bowtie R_?(e_2) \bowtie \cdots \bowtie R_?(e_k) \bowtie \left(\bowtie_{e\in\bar{\mathcal{E}}} R(e)\right)$ by any order.
(2.3) Compute $R_H(e_1) \bowtie R'$.

We analyze the load in each step: (2.1) is a primitive operation that incurs linear load. To bound the load of (2.2), observe that $|R'| \le \frac{\text{OUT}}{\tau}$, since each tuple in R' joins with at least τ tuples in $R_H(e_1)$, each producing one final join result. Thus, the load is bounded by $O(\frac{\text{IN}}{p} + \frac{\text{OUT}}{p\cdot\tau})$. The binary join in (2.3) has input size $\frac{\text{OUT}}{\tau} + \text{IN}$ and output size OUT, incurring a load of $O(\frac{\text{IN}}{p} + \frac{\text{OUT}}{p\cdot\tau} + \sqrt{\frac{\text{OUT}}{p}})$, which dominates the first two steps. Plugging in the value of τ, the total load is bounded by $O(\frac{\text{IN}}{p} + \frac{\sqrt{N_\beta\cdot\text{OUT}}}{p} + \sqrt{\frac{\text{OUT}}{p}})$, as desired.

Step (3): The sub-join with all $R_L(e_i)$. It remains to compute the following sub-join:

$$R(e_0) \bowtie R_L(e_1) \bowtie \cdots \bowtie R_L(e_k) \bowtie \left(\bowtie_{e\in\bar{\mathcal{E}}} R(e)\right).$$

We further divide $R(e_0)$ into heavy and light tuples, as follows. Let $s = s_1 \cup s_2 \cup \cdots \cup s_k$, and let v be an assignment over attributes s. The set of *heavy* assignments in $R(e_0)$ is define as

$$H(s, e_0) = \{v \in \pi_s R(e_0) : \prod_{i=1}^{k} |\sigma_{s_i=\pi_{s_i} v} R_L(e_i)| \ge \tau\}.$$

Tuples in $R(e_0)$ are classified as *heavy* or *light*, depending on their projection on attributes s, i.e., a tuple $t \in R(e_0)$ is heavy if $\pi_s t \in H(s, e_0)$, and light otherwise. Similarly, denote the heavy and light tuples in $R(e_0)$ as $R_H(e_0)$ and $R_L(e_0)$, respectively.

These statistics can also be computed using the primitives, but with some more care. For each relation $R_L(e_i)$, we first use sum-by-key to compute $|\sigma_{s_i=v_i} R_L(e_i)|$ for every $v_i \in \pi_{s_i} R_L(e_i)$. This gives us a list of $(v_i, |\sigma_{s_i=v_i} R_L(e_i)|)$ pairs. Then, we use multi-search to find, for each tuple $t \in R(e_0)$, the up to k pairs $(v_i, |\sigma_{s_i=v_i} R_L(e_i)|)$ such that $\sigma_{s_i} t = v_i$. After this step, each tuple $t \in R(e_0)$ is attached with k values, and we multiply them together to decide if t is heavy or light.

Step (3.1): The sub-join with $R_H(e_0)$. We first compute the following sub-join:

$$R_H(e_0) \bowtie R_L(e_1) \bowtie \cdots \bowtie R_L(e_k) \bowtie \left(\bowtie_{e\in\bar{\mathcal{E}}} R(e)\right).$$

The algorithm consists of three steps:

(3.1.1) Compute $R'(e_0) = R_H(e_0) \bowtie \left(\bowtie_{e\in\bar{\mathcal{E}}} R(e)\right)$ by any order.
(3.1.2) Compute $R'(e_i) = R_H(e_0) \bowtie R_L(e_i)$ for each $i = 1, \cdots, k$.
(3.1.3) Compute $R'(e_0) \bowtie R'(e_1) \bowtie \cdots \bowtie R'(e_k)$. Note that each of these relations contains all attributes in e_0, so it is a hierarchical join (it is actually tall-flat), so we can use the instance-optimal algorithm in Section 3 to compute this join.

