Parallel Algorithms for Sparse Matrix Multiplication and Join-Aggregate Queries

Xiao Hu*
Duke University
xh102@cs.duke.edu

Ke Yi†
HKUST
yike@ust.hk

ABSTRACT

In this paper, we design massively parallel algorithms for sparse matrix multiplication, as well as more general join-aggregate queries, where the join hypergraph is a tree with arbitrary output attributes. For each case, we obtain asymptotic improvement over existing algorithms. In particular, our matrix multiplication algorithm is shown to be optimal in the semiring model.

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**; **Database query processing and optimization (theory)**.

KEYWORDS

parallel algorithms; sparse matrix multiplication; join-aggregate query

ACM Reference Format:
Xiao Hu and Ke Yi. 2020. Parallel Algorithms for Sparse Matrix Multiplication and Join-Aggregate Queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3375395.3387657

1 INTRODUCTION

Over the span of half of a century, the running time for multiplying two $U \times U$ matrices has progressively improved from $O(U^{2.807})$ [22] to $O(U^{2.373})$ [20]. However, all these fast matrix multiplication algorithms rely on the algebraic properties of the ring, in particular the existence of additive inverses.

Consider the multiplication of two matrices (a_{ij}) and (b_{jk}). By treating each nonzero entry a_{ij} as a tuple (i, j, a_{ij}) (and similarly for b_{ij}), matrix multiplication can be written as a join-aggregate query over two relations (formal definition given below). However, join-aggregate queries posed on a relational database often have the following additional requirements: (1) The "multiplication" and "addition" operations over the matrix elements may not conform to those in a ring. In general, we would like to process such queries over any semiring [11, 15], which does not have additive inverses. (2) The matrices, both input and output, are sparse, i.e., the number of nonzero entries can be much smaller than U^2. In other words, denoting the number of nonzero entries in the input and output matrices as N and OUT, respectively, we would like running times that depend on N and OUT, but not U. In fact, if the matrices are dense, namely $N = \Theta(U^2)$, then the standard $O(U^3)$-time algorithm is already optimal under the semiring model [12, 21]. In the semiring model, the only way to create new semiring elements is by multiplying or adding existing semiring elements. Thus, as argued in [12, 21], the algorithm must compute all the U^3 *elementary products*, namely $a_{ij}b_{jk}$ for all $1 \le i, j, k \le U$.

1.1 Join-aggregate queries

We now formally define the class of queries studied in this paper. A (natural) *join* is defined as a hypergraph $Q = (\mathcal{V}, \mathcal{E})$, where the vertices $\mathcal{V} = \{x_1, \ldots, x_m\}$ model the *attributes* and the hyperedges $\mathcal{E} = \{e_1, \ldots, e_n\} \subseteq 2^{\mathcal{V}}$ model the *relations*. We restrict our study to joins defined by an acyclic hypergraph where each e_i consists of two attributes, i.e., Q is a tree.

Let $\mathrm{dom}(x)$ be the *domain* of attribute $x \in \mathcal{V}$. An *instance* of Q is a set of relations $\mathcal{R} = \{R_e : e \in \mathcal{E}\}$. Each relation R_e consists a set of *tuples*, where each tuple is an assignment that assigns a value from $\mathrm{dom}(x)$ to x for every $x \in e$. The *full join results* of Q on $\mathcal{R}$, denoted as $Q(\mathcal{R})$, consist of all combinations of tuples, one from each R_e, such that they share common values on their common attributes.

We consider join-aggregate queries over *annotated relations* [11, 15]. Let $(\mathbb{R}, \oplus, \otimes)$ be a commutative semiring. Every tuple t is associated with an *annotation* $w(t) \in \mathbb{R}$. The annotation of a join result $t \in Q(\mathcal{R})$ is

$$w(t) := \prod_{t' \in R_e, \pi_e t = t', e \in \mathcal{E}} w(t'),$$

where the multiplication is done using the $\otimes$ operator. 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* $Q_{\mathbf{y}}(\mathcal{R})$ asks us to compute

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

where the summation is done using the $\oplus$ operator. In plain language, a join-aggregate query (semantically) first computes the full join results $Q(\mathcal{R})$ and the annotation of each result, which is the $\otimes$-aggregate of the tuples comprising the join result. Then it partitions $Q(\mathcal{R})$ into groups by the attributes in $\mathbf{y}$. Finally, for each

*Work done while the author was at HKUST.
†Ke Yi is supported by HKRGC under grants 16202317, 16201318, and 16201819.

PODS'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-7108-7/20/06…$15.00
https://doi.org/10.1145/3375395.3387657

group, it computes the $\oplus$-aggregate of the annotations of the join results in that group.

Join-aggregate queries include many commonly seen database queries as special cases. For example, if we ignore the annotations, then it becomes a join-project query $\pi_{\mathbf{y}}\mathcal{Q}(\mathcal{R})$, also known as a *conjunctive query*. If we take $\mathbb{R}$ be the domain of integers and set $w(t) = 1$, then it becomes the `COUNT(*) GROUP BY y` query; in particular, if $\mathbf{y} = \emptyset$, the query computes the full join size $|\mathcal{Q}(\mathcal{R})|$. If we take $\mathcal{V} = \{A, B, C\}$ with $\mathbf{y} = \{A, C\}$, and $\mathcal{E} = \{e_1 = \{A, B\}, e_2 = \{B, C\}\}$, then it degenerates to matrix multiplication.

We use $N = \sum_{e\in\mathcal{E}} |R_e|$ to denote the input size and $\mathrm{OUT} = |\pi_{\mathbf{y}}\mathcal{Q}(\mathcal{R})|$ the output size. Let $[n]$ stand for $\{1, 2, \cdots, n\}$. We study the data complexity of algorithms, i.e., n and m are considered as constants.

1.2 The Yannakakis algorithm

In the RAM model, there is essentially only one algorithm for computing such queries, known as the Yannakakis algorithm [25] dated back to 1981, which was originally designed for join-project queries. It first removes all the *dangling tuples*, namely tuples that will not appear in the full join results, in $O(N)$ time via a series of semijoins. Then, it picks an arbitrary attribute as the root of the query tree $\mathcal{Q}$, and performs joins and projections in a bottom-up fashion. Specifically, it takes two relation R_e and $R_{e'}$ such that e is a leaf and e' is the parent of e, and replaces $R_{e'}$ with $\pi_{\mathbf{y}\cup\mathrm{anc}(e')}R_e \bowtie R_{e'}$, where $\mathrm{anc}(e')$ are the set of attributes in e' that appear in the ancestors of e'. Then R_e is removed and the step repeats 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 $\pi_{\mathbf{y}\cup\mathrm{anc}(e')}$ by an aggregation [15].

Note that when applied to matrix multiplication $\sum_B R_1(A, B) \bowtie R_2(B, C)$, this algorithm just computes the join $R = R_1(A, B) \bowtie R_2(B, C)$ and then does the aggregation $\sum_B R$.

The running time of this algorithm (after dangling tuples have been removed) is proportional to the largest intermediate join size $|R_e \bowtie R_{e'}|$. It is known that if the query is *free-connex*[1], then the maximum intermediate join size is $O(\mathrm{OUT})$ [3, 15]. For non-free-connex queries, Yannakakis gave an upper bound of $O(N \cdot \mathrm{OUT})$ in his original paper [25]. For matrix multiplication, which is the simplest non-free-connex query, this has been tightened to $O(N\sqrt{\mathrm{OUT}})$ [2], which is also shown to be optimal in the semiring model, as there are instances with $\Omega(N\sqrt{\mathrm{OUT}})$ elementary products. This bound also extends to star queries (see Section 5 for formal definition), for which the bound becomes $O(N \cdot \mathrm{OUT}^{1-1/n})$.

1.3 Massively parallel computation

This paper is concerned with the *Massively Parallel Computation* (MPC) model [5, 6, 18, 19]. In the MPC model, there are p servers connected by a complete communication network. Data is initially distributed across p servers with each server holding N/p tuples. Computation proceeds in terms of rounds. 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 [24], 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.

We will adopt the mild assumption $N \geq p^{1+\epsilon}$ where $\epsilon > 0$ is any small constant. This assumption clearly holds on any reasonable values of N 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 [9]. When $N \geq p^{1+\epsilon}$, many basic operations (see Section 2.1) can be performed in $O(1)$ rounds and $O(N/p)$ load, which is often called "linear load", as it is the load needed to shuffle all input data once.

For upper bounds, we assume that every tuple, any semiring element, as well as any integer of $O(\log N)$ bits, consumes one unit of communication. For lower bounds, similar to [12], we require the algorithm to work over any semiring, and 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. We call this model the *semiring MPC* model.

1.4 Previous MPC algorithms

For a two-way join $R_1(A, B) \bowtie R_2(B, C)$, there is an optimal MPC algorithm with load $O\left(\frac{N}{p} + \sqrt{\frac{\mathrm{OUT}}{p}}\right)$ [5, 13]. Plugging this algorithm into the Yannakakis algorithm, together with the MPC primitives for semijoin and aggregation (see Section 2.1), we can run the Yannakakis algorithm to compute join-project or join-aggregate queries in the MPC model. This is referred to as the *distributed Yannakakis algorithm* in [1, 15]. The load of this algorithm is $O\left(\frac{N}{p} + \frac{J}{p}\right)$, where J is the maximum intermediate join size[2]. Combined with the previously known bounds on J [2, 3, 15, 25], this implies that it can compute free-connex queries with load $O\left(\frac{N}{p} + \frac{\mathrm{OUT}}{p}\right)$, matrix multiplication with load $O\left(\frac{N}{p} + \frac{N\sqrt{\mathrm{OUT}}}{p}\right)$, star queries with load $O\left(\frac{N}{p} + \frac{N\cdot\mathrm{OUT}^{1-1/n}}{p}\right)$, and general acyclic join-aggregate queries with load $O\left(\frac{N}{p} + \frac{N\cdot\mathrm{OUT}}{p}\right)$. The results are summarized in Table 1.

Interestingly, while the Yannakakis algorithm's $O(N + \mathrm{OUT})$ running time for free-connex queries is clearly optimal in the RAM

[1] In the case of tree queries, being free-connex means that all the output attributes form a connected subtree in $\mathcal{Q}$.

[2] The bounds stated in [1, 15] are worse, as they did not plug in the optimal two-way join algorithm of [5, 13].

model, its $O(\frac{N}{p} + \frac{\text{OUT}}{p})$ load is not optimal in the MPC model. We recently designed an algorithm for free-connex queries with load $O(\frac{N}{p} + \frac{\sqrt{N \cdot \text{OUT}}}{p})$ [14], which is also shown to be optimal when $\text{OUT} \leq c \cdot p \cdot N$ for some sufficiently large constant $c > 0$.

There are also worst-case optimal MPC algorithms [16, 23] for computing full queries, i.e., $\mathbf{y} = \mathcal{V}$. These algorithms have load $O\left(\frac{N}{p^{1/\rho^*}}\right)$, where ρ^* is the fractional cover number of the join graph $\mathcal{Q}$. For join-aggregate queries, one can use their algorithms to compute the full join first, and then perform the aggregation. However, the aggregation step will become the bottleneck with a load of $O(\frac{\text{OUT}_f}{p})$, where OUT_f is the size of the full join results. This approach is thus no better than the Yannakakis algorithm since we always have $\text{OUT}_f \geq J$.

1.5 Our results

In this paper, we study non-free-connex tree queries where the output attributes can be arbitrary. Table 1 summarizes our results, where the $\widetilde{O}$ notation hides polylogarithmic factors.

We start with the 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). The load is just $O(1)$. Note that when $N_1 = 1$ or $N_2 = 1$, for each $(a, c) \in \pi_{A,C} R_1 \bowtie R_2$, there is only one b such that $(a, b) \in R_1, (b, c) \in R_2$, so there is no need to add semiring elements. Thus all query results can be computed locally after the broadcast.

For $N_1, N_2 \geq 2$, we design an MPC algorithm for matrix multiplication with load $\widetilde{O}\left(\frac{N_1+N_2}{p} + \min\left\{\sqrt{\frac{N_1 N_2}{p}}, \frac{N_1^{1/3} \cdot N_2^{1/3} \cdot \text{OUT}^{1/3}}{p^{2/3}}\right\}\right)$, which simplifies to $\widetilde{O}\left(\frac{N}{p} + \min\left\{\frac{N}{\sqrt{p}}, \frac{N^{2/3} \cdot \text{OUT}^{1/3}}{p^{2/3}}\right\}\right)$ when $N_1 = N_2 = N$. One can verify that this presents an asymptotic improvement over the Yannakakis algorithm for any $\text{OUT} = \omega(1)$. In fact, our algorithm performs the same amount of computation as the Yannakakis algorithm and computes all the $O(N\sqrt{\text{OUT}})$ 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.

Furthermore, we show that this bound is optimal in the semiring MPC model. Specifically, we show that for any $N_1, N_2 \geq 2$ and any $\max\{N_1, N_2\} \leq \text{OUT} \leq N_1 N_2$, there is an instance with input size N_1, N_2 and output size OUT, on which the load of any algorithm has to be at least this bound in the semiring MPC model. In fact, the lower bound holds even if the algorithm is only required to work over *idempotent* semirings, i.e., $a \oplus a = a$ for any element a in the semiring.

Next, we generalize our matrix multiplication algorithm to line queries and star queries, defined as follows.

(1) Line query: $\sum_{A_2, A_3, \cdots, A_n} R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie \cdots \bowtie R_n(A_n, A_{n+1})$;
(2) Star query: $\sum_B R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \cdots \bowtie R_n(A_n, B)$.

Finally, we give an algorithm that can handle tree queries with arbitrary output attributes. As seen in Table 1, we have been able to obtain asymptotic improvement over the Yannakakis algorithm in each case, although we do not have matching lower bounds.

One major technical difficulty we had to overcome is that the value of OUT is not readily available. Although it is known that OUT can be computed for free-connex queries with linear load [14, 15], computing the query size for non-free-connex queries still remains an open problem, even in the RAM model. In fact, the only known method for computing the query size for non-free-connex queries is to find all the query results, which is exactly the problem we aim to solve. We get around this chicken-and-egg problem by two approaches: (1) For matrix multiplication and line queries, we are able to obtain a constant-factor approximation of OUT with linear load, which is sufficient for achieving the asymptotic bounds we claim. (2) For star and tree queries, even a constant-factor approximation looks difficult, so we try to make the algorithms *oblivious* to OUT, i.e., OUT is only used in the analysis but not needed by the algorithm itself.

2 PRELIMINARIES

2.1 MPC Primitives

We mention the following deterministic primitives in the MPC model, which can be computed with load $O(\frac{N}{p})$ in $O(1)$ rounds. Assume $N > p^{1+\epsilon}$ where $\epsilon > 0$ is any small constant.

Sorting [10]. Assume all servers are labeled as $1, 2, \cdots, p$. Given N elements, redistribute them so that each server has $O(\frac{N}{p})$ 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 [13]. Given N pairs in terms of (key, value), compute the "sum" of values for each key, where the "sum" is defined by any associative operator. Note that an aggregation $\sum_{\mathbf{y}} R$ 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 $a \in \text{dom}(v)$ in relation R_e is defined as the number of tuples in R_e having this value in attribute v, i.e., $|\sigma_{v=a} R_e|$. Each tuple $t \in R_e$ is considered to have "key" $\pi_v t$ and "value" 1.

Multi-search [13]. 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 $N = N_1 + N_2$. For each x_i, find its predecessor in Y, i.e., the largest element in Y but smaller than x_i. Note that a semijoin can also be computed by a multi-search.

Remove dangling tuples [14, 25]. For any acyclic join, all dangling tuples, i.e., those that will not participate in the full join results, can be removed by a series of semijoins. Note that after dangling tuples are removed, we can determine whether the output of a query is empty or not.

Parallel-packing [14]. Given N numbers $x_1, x_2, \cdots, x_N$ where $0 < x_i \leq 1$ for $i \in [N]$, 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 N 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 $m \leq 1 + 2 \sum_i x_i$.

Query	The Yannakakis algorithm	New results
Matrix Multiplication	$O\left(\frac{N}{p} + \frac{N\cdot\sqrt{\text{OUT}}}{p}\right)$ [2, 15]	$\widetilde{O}\left(\frac{N_1+N_2}{p} + \min\left\{\sqrt{\frac{N_1N_2}{p}}, \frac{N_1^{1/3}\cdot N_2^{1/3}\cdot \text{OUT}^{1/3}}{p^{2/3}}\right\}\right)$ (1) optimal up to poly-logarithmic factors for $N_1, N_2 \geq 2$; (2) $\widetilde{O}\left(\frac{N}{p} + \min\left\{\frac{N}{\sqrt{p}}, \frac{N^{2/3}\cdot \text{OUT}^{1/3}}{p^{2/3}}\right\}\right)$ if $N_1 = N_2 = N$;
Star	$O\left(\frac{N}{p} + \frac{N\cdot \text{OUT}^{1-\frac{1}{n}}}{p}\right)$ [2, 15]	$\widetilde{O}\left((\frac{N\cdot\text{OUT}}{p})^{2/3} + \frac{N\cdot\text{OUT}^{1/2}}{p} + \frac{N+\text{OUT}}{p}\right)$
Line	$O\left(\frac{N}{p} + \frac{N\cdot\text{OUT}}{p}\right)$ [15]	
Tree		$\widetilde{O}\left(\frac{N\cdot\text{OUT}^{2/3}}{p} + \frac{N+\text{OUT}}{p}\right)$

Table 1: Summary of results. 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 N where there are at most N tuples in each relation. OUT **is the output size. p is the number of servers in the MPC model.**

2.2 Estimate OUT

We show how to obtain a constant-factor approximation of OUT for line queries (including matrix multiplication as a special case) in $O(1)$ rounds with linear load. The similar idea has been used by [8] in the RAM model.

We borrow the technique of k *minimum values* (KMV) [4, 7], which is more commonly used to estimate the number of distinct elements in the streaming model. KMV works by applying a hash function to the input items, and keeping the k minimum hash values, denoted as $v_1, v_2, \cdots, v_k$. It has been shown that, with $k = O(\frac{1}{\epsilon^2})$, the estimator $\frac{k-1}{v_k}$ is an $(1+\epsilon)$-approximation of the number of distinct items in the data stream, with at least constant probability. Moreover, given the KMVs of two sets, the KMV of the union of the two sets can be computed by simply "merging" the two KMVs, i.e., keeping the k minimum values of the $2k$ values from the two KMVs, provided that they use the same hash function.

On a line query, for each $a \in \text{dom}(A_1)$, we will obtain a constant-factor approximation of $\text{OUT}_a = |\pi_{A_{n+1}} R_1(a, A_2) \bowtie R_2(A_2, A_3) \bowtie \cdots \bowtie R_n(A_n, A_{n+1})|$. Note that $\text{OUT} = \sum_{a\in\text{dom}(A_1)} \text{OUT}_a$. To do so, we compute a hash value for each distinct value in $\text{dom}(A_{n+1})$. Then it suffices to compute, for each $a \in \text{dom}(A_1)$, a KMV (with a constant k) over all distinct values in $\text{dom}(A_{n+1})$ that can join with a. This can be done by calling the reduce-by-key primitive n times using the merge operation above to compute the "sum". More precisely, for $i = n, n-1, \ldots, 1$, we use reduce-by-key to compute the KMV for each $a \in \text{dom}(A_i)$, by merging all the KMVs for $b \in \text{dom}(A_{i+1})$ such that $(a, b) \in R_i(A_i, A_{i+1})$.

The KMV obtained from each $a \in \text{dom}(A_1)$ gives us a constant-factor approximation of OUT_a only with constant probability. To boost the success probability, we run $O(\log N)$ instances of this algorithm in parallel using independent random hash functions, and return the median estimator for each OUT_a. This boosts the success probability to $1-1/N^{O(1)}$ for each OUT_a. By a union bound, the probability that all estimators are constant-factor approximations is also $1 - 1/N^{O(1)}$. Then, we also have a constant-factor approximation of OUT. The load of this algorithm is $\widetilde{O}(\frac{N}{p})$.

3 MATRIX MULTIPLICATION

In this section, we study the matrix multiplication problem, i.e., the query $\sum_B R_1(A, B) \bowtie R_2(B, C)$. Suppose R_1 and R_2 have N_1, N_2 tuples respectively, and the query has output size OUT. As mentioned in Section 1.5, the problem is trivial if $N_1 = 1$ or $N_2 = 1$. For $N_1, N_2 \geq 2$, we show the following result.

Theorem 1. *Matrix multiplication can be computed in the MPC model in $O(1)$ rounds with load w.h.p.*[3]

$$\widetilde{O}\left(\frac{N_1+N_2}{p} + \min\left\{\sqrt{\frac{N_1N_2}{p}}, \frac{N_1^{1/3}\cdot N_2^{1/3}\cdot \text{OUT}^{1/3}}{p^{2/3}}\right\}\right).$$

First, note that if $N_1/N_2 < 1/p$ or $N_1/N_2 > p$, then $\frac{N_1+N_2}{p}$ asymptotically dominates $\sqrt{\frac{N_1N_2}{p}}$, so the bound simplifies to $\widetilde{O}(\frac{N_1+N_2}{p})$. For this case, we give a simple algorithm. Assume $N_1/N_2 < 1/p$ (the other case is symmetric). We first remove the dangling tuples. Then sort R_2 by attribute C. We note that after dangling tuples are removed, for any $c \in \text{dom}(C)$, we have $|R_2(B, c)| \leq |\pi_B R_1(A, B)| \leq N_1 \leq \frac{N_2}{p}$. So after sorting, tuples in R_2 with the same value on attribute C must land on the same server or two consecutive servers. In the latter case, we use another round to move them to the same server. Next, we broadcast R_1 to all servers, and ask each server to compute the query on its local data. No further aggregation is needed as tuples from different servers have disjoint C values. This algorithm has a load of $O(N_1 + \frac{N_2}{p}) = O(\frac{N_1+N_2}{p})$. Below, we assume $1/p \leq N_1/N_2 \leq p$.

3.1 Worst-case optimal algorithm

We first describe an algorithm with load $O\left(\sqrt{\frac{N_1N_2}{p}}\right)$. This is actually worst-case optimal because when $|\text{dom}(B)| = 1$, there are N_1N_2 elementary products. A server with load L can compute $O(L^2)$ of them in a constant number of rounds, so we have $pL^2 = \Omega(N_1N_2)$, i.e., $L = \Omega\left(\sqrt{\frac{N_1N_2}{p}}\right)$.

Set $L = \sqrt{\frac{N_1N_2}{p}}$. We describe an algorithm with load $O(L)$ below.

Step 1: Compute data statistics. We first compute, for each value $a \in \text{dom}(A)$, its degree in $R_1(A, B)$, i.e., $|R_1(a, B)|$, as well as the degree of each $c \in \text{dom}(C)$ in $R_2(B, C)$, using the reduce-by-key primitive. A value $a \in \text{dom}(A)$ is *heavy* if its degree in $R_1(A, B)$

[3] The "w.h.p." means that the load complexity holds with probability at least $1-1/N^{O(1)}$. In the following results, it can be phased in the same way.

is greater than L, and *light* otherwise. The set of heavy values in $\mathrm{dom}(A)$ is denoted as

$$A^{\text{heavy}} = \{a \in \mathrm{dom}(A) : |R_1(a, B)| \geq L\},$$

and the set of light values in $\mathrm{dom}(A)$ is $A^{\text{light}} = \mathrm{dom}(A) - A^{\text{heavy}}$. Similarly, a value $c \in \mathrm{dom}(C)$ is *heavy* if its degree is greater than L in $R_2(B, C)$, and *light* otherwise. Similarly, the set of heavy values in $\mathrm{dom}(C)$ is denoted as C^{heavy} and C^{light}. Observe that $|A^{\text{heavy}}| \leq \frac{N_1}{L}$ and $|C^{\text{heavy}}| \leq \frac{N_2}{L}$.

In this way, the original query can be decomposed into four subqueries:

$$\sum_B R_1(A^X, B) \bowtie R_2(B, C^Y),$$

where X, Y can be either heavy or light. Note that results produced by these subqueries are disjoint and the final aggregated result is just their union. We handle each subquery separately.

Step 2: Heavy-heavy. For each pair (a, c) with $a \in A^{\text{heavy}}$ and $c \in C^{\text{heavy}}$, we allocate

$$p_{a,c} = \left\lceil \frac{|R_1(a, B)|}{L} + \frac{|R_2(B, c)|}{L} \right\rceil$$

servers to compute the subquery $\sum_B R_1(a, B) \bowtie R_2(B, c)$ by using multi-search and reduce-by-key primitives. The total number of servers allocated is

$$\begin{aligned}\sum_{a,c} p_{a,c} =& \frac{N_1 N_2}{L^2} + \frac{1}{L} \cdot \sum_{a,c} (|R_1(a, B)| + |R_2(B, c)|) \\ \leq& \frac{N_1 N_2}{L^2} + \frac{1}{L} \cdot \frac{N_2}{L} \cdot N_1 + \frac{1}{L} \cdot \frac{N_1}{L} \cdot N_2 = O(p).\end{aligned}$$

Note that all subqueries can be computed in parallel and each server has a load of $O(L)$.

Step 3. Heavy-light. For each value $a \in A^{\text{heavy}}$, we allocate

$$p_a = \left\lceil \frac{|R_1(a, B)| + |R_2(B, C^{\text{light}})|}{L} \right\rceil$$

servers to compute the subquery $\sum_B R_1(a, B) \bowtie R_2(B, C^{\text{light}})$ by using multi-search and reduce-by-key primitives. The total number of servers used is

$$\begin{aligned}\sum_a p_a =& \frac{N_1}{L} + \frac{1}{L} \sum_a (|R_1(a, B)| + |R_2(B, C^{\text{light}})|) \\ \leq& 2 \cdot \frac{N_1}{L} + \frac{1}{L} \cdot \frac{N_1}{L} \cdot N_2 = O(p),\end{aligned}$$

where the last inequality is implied by the fact that $\frac{N_1}{p} \leq L$ under the assumption that $1/p \leq N_1/N_2 \leq p$.