Now we analyze the load of each step: First, observe that $|R'(e_0)| \le \frac{\text{OUT}}{\tau}$. This is because the projection of each tuple in $R'(e_0)$ on s is a heavy assignment, so it will produce at least τ join results after joining with the $R_L(e_i)$'s. Therefore, the load of computing the join in (3.1.1) is $O(\frac{\text{IN}}{p} + \frac{\text{OUT}}{p\cdot\tau})$. Each binary join in (3.1.2) has a load of $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$. Note that each join result $R'(e_i)$ has size bounded by $N_\beta \cdot \tau$, since any tuple in $R(e_0)$ can join with at most τ tuples in $R_L(e_i)$. Thus, the hierarchical join in (3.1.3) has input size $O(N_\beta \cdot \tau + \frac{\text{OUT}}{\tau})$ and output size OUT, so the instance-optimal algorithm has load $O(\frac{N_\beta\cdot\tau}{p} + \frac{\text{OUT}}{p\cdot\tau} + \sqrt{\frac{\text{OUT}}{p}})$ according to Corollary 3.5. All the loads are bounded by $O(\frac{\text{IN}}{p} + \frac{\sqrt{N_\beta\cdot\text{OUT}}}{p} + \sqrt{\frac{\text{OUT}}{p}})$, as desired.

Step (3.2): The sub-join with $R_L(e_0)$. Finally, we are left with the sub-join

$$R_L(e_0) \bowtie R_L(e_1) \bowtie \cdots \bowtie R_L(e_k) \bowtie \left(\bowtie_{e\in\bar{\mathcal{E}}} R(e)\right).$$

This is actually the only case where we need recursion:

(3.2.1) Compute $R'_L(e_0) = R_L(e_0) \bowtie R_L(e_1) \bowtie \cdots \bowtie R_L(e_k)$ by any order.
(3.2.2) If $\bar{\mathcal{E}} \ne \emptyset$, compute $R'_L(e_0) \bowtie \left(\bowtie_{e\in\bar{\mathcal{E}}} R(e)\right)$ recursively.

Now we analyze the load: First, we have $|R'_L(e_0)| \le N_\beta \cdot \tau$, since the projection of each tuple in $R_L(e_0)$ on s is a light assignment. Thus, the load of step (3.2.1) is $O(\frac{\text{IN}}{p} + \frac{N_\beta\cdot\tau}{p})$, which is also bounded by $O(\frac{\text{IN}}{p} + \frac{\sqrt{N_\beta\cdot\text{OUT}}}{p} + \sqrt{\frac{\text{OUT}}{p}})$. So far, we have completed the base case of the induction proof.

For the join to be computed recursively in step (3.2.2), its input size is at most $\text{IN} + N_\beta \cdot \tau$ and output size is at most OUT. More importantly, N_β can only become smaller, since e_0 becomes a leaf in the residual join and $|R(e_0)|$ is no longer included in N_β, no matter which node in the residual join is picked to be its new e_0. By the induction hypothesis, computing the residual join recursively incurs a load of $O(\frac{\text{IN}}{p} + \frac{N_\beta\cdot\tau}{p} + \frac{\sqrt{N_\beta\cdot\text{OUT}}}{p} + \sqrt{\frac{\text{OUT}}{p}})$, thus bounded by $O(\frac{\text{IN}}{p} + \frac{\sqrt{N_\beta\cdot\text{OUT}}}{p} + \sqrt{\frac{\text{OUT}}{p}})$.

Note that the recursion will increase the constant in the big-Oh, but as the recursion depth depends only on the query not the data size, it does not change the asymptotic result.

This completes the induction proof that the algorithm has a load of $O(\frac{\text{IN}}{p} + \frac{\sqrt{N_\beta\cdot\text{OUT}}}{p} + \sqrt{\frac{\text{OUT}}{p}})$. Observing that $N_\beta \le \text{IN}$ and $\sqrt{\frac{\text{OUT}}{p}} \le \frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p}$, we obtain the following result.