Note that all the subqueries can be computed in parallel and this step has a load of $O(\max_a \frac{|R_1(a,B)|+|R_2(B,C^{\text{light}})|}{p_a}) = O(L)$.

The light-heavy case can be handled symmetrically.

Step 4. Light-light. In this case, we use parallel-packing to divide A^{light} 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 $R_1(A^{\text{light}}, B)$. Similarly, we divide C^{light} into $l = O(\frac{N_2}{L})$ disjoint groups $C_1, C_2, \cdots, C_l$ such that each group has total degree $O(L)$ in $R_2(B, C^{\text{light}})$. 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 coordinates (i, j) for $i \in [\lceil \frac{N_1}{L} \rceil], j \in [\lceil \frac{N_2}{L} \rceil]$. The total number of servers used is

$$\left\lceil \frac{N_1}{L} \right\rceil \cdot \left\lceil \frac{N_2}{L} \right\rceil \leq \frac{N_1}{L} \cdot \frac{N_2}{L} + \frac{N_1}{L} + \frac{N_2}{L} + 1 = O(p),$$

where $\frac{N_1+N_2}{p} \leq L$ holds under the assumption that $1/p \leq N_1/N_2 \leq p$. The server (i, j) will receive all tuples in $\sigma_{A \in A_i} R_1(A, B)$ and $\sigma_{C \in C_j} R_2(B, C)$ and then compute the subquery $\sum_B \sigma_{A \in A_i} R_1(A, B) \bowtie \sigma_{C \in C_j} R_2(B, C)$ locally.

Obviously, the results emitted by different servers are disjoint and the union is the final aggregate result. In this step, each server has a load of $O(L)$ as the number of tuples received from both relations can be bounded by L.

Lemma 1. *For any sparse matrix multiplication with input sizes $N_1, N_2 \geq 2$, there is an algorithm computing it in $O(1)$ rounds with load $O\left(\frac{N_1+N_2}{p} + \sqrt{\frac{N_1 N_2}{p}}\right)$.*

3.2 Output-sensitive algorithm

In this part, we consider output-sensitive algorithms for computing sparse matrix multiplication. We first remove all dangling tuples with linear load. Then we obtain a constant-factor approximation of OUT as described in Section 2.2. Below, we will not distinguish between OUT and its constant-factor approximation.

We begin with an observation that, if $\mathrm{OUT} \leq N/p$, then the matrix multiplication can be computed in $O(1)$ rounds with load $O\left(\frac{N}{p}\right)$. The algorithm, referred to as LinearSparseMM, is very simple. Note that the degree of any $b \in \mathrm{dom}(B)$ in either R_1 or R_2 must be smaller than OUT, hence smaller than N/p. First, sort all tuples by attribute B. After this step, all tuples with the same value b will land on the same server (or on two consecutive servers, in which case we bring them to one server using another round). Then let each server compute its local aggregate results $\sum_B R_1^i(A, B) \bowtie R_2^i(B, C)$, where R_1^i (resp. R_2^i are tuples from R_1, (resp. R_2) landing on server i. Note that each local query produces at most $\mathrm{OUT} \leq N/p$ results. At last, use reduce-by-key on all the local results to obtain the final aggregated results.

Below, we consider the case $\mathrm{OUT} > N/p$. Set $L = \left(\frac{N_1 \cdot N_2 \cdot \mathrm{OUT}}{p^2}\right)^{1/3} + \frac{N_1+N_2}{p}$.

Step 1. Compute data statistics. Recall that when computing a constant-factor approximation for OUT, we have also obtained a constant-factor approximation of OUT_a, for each value $a \in \mathrm{dom}(A)$, i.e., the number of final aggregate results in which a participates.

A value $a \in \mathrm{dom}(A)$ is *heavy* if $\mathrm{OUT}_a \geq \sqrt{\frac{N_2 \cdot \mathrm{OUT} \cdot L}{N_1}}$, and *light* otherwise. The set of heavy values in $\mathrm{dom}(A)$ is denoted as A^{heavy}, and the set of light values A^{light}. Observe that $|A^{\text{heavy}}| \leq \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_1}{N_2}}$ since the (disjoint) union of results produced by each value in A^{heavy} must be the final aggregate result.

Step 2. Handle heavy rows. We use the Yannakakis algorithm to compute $\sum_B R_1(A^{\text{heavy}}, B) \bowtie R_2(B, C)$ with load $O(\frac{J}{p})$, where $J = |R_1(A^{\text{heavy}}, B) \bowtie R_2(B, C)|$. For this join, J can be bounded

by $O\left(\sqrt{\frac{N_1 N_2 \mathrm{OUT}}{L}}\right)$, since each tuple in $R_2(B,C)$ can join with at most $O\left(\sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_1}{N_2}}\right)$ values in A^{heavy}. So this step has a load of $O\left(\frac{1}{p} \cdot \sqrt{\frac{N_1 N_2 \mathrm{OUT}}{L}}\right) = O(L)$.

Step 3. Handle light rows and heavy columns. We then divide A^{light} into $k_1 = O\left(\sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_1}{N_2}}\right)$ disjoint groups $A_1, A_2, \cdots, A_{k_1}$ such that values in each group appear in $O\left(\sqrt{\frac{N_2 \cdot \mathrm{OUT} \cdot L}{N_1}}\right)$ final results. For group A_i, if $|\sigma_{A \in A_i} R_1(A,B)| \le L$, then $|\pi_A \sigma_{A \in A_i} R_1(A,B) \bowtie R_2(B,c)| \le L$ holds for each value $c \in \mathrm{dom}(C)$. In this step, we only tackle the groups with $|\sigma_{A \in A_i} R_1(A,B)| > L$. Note that there are at most $O\left(\min\left\{\frac{N_1}{L}, \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_1}{N_2}}\right\}\right)$ such groups.

We first estimate for each group A_i, the number of aggregate results participated by each value $c \in \mathrm{dom}(C)$ in the subquery induced by A_i, i.e., $|\pi_A \sigma_{A \in A_i} R_1(A,B) \bowtie R_2(B,c)|$. We allocate

$$p_i = \left\lceil \frac{|\sigma_{A \in A_i} R_1(A,B)| + |R_2(B,C)|}{L} \right\rceil$$

servers for each such A_i and run the primitive described in Section 2.2. It can be easily checked that each group incurs a load of $O\left(\max_i \frac{|\sigma_{A \in A_i} R_1(A,B)| + |R_2(B,C)|}{p_i}\right) = O(L)$. The number of servers used for computing these statistics is

$$\begin{aligned}\sum_i p_i &= \sum_i \left(\frac{|\sigma_{A \in A_i} R_1(A,B)| + |R_2(B,C)|}{L} + 1\right) \\ &\le \frac{N_1}{L} + \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_1}{N_2}} \cdot \frac{N_2}{L} + \frac{N_1}{L} = O(p).\end{aligned}$$

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 values in $\mathrm{dom}(C)$, with respect to A_i, is denoted as

$$C_i^{\text{heavy}} = \{c \in \mathrm{dom}(C) : |\pi_A R_1(A^{\text{light}}, B) \bowtie \sigma_{C=c} R_2(B,C)| \ge L\}$$

and the set of light values is $C_i^{\text{light}} = \mathrm{dom}(C) - C_i^{\text{heavy}}$. Observe that $|C_i^{\text{heavy}}| \le \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_2}{N_1}}$. For each value $c \in C_i^{\text{heavy}}$, we allocate

$$p_{i,c} = \left\lceil \frac{|\sigma_{A \in A_i} R_1(A,B)| + |\sigma_{C=c} R_2(B,C)|}{L} \right\rceil$$

servers to compute the subquery $\sum_B \sigma_{A \in A_i} R_1(A,B) \bowtie \sigma_{C=c} R_2(B,C)$ by multi-search and reduce-by-key primitives. Each server has a load of $O\left(\frac{|\sigma_{A \in A_i} R_1(A,B)| + |\sigma_{C=c} R_2(B,C)|}{p_{i,c}}\right) = O(L)$.

Note that all the subqueries are computed in parallel and the total number of servers used is

$$\begin{aligned}\sum_i \sum_c p_{i,c} &= \sum_i \sum_c \left(\frac{|\sigma_{A \in A_i} R_1(A,B)| + |\sigma_{C=c} R_2(B,C)|}{L} + 1\right) \\ &= \frac{N_1}{L} \cdot \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_2}{N_1}} + \frac{N_2}{L} \cdot \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_1}{N_2}} \\ &\quad + \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_2}{N_1}} \cdot \frac{N_1}{L} = 3 \cdot \sqrt{\frac{N_1 \cdot N_2 \cdot \mathrm{OUT}}{L^3}} = O(p).\end{aligned}$$

Step 4. Handle light rows and light columns. Note that on any group A_i, $|\sum_B \sigma_{A \in A_i} R_1(A,B) \bowtie \sigma_{C=c} R_2(B,C)| \le L$ holds for each value $c \in C_i^{\text{light}}$ by definition. For each group A_i, we allocate p_i servers as the same in Step 3 and run the parallel-packing primitive to divide C_i^{light} into $k_2 = O\left(\sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_2}{N_1}}\right)$ 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 $\sum_B \sigma_{A \in A_i} R_1(A,B) \bowtie R_2(B,C)$. The number of servers used for computing these statistics is $O(p)$ following the same argument in Step 3.

Note that each pair of (A_i, C_j^i) further defines a subquery as

$$\sum_B \sigma_{A \in A_i} R_1(A,B) \bowtie \sigma_{C \in C_j^i} R_2(B,C).$$

which derives an instance $R_{i,j} = \sigma_{A \in A_i} R_1(A,B) \cup \sigma_{C \in C_j^i} R_2(B,C)$. Over all subqueries, the number of (duplicated) input tuples is

$$\sum_{i,j} |\sigma_{A \in A_i} R_1(A,B)| + |\sigma_{C \in C_j^i} R_2(B,C)| \le \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{N_1 N_2}.$$

Sort input tuples over all instances by (i,j) lexicographically with load $O\left(\frac{\sqrt{N_1 N_2}}{p} \cdot \sqrt{\frac{\mathrm{OUT}}{L}}\right) = O(L)$. We distinguish the subquery induced by (A_i, C_j) into two cases. If all input tuples in $R_{i,j}$ land on a single server, then let this server compute the $\sum_B \sigma_{A \in A_i} R_1(A,B) \bowtie \sigma_{C \in C_j^i} R_2(B,C)$ locally. Otherwise, input tuples in R_{ij} land on more than 2 servers. Note that the number of such instances is at most p. For each subquery defined by (A_i, C_j^i), we allocate

$$p_{i,j} = \left\lceil \frac{|\sigma_{A \in A_i} R_1(A,B)| + |\sigma_{C \in C_j^i} R_2(B,C)|}{L} \right\rceil$$

servers and invoke LinearSparseMM algorithm to compute the $\sum_B \sigma_{A \in A_i} R_1(A,B) \bowtie \sigma_{C \in C_j^i} R_2(B,C)$. Note that each subquery has output size smaller than OUT, thus can be computed with load $O\left(\frac{|R_{i,j}|}{p_{i,j}} + L\right) = O(L)$. Note that all the subqueries are computed in parallel and the total number of servers used is

$$\begin{aligned}\sum_{i,j} p_{i,j} &= \sum_{i,j} \left(\frac{|\sigma_{A \in A_i} R_1(A,B)| + |\sigma_{C \in C_j^i} R_2(B,C)|}{L} + 1\right) \\ &\le \frac{N_1}{L} \cdot \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_2}{N_1}} + \frac{N_2}{L} \cdot \sqrt{\frac{\mathrm{OUT}}{L}} \cdot \sqrt{\frac{N_1}{N_2}} + p = O(p).\end{aligned}$$

Over all steps, this algorithm has a load of $O(L)$.

Lemma 2. *For any sparse matrix multiplication with input sizes $N_1, N_2 \ge 2$ and output size* OUT*, there is an algorithm computing it in $O(1)$ rounds with load $\widetilde{O}\left(\frac{N_1+N_2}{p} + \frac{N_1^{1/3} \cdot N_2^{1/3} \cdot \mathrm{OUT}^{1/3}}{p^{2/3}}\right)$ w.h.p.*

Finally, depending on the value of OUT, we choose to run either the algorithm in Section 3.1 or the one in Section 3.2, thus obtaining the bound claimed in Theorem 1.

3.3 Lower Bounds

We prove the following two lower bounds, which together show that the bound in Theorem 1 is optimal. Both lower bounds hold over idempotent semirings, i.e., semirings with the property that $a \oplus a = a$ for any semiring element a.