THEOREM 5.1. *There is an algorithm that computes any acyclic join in $O(1)$ rounds with load $O(\frac{\text{IN}}{p} + \frac{\sqrt{\text{IN}\cdot\text{OUT}}}{p})$.*

5.2 Lower bound

In Section 4.2 we have constructed a hard instance for the line-3 join and have shown that any algorithm must incur a load of $\Omega(\min\{\frac{\sqrt{\mathrm{IN}\cdot\mathrm{OUT}}}{p\cdot\log\mathrm{IN}}, \frac{\mathrm{IN}}{\sqrt{p}}\})$ on this instance. In this section, we generalize this lower bound to an arbitrary acyclic join that is not r-hierarchical. Note that for r-hierarchical joins, we can achieve a smaller load $O(\frac{\mathrm{IN}}{p} + \sqrt{\frac{\mathrm{OUT}}{p}})$ (see the full version [18]), so this establishes a separation between r-hierarchical joins and acyclic joins.

The basic idea in the lower bound is that any acyclic join must "include" a line-3 join, such that any algorithm computing the acyclic join must also compute the line-3 join. This is more formally captured by the following structural lemma on acyclic and r-hierarchical joins. To state the lemma, we need some terminology. In a hypergraph $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$, a *path* between $x, y \in \mathcal{V}$, denoted $P(x, y)$, is a sequence of vertices starting with x and ending with y, such that each consecutive pair of vertices appear together in an edge. The length of a path is defined as the number of vertices in $P(x, y)$ minus 1. A path $P(x, y)$ is *minimal* if there is no other path $P'(x, y)$ that is a strict subsequence of $P(x, y)$. It is easy to see that $P(x_1, x_k) = (x_1, x_2, \cdots, x_k)$ is minimal if and only if there exists no edge $e \in \mathcal{E}$ containing x_i and x_j with $|j - i| > 1$. Note that a shortest path must be minimal, but not vice versa.

Lemma 5.2. *An acyclic join query is not r-hierarchical if and only if it has a minimal path of length* 3.

The proof is given in the full version [18]. With this lemma, we can extend the hard instance for the line-3 join to any acyclic but non-r-hierarchical join $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$.

Let (x_1, x_2, x_3, x_4) be a minimal path of length 3 in $\mathcal{Q}$, and suppose $\{x_1, x_2\} \subseteq e_1$, $\{x_2, x_3\} \subseteq e_2$, $\{x_3, x_4\} \subseteq e_3$. Let $\mathcal{R} = \{R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4)\}$ be the hard instance for the line-3 join. We construct the hard instance $\mathcal{R}' = \{R'(e) : e \in \mathcal{E}\}$ for $\mathcal{Q}$ as follows. The domain of $x_i, i = 1, 2, 3, 4$ is the same as in $\mathcal{R}$. For any other attribute y, set $|\mathrm{dom}(y)| = 1$.

Since the path is minimal, each $e \in \mathcal{E}$ must fall into one of the following three cases:

(1) For any e with $e \cap \{x_1, x_2, x_3, x_4\} = \emptyset$, $R'(e)$ contains only one tuple connecting the only value in the domains of attributes in e.
(2) If $e \cap \{x_1, x_2, x_3, x_4\} = \{x_i\}, i = 1, 2, 3, 4$, then $R'(e)$ contains $|\mathrm{dom}(x_i)|$ tuples, each having a distinct value of $\mathrm{dom}(x_i)$.
(3) If $e \cap \{x_1, x_2, x_3, x_4\} = \{x_i, x_{i+1}\}, i = 1, 2, 3$, then $R'(e)$ contains $|R_i(x_i, x_{i+1})|$ tuples such that $\pi_{x_i, x_{i+1}} R'(e) = R_i(x_i, x_{i+1})$.

It can be easily verified that $\mathcal{Q}(\mathcal{R}')$ is exactly the join results of the line-3 join on $\mathcal{R}$, so the same lower bound applies. However, since the output size of the line-3 join is at most IN^2, we do have a condition on OUT:

Theorem 5.3. *For an acyclic but non-r-hierarchical join and any* $\mathrm{IN} \geq p^{3/2}, \mathrm{OUT} \leq \mathrm{IN}^2$*, 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* $\Omega(\min\{\frac{\sqrt{\mathrm{IN}\cdot\mathrm{OUT}}}{p\cdot\log\mathrm{IN}}, \frac{\mathrm{IN}}{\sqrt{p}}\})$.

Similar to the line-3 join, this lower bound shows that our acyclic join algorithm is output-optimal (up to a logarithmic factor) when $\mathrm{OUT} \leq p \cdot \mathrm{IN}$.

Furthermore, the same argument for Corollary 4.3 can be used here to show that instance-optimal algorithms do not exist for any acyclic but non-r-hierarchical join.

Corollary 5.4. *For any* $\mathrm{IN} \geq p^{3/2}$*, there is an instance* $\mathcal{R}$ *with input size* $\Theta(\mathrm{IN})$ *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* $\Omega(\frac{\mathrm{IN}}{p^{1/2}\log\mathrm{IN}})$*, while* $L_{instance}(p, \mathcal{R}) = O(\frac{\mathrm{IN}}{p})$.

6 JOIN-AGGREGATE QUERIES

We consider join-aggregate queries over *annotated relations* [14, 20]. Let $(\mathbb{R}, \oplus, \otimes)$ be a commutative semiring. Every tuple t is associated with an *annotation* $w(t) \in \mathbb{R}$. Let $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ be a join hypergraph. 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* and $\bar{\mathbf{y}} = \mathcal{V} - \mathbf{y}$ the non-output attributes. 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, and we extend the terminology from [5] to join-aggregate queries. 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) (coherence) for each attribute $x \in \mathcal{V}$, the nodes containing x are connected in $\mathcal{T}$;
(2) (edge coverage) for each hyperedge $e \in \mathcal{E}$, there exists a node $u \in \mathcal{T}$ such that $e \subseteq u$; and

(3) (width-1) 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 y (a.k.a. *free variables*), we say that $\mathcal{T}$ is *free-connex* if there is a subset of connected nodes of $\mathcal{T}$ including its root, denoted as $\mathcal{T}'$ (such a $\mathcal{T}'$ is said to be a *connex* subset), such that $\mathrm{y} = \bigcup_{u \in \mathcal{T}'} u$. A join-aggregate query Q_{y} is *free-connex* if it has a free-connex width-1 GHD.

As preprocessing, we remove the dangling tuples and apply the reduce procedure repeatedly to remove an $e \in \mathcal{E}$ if there is another $e' \in \mathcal{E}$ such that $e \subset e'$. Note that while dangling tuples can be just discarded, we cannot simply discard $R(e)$ in the reduce procedure. To ensure that the annotations will be computed correctly, we should 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 annotation in $R(e)$ are aggregated into those in $R(e')$.

We find a free-connex width-1 GHD $\mathcal{T}$ of Q [4, 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 $\mathrm{y} = \bigcup_{u \in \mathcal{T}'} u$. We construct an instance $\mathcal{R}_{\mathcal{T}} = \{R(u) : u \in \mathcal{T}\}$ such that $Q_{\mathrm{y}}(\mathcal{R}) = \mathcal{T}(\mathcal{R}_{\mathcal{T}})$, where $\mathcal{T}(\mathcal{R}_{\mathcal{T}})$ denotes the result of running the query defined by $\pi_{\mathrm{y}}\mathcal{T}$ on $\mathcal{R}_{\mathcal{T}}$. Observe that on a reduced 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). Below, we will focus on computing $\mathcal{T}(\mathcal{R}_{\mathcal{T}})$.

Joglekar et al. [20] modified the Yannakakis algorithm into AGGROYANNAKAKIS, and showed that it has load $O(\frac{\mathrm{IN}}{p} + \frac{\mathrm{OUT}}{p})$ on any free-connex join-aggregate query[8]. Since we want to avoid the sub-optimal $O(\frac{\mathrm{OUT}}{p})$ term, we modify their algorithm into LINEARAGGROYANNAKAKIS, which runs with linear load. It aggregates over all the non-output attributes, returning a modified query $\mathcal{T}'(\mathcal{R}_{\mathcal{T}'})$ that only has the output attributes. The details of LINEARAGGROYANNAKAKIS, as well as its guarantees stated in the following lemma, are given in the full version [18].

[8]The bound stated in [20] is actually $O(\frac{(\mathrm{IN}+\mathrm{OUT})^2}{p})$, because they used a sub-optimal binary join algorithm as the subroutine following [1]. Replacing it with the optimal binary join algorithm in [7, 16] yields the claimed bound. In addition, they only considered *simple* join-aggregate queries, which are a strict subclass of free-connex queries. But after our conversion from $Q_{\mathrm{y}}(\mathcal{R})$ to $\mathcal{T}(\mathcal{R}_{\mathcal{T}})$, their algorithm actually works for all free-connex queries.