Theorem 2. *For any $N_1, N_2 \geq 2$ and $\max\{N_1, N_2\} \leq \text{OUT} \leq N_1 N_2$, there is an instance $\mathcal{R}$ with input sizes $\Theta(N_1), \Theta(N_2)$ and output size $\Theta(\text{OUT})$, such that any algorithm computing the matrix multiplication on $\mathcal{R}$ in $O(1)$ rounds must incur a load of $\Omega\left(\frac{N_1+N_2}{p}\right)$ in the idempotent semiring MPC model.*

Proof. Without loss of generality, assume $N_2 \geq N_1 \geq 2$. The hard instance is constructed as follows. Set $\text{dom}(A) = \{a\}$, $\text{dom}(B) = \{b_i : i \in [N_1]\}$ and $\text{dom}(C) = \{c_j : j \in [\frac{N_2}{2}]\}$. Set the instance as $R_1(A, B) = \{a\} \times \text{dom}(B)$ and $R_2(B, C) = \{b_1, b_2\} \times \text{dom}(C)$. Note that the output size is $1 \times 2 \times \frac{N_2}{2} = N_2$. Then we add $O(N_1)$ dummy tuples to R_1 and $O(N_2)$ dummy tuples to R_2 so that the output size is $\Theta(\text{OUT})$. The initial data distribution is a way such that no two tuples of R_2 with the same value $c \in \text{dom}(C)$ are on the same server.

By the semiring model requirement, for each $c \in \text{dom}(C)$, the two tuples $(b_1, c), (b_2, c)$ in R_2 have to meet on at least one server to produce the aggregate for (a, c). Under the initial data distribution, at least $N_2/2$ tuples have to be sent to other servers for computing the aggregation, which incurs a load of at least $N_2/2p$. So lower bound of the load is $\Omega(\frac{N_2}{p}) = \Omega(\frac{N_1+N_2}{p})$. □

Theorem 3. *For any $N_1, N_2 \geq 2$ and $\max\{N_1, N_2\} \leq \text{OUT} \leq N_1 N_2$, there exists an instance $\mathcal{R}$ for sparse matrix multiplication with input sizes $\Theta(N_1), \Theta(N_2)$ and output size $\Theta(\text{OUT})$, such that any algorithm computing $\mathcal{R}$ in $O(1)$ rounds in the idempotent semiring MPC model must incur a load of*

$$\Omega\left(\min\left\{\sqrt{\frac{N_1 N_2}{p}}, \frac{N_1^{1/3} \cdot N_2^{1/3} \cdot \text{OUT}^{1/3}}{p^{2/3}}\right\}\right).$$

Proof. First consider the case $N_2/N_1 > \text{OUT}$. In this case, it is sufficient to prove an $\Omega\left(\left(\frac{N_2}{p}\right)^{2/3}\right)$ lower bound, which is at least as large as the second term in the claimed lower bound. If $N_2 < p$, this degenerates to the trivial bound $\Omega(1)$; otherwise, we have $\left(\frac{N_2}{p}\right)^{2/3} < \frac{N_2}{p} \leq \frac{N_1+N_2}{p}$, and we apply Theorem 2. The case when $N_1/N_2 > \text{OUT}$ can be argued similarly.

Below, we assume that $1/\text{OUT} \leq N_1/N_2 \leq \text{OUT}$. The hard instance is constructed as follows. There are $\sqrt{\frac{N_1 \cdot \text{OUT}}{N_2}}$, $\sqrt{\frac{N_1 N_2}{\text{OUT}}}$, $\sqrt{\frac{N_2 \cdot \text{OUT}}{N_1}}$ distinct values in $\text{dom}(A)$, $\text{dom}(B)$, $\text{dom}(C)$ respectively. Note that all three numbers are at least 1 when $1/\text{OUT} \leq N_1/N_2 \leq \text{OUT}$; thus we can ignore rounding issues without affecting the asymptotic results. Set $R_1(A, B) = \text{dom}(A) \times \text{dom}(B)$ and $R_2(B, C) = \text{dom}(B) \times \text{dom}(C)$. In this constructed instance, there are N_1, N_2 tuples in $R_1(A, B)$, $R_2(B, C)$, respectively. Every value $a \in \text{dom}(A)$ can join with every value $c \in \text{dom}(C)$, so the output size is $|\text{dom}(A)| \cdot |\text{dom}(C)| = \text{OUT}$.

There are two types of tuples in the semiring MPC model: (1) original tuples from R_1, R_2 (or their copies); and (2) aggregate tuples (or their copies) in the form of $(a, c, w, B(ac))$ where $B(ac) \subseteq \text{dom}(B)$ and $w = \sum_{b \in B(ac):(a,b,c) \in R_1 \bowtie R_2} w(a, b) \times w(b, c)$. Note that if $B(ac) = \text{dom}(B)$, the tuple is a final result. We will lower bound the load in terms of the number of type (1) and type (2) tuples received by any server, even assuming $B(ac)$ can be encoded by a constant-size message.

The idempotent semiring model only allows the following two operations to produce new tuples:

(i) $(a, b, w(a, b)) \otimes (b, c, w(b, c)) \to (a, c, w(a, b) \otimes w(b, c), \{b\})$;
(ii) $(a, c, w_1, B(ac)) \oplus (a, c, w_2, B'(ac)) \to (a, c, w_1 \oplus w_2, B(ac) \cup B'(ac))$.

Our hard instance is initially distributed in such a way that R_1 and R_2 are on disjoint servers. This means that initially, there are only type (1) tuples without any communication. We allow each server to store all tuples it has received from previous rounds, thus we may assume that they emit the final results only in the last round. We may assume w.l.o.g. that at the end of each round (i.e., after local computation and before sending messages to the next round), a server keeps at most one aggregate tuple for each (a, c) pair, because if there are two, they can be combined using operation (ii) to produce a more useful aggregate tuple. We call such an aggregate tuple a *partial result* for (a, c) if $B(ac) \neq \text{dom}(B)$. Each partial result must be later aggregated with another produced by some other server (otherwise there is no point in producing this partial result at all), so at least half of all partial results must be shuffled.

Consider the time point before the last round of shuffling. Let $(a, c, w_i, B^i(ac))$ be the aggregate tuple stored on server i for (a, c) if it exists, and define $B^i(ac) = \emptyset$ otherwise. We call an output pair (a, c) *half-complete* if $|\cup_{i \in [p]} B^i(ac)| > \frac{1}{2}|\text{dom}(B)|$, otherwise *half-incomplete*. Consider the following two cases:

Case 1: More than $\frac{\text{OUT}}{2}$ pairs are half-incomplete. For each half-incomplete (a, c) pair, all the original tuples not aggregated so far must be brought to one server in the last round. There are at least $\frac{1}{2}|\text{dom}(B)|$ such tuples in $\sigma_{A=a} R_1$ and also $\frac{1}{2}|\text{dom}(B)|$ in $\sigma_{C=c} R_2$. If the load is L, a server can load all such tuples for at most $\frac{2L}{|\text{dom}(B)|}$ different a's and $\frac{2L}{|\text{dom}(B)|}$ different c's. Thus, the server can complete at most $\left(\frac{2L}{|\text{dom}(B)|}\right)^2 = O\left(\frac{L^2 \cdot \text{OUT}}{N_1 N_2}\right)$ half-incomplete pairs. Over p servers, we must have $p \cdot \frac{L^2 \cdot \text{OUT}}{N_1 N_2} \geq \frac{\text{OUT}}{2}$, thus $L = \Omega\left(\sqrt{\frac{N_1 N_2}{p}}\right)$.

Case 2: More than $\frac{\text{OUT}}{2}$ pairs are half-complete. Then the total "work" done so far is $\sum_{(a,c)} |\cup_{i \in [p]} B^i(ac)| = \Omega(\text{OUT} \cdot |\text{dom}(B)|)$. Let x_i be the number of partial results generated by server i so far. As argued above, at least half of the partial results must be shuffled, so $\sum_i x_i = O(pL)$.

Now, let's consider how much these partial results can contribute to the work done so far. The key is to interpret the work done as a subset of the full join $R_1 \bowtie R_2$, where a full join result (a, b, c) is "done" if $b \in \cup_{i \in [p]} B^i(ac)$. Note that a server contributes (a, b, c) to the work done in a round only if it has the original tuples $(a, b, w(a, b))$, $(b, c, w(b, c))$, and it keeps the partial result for (a, c) at the end of the round. This becomes the triangle join problem, where the server has $O(L)$ tuples in $R_1(A, B)$, $O(L)$ tuples in $R_2(B, C)$, and x_i tuples in an imaginary relation $R_3(A, C)$ that consists of all the (a, c) pairs for which the servers has ever kept a partial result. By the AGM bound, the total work contributed by the server is

therefore at most $O(L \cdot \sqrt{x_i})$. So, the maximum contribution made by p servers is characterized by the following linear program.

$$\max \sum_{i=1}^{p} L \cdot \sqrt{x_i}$$
$$\text{s.t. } x_1 + x_2 + \cdots + x_p \le p \cdot L$$
$$x_i \ge 0, i = 1, \ldots p.$$

The optimal solution for the LP above is $p \cdot L^{3/2}$, when all x_i's take the same value L. Thus, we must have

$$p \cdot L^{3/2} = \Omega(\text{OUT} \cdot |\text{dom}(B)|) = \Omega(\sqrt{N_1 N_2 \text{OUT}}),$$

i.e., $L = \Omega\left(\frac{N_1^{1/3} \cdot N_2^{1/3} \cdot \text{OUT}^{1/3}}{p^{2/3}}\right)$.

Taking the minimum of the two cases, we obtain the desired lower bound. □

Remark. We note that there is an $\Omega\left(\min\left\{\frac{N}{B}\sqrt{\frac{\text{OUT}}{M}}, \frac{N^2}{MB}\right\}\right)$ lower bound for this problem in the semiring external memory model when $N_1 = N_2 = N$ [21]. Meanwhile, there is an MPC-to-EM reduction [17], which states that any MPC algorithm running in r rounds with load $L(N, \text{OUT}, p)$ can be converted to an external memory algorithm incurring $\widetilde{O}\left(\frac{N}{B} + rp^*\frac{M}{B}\right)$ I/Os, where $p^* = \min_p\{L(N, \text{OUT}, p) \le M/r\}$. Taking $M = \Theta(B)$, the external memory lower bound implies a lower bound for constant-round matrix multiplication algorithms in the semiring MPC model as $\widetilde{\Omega}\left(\min\left\{(\frac{N}{p})^{2/3} \cdot \text{OUT}^{1/3}, \frac{N}{\sqrt{p}}\right\}\right)$. However, our lower bound holds for unequal matrix sizes N_1, N_2 as well. Moreover, since our proof is conducted in the MPC model directly, it avoids the poly-logarithmic factors introduced during the MPC-to-EM reduction.

4 LINE QUERIES

In this section, we investigate the chain matrix multiplication problem, which is equivalent to a line query with the two "boundary" attributes being the output attributes:

$$\sum_{A_2, A_3, \cdots, A_n} R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie \cdots \bowtie R_n(A_n, A_{n+1}).$$

We will design a recursive algorithm for handling such a line query over n relations, with the base $n = 2$ being just matrix multiplication. We will assume that all n relations have the same size N, but during the recursion, we will actually need the matrix multiplication algorithm for unequal matrix sizes as described in Section 3.

4.1 The algorithm

We first remove all dangling tuples and then obtain a constant-factor approximation of OUT using the primitive described in Section 2.2.

Step 1: Compute data statistics. We first compute for each value $a \in \text{dom}(A_2)$ its degree in relation $R_1(A_1, A_2)$ using the reduce-by-key primitive. The value $a \in \text{dom}(A_2)$ is *heavy* if its degree in relation $R_1(A_1, A_2)$ is greater than $\sqrt{\text{OUT}}$ and *light* otherwise. The set of heavy values in $\text{dom}(A_2)$ is denoted as

$$A_2^{\text{heavy}} = \{a \in \text{dom}(A_2) : |\sigma_{A_2 = a} R_1(A_1, A_2)| \ge \sqrt{\text{OUT}}\}$$

and the set of light values in $\text{dom}(A_2)$ is $A_2^{\text{light}} = \text{dom}(A_2) - A_2^{\text{heavy}}$. The tuples in $R_1(A_1, A_2)$ and $R_2(A_2, A_3)$ are also identified as *heavy* or *light* correspondingly, depending on their values in attribute A_2.

In this way, we decompose the original query into the following two subqueries:

$$Q^X = \sum_{A_2, \cdots, A_n} R_1(A_1, A_2^X) \bowtie R_2(A_2^X, A_3) \bowtie \cdots \bowtie R_n(A_n, A_{n+1})$$

where X can be either heavy or light.