LEMMA 6.1. *LINEARAGGROYANNAKAKIS is a constant-round, linear-load algorithm that, given any free-connex width-1 GHD $\mathcal{T}$ and an instance $\mathcal{R}_{\mathcal{T}}$, returns an instance $\mathcal{R}_{\mathcal{T}'}$ such that $\mathcal{T}(\mathcal{R}_{\mathcal{T}}) = \mathcal{T}'(\mathcal{R}_{\mathcal{T}'})$, where $\mathcal{T}'$ is the free-connex subset of $\mathcal{T}$.*

Because $\mathcal{T}'$ is acyclic, we can run our output-optimal algorithm to compute $\mathcal{T}'(\mathcal{R}_{\mathcal{T}'})$. More precisely, when the algorithm emits a join result, we compute the $\otimes$-aggregate of the tuples comprising the join result. Note that in the following result, $\mathrm{OUT} = |Q_{\mathrm{y}}(\mathcal{R})|$, i.e., the size of the final output, which can be much smaller than $|Q(\mathcal{R})|$.

THEOREM 6.2. *There is an algorithm that computes any free-connex join-aggregate query in $O(1)$ rounds with load $O(\frac{\mathrm{IN}}{p} + \frac{\sqrt{\mathrm{IN}\cdot\mathrm{OUT}}}{p})$.*

Observing that the join size of a (non-aggregate) join is a special join-aggregate query with $\mathrm{y} = \emptyset$, we obtain the following result, which has been used as a primitive. Note that there is no circular dependency here, because it only uses LINEARAGGROYANNAKAKIS.

COROLLARY 6.3. *For any acyclic join Q and any instance $\mathcal{R}$, $|Q(\mathcal{R})|$ can be computed in $O(1)$ rounds with load $O(\frac{\mathrm{IN}}{p})$.*

Furthermore, if $\mathcal{T}'$ is r-hierarchical, we run our instance-optimal algorithm to compute $\mathcal{T}'(\mathcal{R}_{\mathcal{T}'})$. In fact, we can precisely characterize the class of queries with an r-hierarchical $\mathcal{T}'$. A query is called *out-hierarchical* if it is free-connex and its residual query by removing all non-output attributes is r-hierarchical. We show the following result and its proof can be found in the full version [18].

LEMMA 6.4. *A join-aggregate query Q_{y} is out-hierarchical if and only if it has a width-1 GHD $\mathcal{T}$ with a connex subset $\mathcal{T}'$ such that $\mathrm{y} = \bigcup_{u \in \mathcal{T}'} u$ and $\mathcal{T}'$ is r-hierarchical.*

THEOREM 6.5. *For out-hierarchical query Q_{y} and any instance $\mathcal{R}$, there is an algorithm computing it in $O(1)$ rounds with load $O(\frac{\mathrm{IN}}{p} + L_{instance}(p, \mathcal{R}, \mathrm{y}))$.*

Note that the instance-optimal lower bound L_{instance} for a join-aggregate query is defined with respect to the output attributes only, i.e., $L_{\mathrm{instance}}(p, \mathcal{R}, \mathrm{y}) := \max_{S \subseteq \mathcal{E}} \left(\frac{|\pi_{\mathrm{y}} Q(\mathcal{R}, S)|}{p}\right)^{\frac{1}{|S|}}$, where $\pi_{\mathrm{y}} Q(\mathcal{R}, S) = \pi_{\mathrm{y}}((\bowtie_{e \in S} R(e)) \ltimes Q(\mathcal{R}))$.

7 A LOWER BOUND ON TRIANGLE JOIN

Finally, we look beyond acyclic joins. In particular, we give an output-sensitive lower bound on the triangle join $Q_{\triangle} = R_1(B, C) \bowtie R_2(A, C) \bowtie R_3(A, B)$. For $Q_{\triangle}$, a worst-case lower bound of $\Omega(\frac{\mathrm{IN}}{p^{2/3}})$ is known, by the following argument: A server loading L tuples can emit at most $O(L^{3/2})$ join results by the AGM bound [3], while the join size of $Q_{\triangle}$ can be as large as $\Omega(\mathrm{IN}^{3/2})$. Then setting $p \cdot L^{3/2} = \Omega(\mathrm{IN}^{3/2})$ yields this lower bound. However, if OUT is used as a parameter, this argument only leads to a lower bound of $\Omega((\frac{\text{OUT}}{p})^{2/3})$. Below, we improve this lower bound to the following:

Theorem 7.1. *For any* $\text{IN}/\log^2 \text{IN} \geq 3p^3$, OUT, *there exists an instance* $\mathcal{R}$ *for* $Q_\triangle$ *with input size* $\Theta(\text{IN})$ *and output size* $\Theta(\text{OUT})$ *such that any tuple-based algorithm computing it in* $O(1)$ *rounds must have a load of* $\Omega(\min\{\frac{\text{IN}}{p} + \frac{\text{OUT}}{p \log N}, \frac{\text{IN}}{p^{2/3}}\})$.

The proof, given in the full version [18], is quite technical, but the intuition is simple: When $\text{OUT} = \Theta(\text{IN}^{3/2})$, the triangles are "dense" enough, so a server can achieve the maximum efficiency and emit $\Theta(L^{3/2})$ 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.

Our lower bound has the following consequences:

(1) When $\text{OUT} \geq \text{IN} \cdot p^{1/3}$, the lower bound becomes $\tilde{\Omega}(\frac{\text{IN}}{p^{2/3}})$, which means that the worst-case optimal algorithm of [23] 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.
(2) 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 $O(\frac{\sqrt{\text{IN} \cdot \text{OUT}}}{p})$. The gap is at least $\tilde{\Omega}(\sqrt{\frac{\text{OUT}}{\text{IN}}})$.

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] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.
[4] G. Bagan. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques.* PhD thesis, Université de Caen, 2009.
[5] 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.
[6] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proc. ACM Symposium on Principles of Database Systems*, 2013.
[7] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *Proc. ACM Symposium on Principles of Database Systems*, 2014.
[8] 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.
[9] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *Proc. ACM Symposium on Principles of Database Systems*, 2017.
[10] A. Björklund, R. Pagh, V. V. Williams, and U. Zwick. Listing triangles. In *Proc. International Colloquium on Automata, Languages, and Programming*, 2014.
[11] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.
[12] R. Fink and D. Olteanu. Dichotomies for queries with negation in probabilistic databases. *ACM Transactions on Database Systems*, 41(1), 2016.
[13] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: structure, algorithms, and applications. In *Lecture Notes in Computer Science*, volume 3787, pages 1–15. Springer, 2005.
[14] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc. ACM Symposium on Principles of Database Systems*, 2007.
[15] M. Grohe and D. Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms*, 11(1):4, 2014.
[16] X. Hu, Y. Tao, and K. Yi. Output-optimal parallel algorithms for similarity joins. In *Proc. ACM Symposium on Principles of Database Systems*, 2017.
[17] 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.
[18] X. Hu and K. Yi. Instance and Output Optimal Parallel Algorithms for Acyclic Joins. *arXiv e-prints*, page arXiv:1903.09717, Mar 2019.
[19] M. Joglekar and C. Ré. It's all a matter of degree: Using degree information to optimize multiway joins. In *Proc. International Conference on Database Theory*, 2016.
[20] 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.
[21] 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.
[22] M. A. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. ACM Symposium on Principles of Database Systems*, 2017.
[23] P. Koutris, P. Beame, and D. Suciu. Worst-case optimal algorithms for parallel query processing. In *Proc. International Conference on Database Theory*, 2016.
[24] P. Koutris, S. Salihoglu, and D. Suciu. *Algorithmic Aspects of Parallel Data Processing.* Now Publishers, 2018.
[25] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *Proc. ACM Symposium on Principles of Database Systems*, 2011.
[26] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM*, 2013.
[27] H. Q. Ngo, D. T. Nguyen, C. Ré, and A. Rudra. Beyond worst-case analysis for joins with minesweeper. In *Proc. ACM Symposium on Principles of Database Systems*, 2014.
[28] 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.
[29] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. In *Proc. ACM Symposium on Principles of Database Systems*, 2014.
[30] M. Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proc. ACM Symposium on Theory of Computing*, 2010.
[31] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
[32] T. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.
[33] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. International Conference on Very Large Data Bases*, pages 82–94, 1981.