Step 2: Handling Q^{heavy}. We first remove all dangling tuples in the heavy subquery. On the reduced subquery, our strategy is to (2.1) compute $R(A_2, A_{n+1}) =$

$$\sum_{A_3, A_4, \cdots, A_n} R_2(A_2^{\text{heavy}}, A_3) \bowtie R_3(A_3, A_4) \bowtie \cdots \bowtie R_n(A_n, A_{n+1})$$

using the Yannakakis algorithm; and (2.2) reduce Q^{heavy} to a matrix multiplication as $\sum_{A_2} R_1(A_1, A_2^{\text{heavy}}) \bowtie R(A_2^{\text{heavy}}, A_{n+1})$ and compute it by the output-sensitive algorithm in Section 3.2. More precisely, when using the Yannakakis algorithm in step (2.1), we compute the query from right to left, i.e., for $i = n-1, n-2, \cdots, 2$, we compute $R(A_i, A_{n+1}) = \sum_{A_{i+1}} R_i(A_i, A_{i+1}) \bowtie R(A_{i+1}, A_{n+1})$.

Step 3: Handling Q^{light}. In this case, our strategy is to (3.1) compute $R(A_1, A_3) = \sum_{A_2} R_1(A_1, A_2^{\text{light}}) \bowtie R_2(A_2^{\text{light}}, A_3)$ using the Yannakakis algorithm (i.e., just compute the join and then the aggregation); and (3.2) reduce Q^{light} to a shorter line query as

$$\sum_{A_3, A_4, \cdots, A_n} R(A_1, A_3) \bowtie R_3(A_3, A_4) \bowtie \cdots \bowtie R_n(A_n, A_{n+1})$$

and compute it recursively.

Step 4: Aggregate the two queries. Note that the two queries Q^{heavy} and Q^{light} may produce results with common values on A_1, A_{n+1}, so finally we need to aggregate all the results by A_1, A_{n+1} using reduce-by-key.

4.2 The analysis

To facilitate the induction proof, we will actually prove a slightly tighter bound of our algorithm, as stated in the following lemma.

Lemma 3. *For a length-n line query with $N_1 = |R_1(A_1, A_2)|$, $N_n = |R_n(A_n, A_{n+1})|$, $N = \max_{i \in \{2,3,\cdots,n-1\}} |R_i(A_i, A_{i+1})|$ and output size as* OUT*, there is an algorithm computing it in $O(1)$ rounds with load $\widetilde{O}(L)$ w.h.p., where $L = \frac{(N \cdot N_1)^{1/3} \cdot \text{OUT}^{1/2}}{p^{2/3}} + \frac{(N_1 N_n)^{1/3} \cdot \text{OUT}^{1/3}}{p^{2/3}} + \frac{N^{2/3} \cdot \text{OUT}^{2/3}}{p^{2/3}} + \frac{N \cdot \text{OUT}^{1/2} + N + N_1 + N_n + \text{OUT}}{p}$.*

By setting $N_1 = N_n = N$ in Lemma 3, we obtain the following theorem, as claimed in Table 1.

Theorem 4. *For a line query with input size N and output size* OUT*, there is an algorithm computing it in $O(1)$ rounds with load $O\left(\frac{N \cdot \text{OUT}^{1/2}}{p} + (\frac{N \cdot \text{OUT}}{p})^{2/3} + \frac{N + \text{OUT}}{p}\right)$.*

We prove Lemma 3 in the rest of this section. Note that the base case $n = 2$ is just Lemma 2. For a line query with $n \ge 3$, we consider the load of our algorithm in each of the four steps.

Step 1 has linear load $O(\frac{N + N_1 + N_n}{p})$, since it only involves primitives. Step 4 incurs a load of $O(\frac{\text{OUT}}{p})$ since each subquery produces

at most OUT results, which are the input for the reduce-by-key primitive. Before analyzing the load complexity for Step 2, we point out one important property of Q^{heavy}:

Lemma 4. *In Q^{heavy}, for any $i \geq 2$, each value $b \in \text{dom}(A_i)$ can join with at least $\sqrt{\text{OUT}}$ disjoint values in $\text{dom}(A_1)$.*

Proof. We prove it by induction on i. The lemma holds for $i = 2$ by the definition of heavy values in attribute A_2. Now assume that the lemma holds for i. Consider an arbitrary value $b \in \text{dom}(A_{i+1})$. Recall that all dangling tuples have been removed, so there must exist a tuple $(b', b) \in R(A_i, A_{i+1})$. By the induction hypothesis, b' can join with at least $\sqrt{\text{OUT}}$ distinct values in attribute A_1, thus b can also join with at least $\sqrt{\text{OUT}}$ distinct values in attribute A_1. □

In Step (2.1), we claim that the size of any intermediate join $R(A_i, A_{i+1}) \bowtie R(A_{i+1}, A_{n+1})$ during the Yannakakis algorithm is at most $N \cdot \sqrt{\text{OUT}}$. Consider any $i \in \{2, 3, \cdots, n-1\}$. To relate the intermediate join size with OUT, we introduce an additional relation $R(A_1, A_i) =$

$$\sum_{A_2, \cdots, A_{i-1}} R(A_1, A_2^{\text{heavy}}) \bowtie R(A_2^{\text{heavy}}, A_3) \bowtie \cdots \bowtie R(A_{i-1}, A_i).$$

Note that this relation is only used in the analysis but not computed by our algorithm. We can bound $|R(A_i, A_{i+1}) \bowtie R(A_{i+1}, A_{n+1})|$ by $N \cdot \sqrt{\text{OUT}}$ since

$$\begin{aligned} N \cdot \text{OUT} =& |R(A_1, A_{n+1})| \cdot |R(A_i, A_{i+1})| \\ \geq& |R(A_1, A_i) \bowtie R_i(A_i, A_{i+1}) \bowtie R(A_{i+1}, A_{n+1})| \\ \geq& |R(A_i, A_{i+1}) \bowtie R(A_{i+1}, A_{n+1})| \cdot \min_b |\sigma_{A_i = b} R(A_1, A_i)| \\ \geq& |R(A_i, A_{i+1}) \bowtie R(A_{i+1}, A_{n+1})| \cdot \sqrt{\text{OUT}}, \end{aligned}$$

where the last inequality follows from Lemma 4. So this step has a load of $O\left(\frac{N}{p} + \frac{N \cdot \sqrt{\text{OUT}}}{p}\right)$. In Step (2.2), the input relations of matrix multiplication problem have their sizes bounded by N_1 and $N \cdot \sqrt{\text{OUT}}$ respectively. Plugging to Lemma 2, it can be computed with load $O\left(\frac{(N N_1)^{1/3} \cdot \text{OUT}^{1/2}}{p^{2/3}} + \frac{N \cdot \sqrt{\text{OUT}}}{p} + \frac{\text{OUT}}{p}\right) = O(L)$.

In Step (3.1), the size of full join $R(A_1, A_2^{\text{light}}) \bowtie R(A_2^{\text{light}}, A_3)$ is at most $N \cdot \sqrt{\text{OUT}}$, since each tuple in $R(A_2^{\text{light}}, A_3)$ can join with at most $\sqrt{\text{OUT}}$ tuples in $R(A_1, A_2^{\text{light}})$, implied by the definition of light values in A_2. So this step has load $O\left(\frac{N + N_1}{p} + \frac{N \cdot \sqrt{\text{OUT}}}{p}\right) = O(L)$. Moreover, the size of $R(A_1, A_3)$ is also bounded by $N \cdot \sqrt{\text{OUT}}$. For Step (3.2), we invoke the induction hypothesis with $N_1 = N \cdot \sqrt{\text{OUT}}$, hence bounding its load by (the big-Oh of)

$$\frac{(N \cdot N \cdot \sqrt{\text{OUT}})^{1/3} \cdot \text{OUT}^{1/2}}{p^{2/3}} + \frac{(N \cdot \sqrt{\text{OUT}} \cdot N_n)^{1/3} \cdot \text{OUT}^{1/3}}{p^{2/3}}$$
$$+ \frac{N^{2/3} \cdot \text{OUT}^{2/3}}{p^{2/3}} + \frac{N + N \cdot \text{OUT}^{1/2} + N_n + \text{OUT}}{p} \leq L.$$

Summing over all steps, we conclude the proof of Lemma 3.

5 STAR QUERIES

In this section, we study the star query

$$\sum_B R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \cdots \bowtie R_n(A_n, B),$$

which can be considered as another way to generalize the matrix multiplication problem.

As with the line query, the basic idea of handling a star query is to recursively reduce n until $n = 2$. However, an inherent difficulty for the star 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 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. However, 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.

Below, we design a constant-round MPC algorithm whose load can be bounded by (a function of) OUT. The idea 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.

As before, we first remove all dangling tuples. Then we compute the query in the following 3 steps.

Step 1: Compute data statistics. We first compute for each value $b \in \text{dom}(B)$ its degree in relation $R_i(A_i, B)$ for $i \in [n]$, denoted as $d_i(b)$, using reduce-by-key primitive. For each $b \in \text{dom}(B)$, we compute a permutation $\phi_b : [n] \to [n]$ such that $d_{\phi_b(i)}(b) \leq d_{\phi_b(j)}(b)$ for any $1 \leq i < j \leq n$. Note that we can determine ϕ_b for all b's by sorting all values in $\text{dom}(B)$, breaking ties by $d_i(b)$.

For a permutation ϕ of $[n]$, let $B_\phi = \{b \in \text{dom}(B) : \phi_b = \phi\}$. This decomposes $\text{dom}(B)$ into $n!$ subsets, which leads to $n!$ subqueries, each defined by a distinct permutation ϕ:

$$Q_\phi = \sum_B \sigma_{B \in B_\phi} R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \cdots \bowtie R_n(A_n, B).$$

We handle each subquery separately by applying the steps below.

Step 2: Reduce to matrix multiplication. Note that we can allocate p servers for each subquery. As the number of subqueries is a constant, the number of servers used in this step is $O(p)$.

Consider Q_ϕ defined by some permutation ϕ. We first compute the following two intermediate joins by the Yannakakis algorithm:

$$R_\phi(\mathcal{A}^{\text{odd}}, B) = \bowtie_{i \in [n]: i \text{ is odd}} \sigma_{B \in B_\phi} R(A_{\phi(i)}, B)$$
$$R_\phi(\mathcal{A}^{\text{even}}, B) = \bowtie_{j \in [n]: j \text{ is even}} \sigma_{B \in B_\phi} R(A_{\phi(j)}, B)$$

where $\mathcal{A}^{\text{odd}} = \{A_{\phi(i)} : i \in [n], i \text{ is odd}\}$ and $\mathcal{A}^{\text{even}} = \{A_{\phi(j)} : j \in [n], j \text{ is even}\}$ partitions the output attributes $\{A_1, A_2, \ldots, A_n\}$. Then, we reduce the subquery Q_ϕ into a matrix multiplication

$$\sum_B R_\phi(\mathcal{A}^{\text{odd}}, B) \bowtie R_\phi(\mathcal{A}^{\text{even}}, B),$$

and compute it by invoking the output-sensitive algorithm in Section 3.2.

Now we analyze the load of this step. The key is to bound the sizes of $R_\phi(\mathcal{A}^{\text{odd}}, B)$ and $R_\phi(\mathcal{A}^{\text{even}}, B)$. We first state the following two facts:

Lemma 5. *For any $b \in \text{dom}(B)$, $\prod_{j=1}^n d_j(b) \leq \text{OUT}$.*

This lemma follows immediately from the fact that for each value $b \in \text{dom}(B)$, $\bowtie_{i \in [n]} \sigma_{B=b} R_i(A_i, B)$ is a subset of final output result.

Lemma 6. *Let $d_1 \le d_2 \le \cdots \le d_n$ be a set of positive integers with $\prod_{i=1}^{n} d_i = \lambda$. Then $\max\{\prod_{i\in I} d_i, \prod_{j\in J} d_j\} \le \sqrt{\lambda}$ where $I = \{i \in [n-2] : i \text{ is odd}\}$ and $J = \{j \in [n-2] : j \text{ is even}\}$.*

Proof. If n is odd. It suffices to show $\prod_{i\in[n-2]:i \text{ is odd}} d_i \le \sqrt{\lambda}$ since the other term derived on even-index elements is dominated. Observe that

$$\prod_{i\in[n-2]:i \text{ is odd}} d_i \le N_{n-1} \cdot \prod_{j\in[n-2]:j \text{ is even}} d_j$$

Moreover, $\left(\prod_{i\in[n-2]:i \text{ is odd}} d_i\right) \cdot \left(N_{n-1} \cdot \prod_{j\in[n-2]:j \text{ is even}} d_j\right) = \prod_{i=1}^{n-1} d_i \le \lambda$, so there must be $\prod_{i\in[n-2]:i \text{ is odd}} d_i \le \sqrt{\lambda}$. The case with even n can be proved symmetrically. □

Assume n is odd (and case of even n is simpler). We can bound the size of $R_\phi(\mathcal{A}^{\text{odd}}, B)$ as follows (the size of $R_\phi(\mathcal{A}^{\text{even}}, B)$ can be bounded similarly):

$$\begin{aligned}|R_\phi(\mathcal{A}^{\text{odd}}, B)| &= \sum_{b\in B_\phi} \prod_{i\in[n]:i \text{ is odd}} d_{\phi(i)}(b) \\ &\le \left(\max_{b\in B_\phi} \prod_{i\in[n-2]:i \text{ is odd}} d_{\phi(i)}(b)\right) \cdot \sum_{b\in B_\phi} d_{\phi(n)}(b) \\ &\le \max_{b\in B_\phi} \sqrt{\prod_{i\in[n]} d_{\phi(i)}(b)} \cdot |R(A_{\phi(n)}, B)| \le \sqrt{\text{OUT}} \cdot N,\end{aligned}$$

where the second inequality is implied by Lemma 6 and the last one is implied by Lemma 5.

Thus, the load of computing $R_\phi(\mathcal{A}^{\text{odd}}, B)$ and $R_\phi(\mathcal{A}^{\text{even}}, B)$ is bounded by $O(\frac{N}{p} + \frac{N\cdot\sqrt{\text{OUT}}}{p})$. Plugging $N_1 = N_2 = N \cdot \sqrt{\text{OUT}}$ to Lemma 2, the load of computing the matrix multiplication is $O\left((\frac{N\cdot\sqrt{\text{OUT}}}{p})^{2/3} \cdot \text{OUT}^{1/3} + \frac{N\cdot\sqrt{\text{OUT}}}{p}\right) = O\left((\frac{N\cdot\text{OUT}}{p})^{2/3} + \frac{N\cdot\sqrt{\text{OUT}}}{p}\right)$.

Step 3: Aggregate all subqueries. Finally, as with the line query, the $n!$ subqueries from the star query may produce results with the same value on the output attributes, so we need to run the reduce-by-key primitive to aggregate all the results. As each subquery produces at most OUT results, the total number of results is still $O(\text{OUT})$. So this step has a load of $O(\frac{\text{OUT}}{p})$.

Summing up the costs of all the steps above, we obtain the following result.

Theorem 5. *For a star query with input size N and output size* OUT*, there is an algorithm computing it in $O(1)$ rounds with load $O\left((\frac{N\cdot\text{OUT}}{p})^{2/3} + \frac{N\cdot\text{OUT}^{1/2}}{p} + \frac{N+\text{OUT}}{p}\right)$.*

6 STAR-LIKE QUERIES

Before tackling general tree queries, in this section we investigate *star-like queries*, which can be seen as a combination of line and star queries. Star-like queries will be the building block of tree queries studies in Section 7. A star-like query is shown in Figure 1. More precisely, a star-like query consists of n line queries that share a common non-output attribute B. We call these line queries the *arms* of the query. Let $\mathcal{T}_i = (\mathcal{V}_i, \mathcal{E}_i)$ be the i-th arm. One end of the arm must be B, while the other end is denoted as A_i, which must be an output attribute. The remaining attributes are denoted as $C_{i1}, C_{i2}, \ldots$, which are non-output attributes. Note that a star-like query degenerates to a line query if $n = 2$, and to a star query if each arm contains only one relation.

As with star queries, we will design an algorithm that is oblivious to OUT. We first remove all dangling tuples. Then we carry out the following steps.

Step 1: Compute data statistics. Consider any $i \in [n]$. In the algorithm for star queries, we first computed $d_i(b)$ for each value $b \in \text{dom}(B)$, which is the number of values in $\text{dom}(A_i)$ that can join with b. When the arm $\mathcal{T}_i$ has more than one relation, we cannot obtain $d_i(b)$ accurately, but this can be estimated within a constant factor by applying the primitive in Section 2.2 to the arm. Below, we will not distinguish between $d_i(b)$ and its approximation, which will not affect our asymptotic analysis.

Similar as star queries, for each $b \in \text{dom}(B)$, we compute a permutation ϕ_b such that $d_{\phi_b(i)}(b) \le d_{\phi_b(j)}(b)$ for any $1 \le i \le j \le n$. Each permutation ϕ defines a subset of $\text{dom}(B)$ as $B_\phi = \{b \in \text{dom}(B) : \phi_b = \phi\}$. Here, we further divide each B_ϕ into two subsets:

$$B_\phi^{\text{small}} = \left\{b \in B_\phi : \prod_{i=1}^{n-1} d_{\phi(i)}(b) \le d_{\phi(n)}(b)\right\}$$

$$B_\phi^{\text{large}} = \left\{b \in B_\phi : \prod_{i=1}^{n-1} d_{\phi(i)}(b) > d_{\phi(n)}(b)\right\}$$

In this way, we can decompose the original query into the following subqueries:

$$Q_\phi^X = \sum_{\mathcal{V}-\{A_1,A_2,\cdots,A_n\}} \bowtie_{e\in\mathcal{E}} \sigma_{B\in B_\phi^X} R(e)$$

where ϕ can be any permutation of $[n]$ and X can be either small or large.

Note that the number of subqueries is $2n!$, which is still a constant. For each subquery, we remove its dangling tuples and apply the following reduction procedure, each using p servers.

Step 2: Compute Q_ϕ^{small}. We reduce Q_ϕ^{small} to a line query through the following two steps. An example is illustrated in Figure 1.

Step (2.1): Shrink all but one branch. For each $j \in \{\phi(1), \ldots, \phi(n-1)\}$ with $|\mathcal{E}_j| > 1$, we compute

$$R_\phi^{\text{small}}(A_j, B) = \sum_{\mathcal{V}_j-\{A_j,B\}} \bowtie_{e\in\mathcal{E}_j} \sigma_{B\in B_\phi^{\text{small}}} R(e)$$

by invoking the Yannakakis algorithm. More specifically, for each $l = 1, 2, \cdots, h-1$, the algorithm iteratively computes $R(A_j, C_{j,l+1}) = \sum_{C_{jl}} R(A_j, C_{jl}) \bowtie R(C_{jl}, C_{j,l+1})$, and at last computes $R_\phi^{\text{small}}(A_j, B) = \sum_{C_{jh}} R(A_j, C_{jh}) \bowtie \sigma_{B\in B_\phi^{\text{small}}} R(C_{jh}, B)$.

Step (2.2): Reduce to line query. We first compute

$$R_\phi(\mathcal{A}^{\text{small}}, B) = \bowtie_{j\in\{\phi(1),\phi(2),\cdots,\phi(n-1)\}} R_\phi^{\text{small}}(A_j, B)$$

where $\mathcal{A}^{\text{small}} = \{A_j : j \in \{\phi(1), \phi(2), \cdots, \phi(n-1)\}\}$ using the Yannakakis algorithm. Regarding $\mathcal{A}^{\text{small}}$ as a "combined" attribute, the query now becomes a line query:

$$\sum_{\mathcal{V}_{\phi(n)}-\{A_{\phi(n)}\}} R_\phi(\mathcal{A}^{\text{small}}, B) \bowtie \left(\bowtie_{e\in\mathcal{E}_\phi(n)} R(e)\right),$$

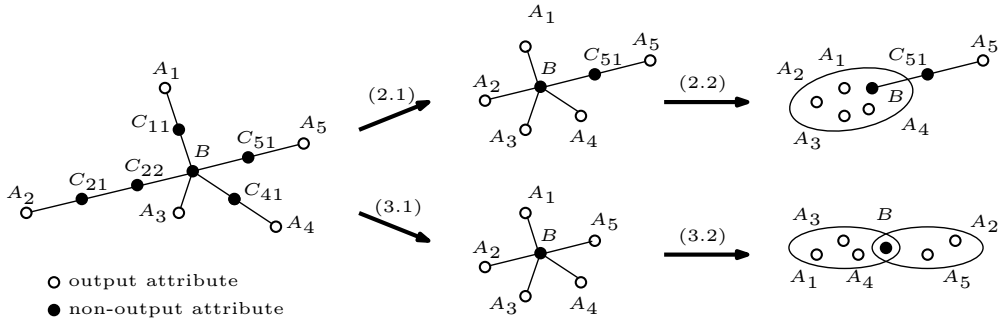

Figure 1: An illustration of a star-like query (on the left) and how the subquery Q_ϕ with $\phi = (4, 2, 1, 3, 5)$ is reduced in Step 2 and Step 3 (on the right). For example on $\mathcal{T}_2$, $\mathcal{V}_2 = \{A_2, C_{21}, C_{22}, B\}$ and $\mathcal{E}_2 = \{(A_2, C_{21}), (C_{21}, C_{22}), (C_{22}, B)\}$.

on which we invoke the algorithm in Section 4.1.

Step 3. Compute Q_ϕ^{large}. We reduce Q_ϕ^{large} to a matrix multiplication through two steps. An example is illustrated in Figure 1.

Step (3.1): Shrink all branches. For each $j \in [n]$ with $|\mathcal{E}_j| > 1$, we compute

$$R_\phi^{\text{large}}(A_j, B) = \sum_{\mathcal{V}_j - \{A_j, B\}} \bowtie_{e \in \mathcal{E}_j} \sigma_{B \in B_\phi^{\text{large}}} R(e)$$

using the Yannakakis algorithm, in the same manner as we computed $R_\phi^{\text{small}}(A_j, B)$ in Step (2.1).

Step (3.2): Reduce to matrix multiplication. Define $I = \{\phi(n), \phi(n-3), \phi(n-6), \cdots, \phi(n - 3 \cdot \lfloor \frac{n}{3} \rfloor)\}$ and $J = [n] - I$. We use the Yannakakis algorithm to compute the following queries:

$$R_\phi(\mathcal{A}_I, B) = \bowtie_{i \in I} R_\phi^{\text{large}}(A_i, B)$$
$$R_\phi(\mathcal{A}_J, B) = \bowtie_{j \in J} R_\phi^{\text{large}}(A_j, B)$$

where $\mathcal{A}_I = \{A_i : i \in I\}$ and $\mathcal{A}_J = \{A_j : j \in J\}$. Then we reduce it to a matrix multiplication problem as $\sum_B R_\phi(\mathcal{A}^I, B) \bowtie R_\phi(\mathcal{A}^J, B)$. Note that the even-odd strategy for star query would fail here. In fact, the size of $R_\phi(A_i, B)$ can be as large as $N \cdot \sqrt{\text{OUT}}$. If applying the same argument, the size of $|R_\phi(\mathcal{A}^{\text{odd}}, B)|$ or $|R_\phi(\mathcal{A}^{\text{even}}, B)|$ can only be bounded by $N \cdot \text{OUT}$. So, we will not obtain any improvement over the standard Yannakakis algorithm. Moreover, for the reduced matrix multiplication, we propose a more fine-grained way to handle it, instead of invoking the output-sensitive algorithm in Section 3.2 directly. The reason will be clearer in the analysis.

Step (3.3): Uniformize matrix multiplications. We first compute for each value $b \in B_\phi$ its degree in relation $R_\phi(\mathcal{A}_I, B)$, by using the reduce-by-key primitive. We further divide values in B_ϕ into $k = O(\log N)$ disjoint groups $B_\phi^1, B_\phi^2, \cdots, B_\phi^k$ such that

$$B_\phi^i = \{b \in B_\phi^{\text{large}} : 2^{i-1} \le |\sigma_{B=b} R_\phi(\mathcal{A}_I, B)| < 2^i\}.$$

Note that there are at most N distinct values in attribute B, i.e., $\sum_\phi \sum_i |B_\phi^i| \le N$.

Each group B_ϕ^i defines a matrix multiplication problem as

$$\sum_B \left(\sigma_{B \in B_\phi^i} R_\phi(\mathcal{A}_I, B)\right) \bowtie \left(\sigma_{B \in B_\phi^i} R_\phi(\mathcal{A}_J, B)\right).$$

For each tuple $t \in R_\phi(\mathcal{A}_I, B)$ or $t \in R_\phi(\mathcal{A}_J, B)$, we attach an index i to it if $\pi_B t \in B_\phi^i$ using the multi-search primitive. Let $N_\phi^i = |\sigma_{B \in B_\phi^i} R_\phi(\mathcal{A}_I, B)| + |\sigma_{B \in B_\phi^i} R_\phi(\mathcal{A}_J, B)|$. The statistics of N_ϕ^i's can be computed using the reduce-by-key primitive.

Step (3.4): Compute all matrix multiplications. Sort all tuples in $R_\phi(\mathcal{A}_I, B)$ and $R_\phi(\mathcal{A}_J, B)$ by the attached index.

If all tuples with index i land on a single server, just let the server locally compute this matrix multiplication induced by B_ϕ^i. Otherwise, the tuples with index i land on more than 2 servers. Note that the number of such indexes is at most p. For such an index i, we allocate

$$p_i = \left\lceil \frac{|B_\phi^i|}{N} \cdot p + \frac{N_\phi^i}{\sum_j N_\phi^j} \cdot p \right\rceil$$

servers to compute the matrix multiplication induced by B_ϕ^i. The total number of servers allocated is

$$\sum_{i=1}^{k} p_i = \sum_{i=1}^{k} \left(\frac{|B_\phi^i|}{N} \cdot p + \frac{N_\phi^i}{\sum_j N_\phi^j} \cdot p + 1 \right) \le 3 \cdot p = O(p).$$

Then, all the matrix multiplications are computed in parallel.

Step 4: Reduce all aggregate results. At last, we use the reduce-by-key primitive to aggregate the results generated by all the subqueries above.

6.1 Analysis

Similarly, we will actually prove a slightly tighter bound of our algorithm, as stated in the following lemma.

Lemma 7. *For a star-like query $Q_y(\mathcal{R})$, let $N = \max_{e \in \mathcal{E}: e \cap y = \emptyset} |R_e|$, $N' = \max_{e \in \mathcal{E}: e \cap y \neq \emptyset} |R_e|$, and* OUT *be the output size. There is an algorithm computing it in $O(1)$ rounds with load $\widetilde{O}(L)$ w.h.p., where* $L = \frac{(NN')^{1/3} \cdot \text{OUT}^{1/2}}{p^{2/3}} + \frac{N'^{2/3} \cdot \text{OUT}^{1/3}}{p^{2/3}} + \frac{N \cdot \text{OUT}^{2/3}}{p} + \frac{N + N' + \text{OUT}}{p}$.

We prove the lemma by induction on n. Note that the base case $n = 2$ is just Lemma 3. For a star-like query with $n > 2$, we consider the load of our algorithm in each of the four steps separately.

Step 1 has linear load, since it only involves primitives. In Step 4, the input size of reduce-by-key can be bounded by $\widetilde{O}(\text{OUT})$ since there are $O(\log N)$ subqueries in total and each subquery produces at most OUT results. So this step has a load of $\widetilde{O}(\frac{\text{OUT}}{p})$. Before analysing the cost of Step 2 and Step 3, we mention some important observations as below.

Lemma 8. *For every $b \in \text{dom}(B)$, $\prod_{j=1}^{n} d_j(b) \le \text{OUT}$.*

Lemma 9. *For any permutation ϕ of $[n]$, and every $b \in \text{dom}(B)$,*

(1) *if $b \in B_\phi^{small}$, then $\prod_{i=1}^{n-1} d_{\phi(i)}(b) \le \sqrt{\text{OUT}}$;*

(2) *if $b \in B_\phi^{large}$, then $d_{\phi(i)}(b) \le \sqrt{\text{OUT}}$ for any $i \in [n]$;*

The proof of Lemma 8 directly follows the fact that for each value $b \in \text{dom}(B)$, the $\pi_{A_1,A_2,\cdots,A_n} \bowtie_{e\in\mathcal{E}_i} \sigma_{B=b} R_e$ is a subset of final output result, whose size is exactly $\prod_{j=1}^{n} d_j(b)$. The proof of Lemma 9 directly follows Lemma 8 and the definition of ϕ.

Lemma 10. *In a reduced star-like subquery Q_ϕ^{small}, on any $\mathcal{T}_j = (\mathcal{V}_j, \mathcal{E}_j)$ with $j \in \{\phi(1), \phi(2), \cdots, \phi(n-1)\}$, every $c \in \text{dom}(C)$ for $C \in \mathcal{V}_j$ can be joined with at most $\sqrt{\text{OUT}}$ values in $\text{dom}(A_j)$.*

Proof. By contradiction, assume there exists some $c \in \text{dom}(C)$ that can be joined with more than $\sqrt{\text{OUT}}$ values in $\text{dom}(A_j)$. Consider any value $b \in B_\phi^{\text{small}}$ that can be joined with c. Such a value always exists otherwise c won't exist in a reduced query. This implies that b can be joined with more than $\sqrt{\text{OUT}}$ values in $\text{dom}(A_j)$, coming to a contradiction of Lemma 9. □

Lemma 11. *Let $d_1 \le d_2 \le \cdots \le d_n$ be a set of positive integers with $\prod_{i=1}^{n} d_i = \lambda$. If $d_n \le \sqrt{\lambda}$, $\max\{\prod_{i\in I} d_i, \prod_{j\in J} d_j\} \le \lambda^{2/3}$ where $I = \{n, n-3, n-6, \cdots, n-3\cdot\lfloor\frac{n}{3}\rfloor\}$ and $J = [n] - I$.*

Proof. For simplicity, we define the configuration over all possible inputs as $\mathcal{D} = \{(d_1, d_2, \cdots, d_n) : d_i \in \mathbb{Z}^+, \prod_{i=1}^{n} d_i = \lambda, d_1 \le d_2 \le \cdots \le d_n, d_n \le \sqrt{\lambda}\}$. It suffices to show (1) $\max_{\mathcal{D}} \prod_{i\in I} d_i \le \lambda^{2/3}$; and (2) $\max_{\mathcal{D}} \prod_{j\notin I} d_j \le \lambda^{2/3}$.

For (1), let $d^* = \arg\max_{\mathcal{D}} \prod_{i\in I} d_i$. We first claim that $d^*_{n+2-3k} = d^*_{n+1-3k} = d^*_{n-3k}$ for any $k \in [\lfloor\frac{n}{3}\rfloor]$. If there exists some $k \in [\lfloor\frac{n}{3}\rfloor]$ such that $d^*_{n+2-3k} > d^*_{n-3k}$, we construct another solution d' such that $d'_i = d^*_i$ for any $i \in [n] - \{n+2-3k, n+1-3k, n-3k\}$ and $d'_j = (d^*_{n+2-3k} d^*_{n+1-3k} d^*_{n-3k})^{1/3}$ for any $j \in \{n+2-3k, n+1-3k, n-3k\}$. Then $\prod_{i\in I} d^*_i < \prod_{i\in I} d'_i$, contradicting to the optimality of d^*.

Let i as the smallest integer such that $d^*_i > 1$. By our observation above, $(n-i) \bmod 3 = 0$. Without loss of generality, assume $i = n-3k$ for $k \in [\lfloor\frac{n}{3}\rfloor]$. We next claim that $k = 1$. If not, let $d^*_n = z, d^*_{n-1} = d^*_{n-2} = d^*_{n-3} = y$ and $d^*_{n+2-3k} = d^*_{n+1-3k} = d^*_{n-3k} = x$. Since $i < n-3$, $x \le y \le z$. We transform it into another solution in the following way: if $xy \le z$, set $x' = 1, y' = xy, z' = z$; otherwise, set $x' = 1, y' = z' = x^{3/4} \cdot y^{3/4} \cdot z^{1/4}$. In this way, we don't change any property of d^* such that $d^*_j = 1$ holds for any $j < n-3$.

We are left with a simpler problem involving only four integers $d^*_n, d^*_{n-1}, d^*_{n-2}, d^*_{n-3} \ge 1$. If $d^*_{n-4} = 1$, $\prod_{i\in I} d^*_i = d^*_n \le \sqrt{\lambda}$; otherwise, $\prod_{i\in I} d^*_i = d^*_{n-4} d^*_n \le d^*_n \cdot (\frac{\lambda}{d^*_n})^{1/3} = \lambda^{1/3} \cdot (d^*_n)^{2/3} \le \lambda^{2/3}$.

The (2) can be proved similarly. Let $d^* = \arg\max_{\mathcal{D}} \prod_{j\in J} d_j$. We can also show that $d^*_j = 1$ for any $j < n-2$. Then, $\prod_{j\in J} d^*_j = d^*_{n-1} d^*_{n-2} \le (d^*_n d^*_{n-1} d^*_{n-2})^{2/3} \le \lambda^{2/3}$. □

In Step (2.1), the load complexity of computing $R_\phi^{\text{small}}(A_j, B)$ for each $j \in \{\phi(1), \phi(2), \cdots, \phi(n-1)\}$ follows the same analysis. Consider an arbitrary $j \in \{\phi(1), \phi(2), \cdots, \phi(n-1)\}$. The size of full join $R(A_j, C_{jl}) \bowtie R(C_{jl}, C_{j,l+1})$ is bounded by $N \cdot \sqrt{\text{OUT}}$, implied by Lemma 10. Then, the Yannakakis algorithm has a load of $O\left(\frac{N\cdot\sqrt{\text{OUT}}}{p}\right)$. In Step (2.2), the size of $R_\phi(\mathcal{A}^{\text{small}}, B)$ is also bounded by $N \cdot \sqrt{\text{OUT}}$ since there are at most N distinct values in B_ϕ and each one has its degree smaller than $\sqrt{\text{OUT}}$ in $R_\phi(\mathcal{A}^{\text{small}}, B)$, implied by Lemma 10. By plugging $N, N_1 = N \cdot \sqrt{\text{OUT}}, N_n = N'$ into Lemma 3, the reduced line query can be computed in $O(1)$ rounds with load $O\left(\frac{N^{2/3}\cdot\text{OUT}^{2/3}}{p^{2/3}} + \frac{(NN')^{1/3}\cdot\text{OUT}^{1/2}}{p^{2/3}} + \frac{N\cdot\text{OUT}^{1/2}}{p} + \frac{N+N'+\text{OUT}}{p}\right) = O(L)$, which also dominates the load complexity of Step 2.

In Step (3.1), the Yannakakis algorithm has a load of $O\left(\frac{N\cdot\sqrt{\text{OUT}}}{p}\right)$, following the similar argument for Step (2.1). With Lemma 11, we can bound the size of $R_\phi(\mathcal{A}_I, B)$ with $I = \{\phi(n), \phi(n-3), \phi(n-6), \cdots, \phi(n-3\cdot\lfloor\frac{n}{3}\rfloor)\}$ as follows (the size of $R_\phi(\mathcal{A}_J, B)$ can be bounded similarly):

$$|R_\phi(\mathcal{A}_I, B)| \le \sum_{b\in B_\phi} \prod_{j\in I} d_j(b) \le |B_\phi| \cdot \text{OUT}^{2/3} \le N \cdot \text{OUT}^{2/3}.$$

So, the Yannakakis algorithm in Step (3.2) has a load of $O\left(\frac{N\cdot\text{OUT}^{2/3}}{p}\right)$. Also, the primitives in Step (3.3) and (3.4) incur load $O\left(\frac{N\cdot\text{OUT}^{2/3}}{p}\right)$. With Lemma 8, we observe on every B_ϕ^i that

$$|\sigma_{B\in B_\phi^i} R_\phi(\mathcal{A}_I, B)| \cdot |\sigma_{B\in B_\phi^i} R_\phi(\mathcal{A}_J, B)| \le |B_\phi^i| \cdot 2^i \cdot |B_\phi^i| \cdot \frac{\text{OUT}}{2^{i-1}}.$$

Plugging to Lemma 2, computing all matrix multiplications in Step (3.4) incurs a load of $O\left(\max_i \frac{(|B_\phi^i|\cdot|B_\phi^i|\cdot\text{OUT})^{1/3}\cdot\text{OUT}^{1/3}}{p_i^{2/3}} + \frac{N_\phi^i}{p_i}\right) = O\left(\frac{N\cdot\text{OUT}^{2/3}}{p}\right)$, since $\sum_j N_\phi^j = |R_\phi(\mathcal{A}_I, B)| + |R_\phi(\mathcal{A}_J, B)| \le 2N \cdot \text{OUT}^{2/3}$. Thus, Step 3 has a load of $O\left(\frac{N\cdot\text{OUT}^{2/3}}{p}\right) = O(L)$.

Over all steps, this algorithm has a load of $\widetilde{O}(L)$. We have completed the induction proof for Lemma 7.

7 TREE QUERIES

Finally, we are ready to tackle general tree queries with arbitrary output attributes, by building upon our algorithm for star-like queries.

As a preprocessing step, we remove all dangling tuples. Then we reduce the tree by iteratively removing a relation R_e if (1) e contains a single attribute; or (2) there is a non-output attribute v that appears only in e. Let e' be any relation such that $e \cap e' \ne \emptyset$. We can remove R_e by attaching annotations of R_e to $R_{e'}$, i.e., for each tuple $t' \in R_{e'}$, apply the update $w(t') \leftarrow w(t') \cdot \sum_{t\in R_e : \pi_{e\cap e'} t = \pi_{e\cap e'} t'} w(t)$, which can be done by the reduce-by-key and multi-search primitives. All these operations incur linear load. It should be clear that after all such R_e's have been removed, every leaf attribute is an output attribute. Please see Figure 2 for an example.

Next, we break the query at every non-leaf output attribute. This decomposes the tree into a number of *twigs*. Observe that in each twig, all the output attributes are exactly the leaves (see Figure 2).

Note that it is sufficient to show how to compute a twig. After computing all the twigs, the remaining attributes are all output attributes, so we can just use the standard Yannakakis algorithm to compute the full join of all the twigs, which has load $O(\frac{\text{OUT}}{p})$.

7.1 The algorithm

Note that twig queries are generalizations of star-like queries, which in turn are generalizations of line and star queries. Figure 2 shows some example of twig queries, among which twig 4 is in a form more general than star-like queries. Our idea is for tackling such twig queries is to remove the non-output attributes recursively until it becomes a star-like query.

For a twig but non-star-like query $\mathcal{T} = (\mathcal{V}, \mathcal{E})$, we introduce the notion of *skeleton*. Let $\mathcal{V}^* \subsetneq \mathcal{V}$ be the set of attributes appearing in more than 2 relations. Note that $\mathcal{V}^* \subseteq \bar{\mathbf{y}}$. Moreover, $|\mathcal{V}^*| \geq 2$; otherwise, $\mathcal{T}$ is a star-like query. Consider the subtree derived by $\mathcal{V}^*$, denoted as $\mathcal{T}_{\mathcal{V}^*}$, such that $v \in \mathcal{V}$ or $e \in \mathcal{E}$ is included by $\mathcal{T}_{\mathcal{V}^*}$ if it lies on the path between any pair of $B, B' \in \mathcal{V}^*$. For each leaf $B \in \mathcal{V}^*$, let e_B be the edge incident to B in $\mathcal{T}_{\mathcal{V}^*}$. Removing e_B will separate $\mathcal{T}$ into two connected subtrees. The one containing B is exactly a star-like query since only B may appear in more than 2 relations, denoted as $\mathcal{T}_B = (\mathcal{V}_B, \mathcal{E}_B)$. We just replace $\mathcal{T}_B$ by B. Note that this procedure can be applied to all leaves in $\mathcal{T}^*_{\mathcal{V}}$ independently since the subtrees to be removed have no intersection. The resulted subtree is the *skeleton* of $\mathcal{T}$, denoted as $\mathcal{T}_S$. Let $\mathcal{S} \subseteq \mathcal{V}$ be the set of leaves in $\mathcal{T}_S$. Observe that $S \cap \bar{\mathbf{y}}$ is exactly the set of leaves in $\mathcal{T}_{\mathcal{V}^*}$. An example is illustrated in Figure 3.

Step 1: Compute data statistics. Consider an attribute $B \in \mathcal{S} \cap \bar{\mathbf{y}}$. We can allocate p servers to each attribute B and compute the following statistics for each value $b \in \text{dom}(B)$. The total number of servers used is at most $O(p)$.

For each value $b \in \text{dom}(B)$, we first estimate the number of values in $\text{dom}(A)$ for $A \in \mathcal{V}_B \cap \mathbf{y}$ that can be joined with b, denoted as $d_A(b)$, which can be done by invoking the algorithm in Section 2.2. Let $x(b) = \prod_{A \in \mathcal{V}_B \cap \mathbf{y}} d_A(b)$, denoting the number of combinations over attributes $\mathcal{V}_B \cap \mathbf{y}$ that can be joined with b.

Meanwhile, we hope to estimate the number of combinations over attributes $\mathbf{y} - \mathcal{V}_B$ that can join with b. This is actually more general than computing OUT for a star query, which is still unknown. Below, we describe a procedure UnderEstimateOutTree (Algorithm 1) to obtain an underestimate for each value $b \in \text{dom}(B)$, which suffices for our needs. For consistency, define $x(a) = 1$ for each value $a \in \text{dom}(A)$ if $A \in \mathcal{S} \cap \mathbf{y}$. Note that line 9 of the algorithm can computed by the reduce-by-key primitive. When the algorithm terminates, the value $y(b)$ is an underestimate for the number of combinations over attributes $\mathbf{y} - \mathcal{V}_B$ that can join with b. The relationship on the $x(b)$'s and $y(b)$'s is stated by Lemma 12

Lemma 12. *For any pair of attributes $B \in \mathcal{S} \cap \bar{\mathbf{y}}, B' \in \mathcal{S} - \{B\}$, if $b \in \text{dom}(B)$ can join with $b' \in \text{dom}(B')$, then $y(b) \geq x(b')$.*

Proof. Rename the attributes lying on the path between B', B as $C_1(B'), C_2, \cdots, C_k, C_{k+1}(B)$ successively. We identify any combination $((c_1(b'), c_2, \cdots, c_{k-1}, c_k(b)) \in R(C_1, C_2) \bowtie R(C_2, C_3) \cdots \bowtie R(C_{k-1}, C_k)$, which can always be found since b', b can be joined. We claim that $y(c_i) \geq y(c_1)$ holds for any $i \in [k]$ in the call of UnderEstimateOutTree($\mathcal{T}_S$, B). This holds for $i = 1$ trivially. By induction, assume $y(c_i) \geq y(c_1)$. The invariant of $y(c_{i+1}) \geq y(c_i)$ always holds in line 9, so $y(c_{i+1}) \geq y(c_1)$. As $y(c_1) = y(b') = x(b')$, then $y(b) \geq x(b')$ as desired. □

Algorithm 1: EstimateOutTree(B)

```
1  Reorganize T_S to root at B;
2  foreach C ∈ S − {B} do
3      foreach value c ∈ dom(C) do
4          y(c) ← x(c);
5  Mark all vertices in S − {B} as "visited";
6  while T_S contains at least one unvisited non-leaf vertex do
7      Find a unvisited vertex C whose children are all visited;
8      for c ∈ dom(C) do in parallel
9          y(c) ← ∏_{C': C' is a child of C} max_{c':(c',c)∈R(C,C')} y(c');
10     Mark C as "visited";
```

Step 2: Divide and Conquer. Consider any attribute $C \in \mathcal{S}$. The value $c \in \text{dom}(C)$ is *heavy* if $x(c) > y(c)$, and *light* otherwise. Depending on the values being heavy or light on each attribute in $\mathcal{S} \cap \bar{\mathbf{y}}$, we can decompose the original query into $O(2^{|\mathcal{S} \cap \bar{\mathbf{y}}|})$ subqueries. Note that every value $a \in \text{dom}(A)$ for $A \in \mathcal{S} \cap \mathbf{y}$ is light, following the fact that $x(a) = 1 \leq y(a)$. We allocate p servers to each subquery and apply the following procedure in parallel.

Consider an arbitrary subquery. We remove its dangling tuples first. For each light non-output attribute $B \in \mathcal{S} \cap \bar{\mathbf{y}}$, we compute

$$Q_B = \sum_{\mathcal{V}_B \cap \bar{\mathbf{y}}} \bowtie_{e \in \mathcal{E}_B} R_e$$

by the Yannakakis algorithm, and materialize the result as a new relation $R(B, \mathcal{V}_B \cap \mathbf{y})$. Computing Q_B is similar to that of reducing Q^{small}_{ϕ} to a star-like query in Section 6, and we omit the details here. Then we replace $\mathcal{T}_B$ with a new edge $(B, \mathcal{V}_B \cap \mathbf{y})$ by regarding $\mathcal{V}_B \cap \mathbf{y}$ as a "combined" attribute. At last, we run this algorithm recursively on the residual tree query. An illustration is given in Figure 4.

The correctness of this step, i.e., $\mathcal{T}$ can be reduced to a smaller query, is implied by Lemma 13. Intuitively, for each subquery there is at most heavy attribute in $\mathcal{S} \cap \bar{\mathbf{y}}$ due to the constraint in Lemma 12. By showing $|\mathcal{S} \cap \bar{\mathbf{y}}| \geq 2$, we prove the existence of light attributes.

Lemma 13. *For each subquery, after dangling tuples being removed, at least one attribute in $\mathcal{S} \cap \bar{\mathbf{y}}$ is light.*

Proof. We first show that each subquery has at most one heavy attribute in $\mathcal{S} \cap \bar{\mathbf{y}}$. By contradiction, assume there exists a pair of heavy attributes $B, B' \in \mathcal{S} \cap \bar{\mathbf{y}}$. If this subquery has non-empty output result, there must exist a pair of values $b \in \text{dom}(B), b' \in \text{dom}(B')$ that appears together in an output result. By Lemma 12, $x(b) > y(b) \geq x(b')$ and $x(b') > y(b') \geq x(b)$, coming to a contradiction.

Next we show that $|\mathcal{S} \cap \bar{\mathbf{y}}| \geq 2$. As $\mathcal{T}$ is not a star-like query, at least two attributes appear in more than 2 relations, i.e., $|\mathcal{V}^*| \geq 2$.

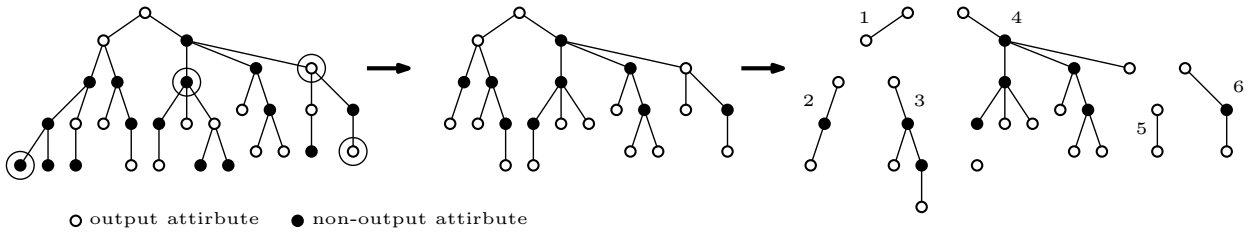

Figure 2: A tree query (on the left), the reduced tree (in the middle) and its twigs (on the right). There are 6 twigs: 1 and 5 contain only one relation whose vertices are output attributes, 2 and 6 are matrix multiplications, 3 is a star-like query and 4 is a twig which will be tackled in Section 7.1.

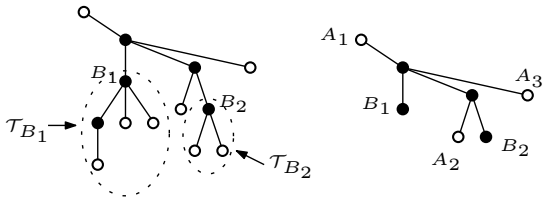

Figure 3: An illustration of twig 4 and its skeleton $\mathcal{T}_S$ with $\mathcal{S} = \{A_1, A_2, A_3, B_1, B_2\}$, $\mathcal{S} \cap \mathbf{y} = \{A_1, A_2, A_3\}$, $\mathcal{S} \cap \bar{\mathbf{y}} = \{B_1, B_2\}$.

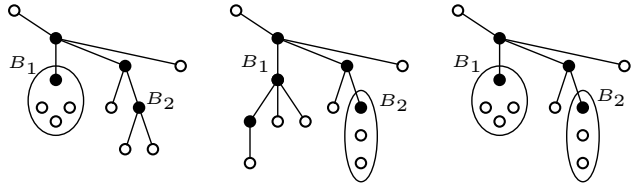

Figure 4: Tree query 4 after reduction. From left to right are subqueries with (B_1: light, B_2: heavy), (B_1: heavy, B_2: light), (B_1:light, B_2: light).

The derived subtree $\mathcal{T}_{\mathcal{V}^*}$ contains at least two leaves, i.e., $|\mathcal{S} \cap \bar{\mathbf{y}}| \geq 2$. These two arguments together will prove the existence of light attribute as desired. □

Step 3: Reduce all aggregate results. In the last step, we use reduce-by-key to aggregate the results generated by all subqueries.

7.2 The analysis

For the load analysis, we will actually prove a slightly tighter bound of our algorithm, as stated in the following lemma.

Lemma 14. *For a twig query $Q_{\mathbf{y}}(\mathcal{R})$, let $N = \max_{e \in \mathcal{E}: e \cap \mathbf{y} = \emptyset} |R_e|$, $N' = \max_{e \in \mathcal{E}: e \cap \mathbf{y} \neq \emptyset} |R_e|$, and* OUT *be the output size. Our algorithm can compute it in $O(1)$ rounds with load $\widetilde{O}(L)$ w.h.p., where $L = \frac{(NN')^{1/3} \cdot \text{OUT}^{1/2}}{p^{2/3}} + \frac{N'^{2/3} \cdot \text{OUT}^{1/3}}{p^{2/3}} + \frac{N \cdot \text{OUT}^{2/3}}{p} + \frac{N+N'+\text{OUT}}{p}$.*

We prove the lemma by induction on n. The base case when $\mathcal{T}$ is a relation or a star-like query is Lemma 7. For a general twig query, we analyze the load of our algorithm in each step above.

Step 1 has linear load of $O\left(\frac{N+N'}{p}\right)$ since it only involves primitives. Step 3 incurs a load of $O\left(\frac{\text{OUT}}{p}\right)$ since there are $O(1)$ subqueries in total and each has at most OUT results. Before analyzing Step 2, we first mention an important property, as stated Lemma 15.

Lemma 15. *For any subquery, after dangling tuples being removed, if $B \in \mathcal{S}$ is light, then $x(b) \leq \sqrt{\text{OUT}}$ for each value $b \in \text{dom}(B)$.*

Proof. For each $b \in \text{dom}(B)$, $x(b) \cdot y(b) \leq \text{OUT}$ since $\pi_{\mathbf{y}} \bowtie_{e \in \mathcal{E}} \sigma_{B=b} R_e$ is a subset of final result. By the definition of light values, i.e., $x(b) \leq y(b)$, we have $x(b) \leq \sqrt{\text{OUT}}$. □

Consider an arbitrary subquery in Step 2. With Lemma 15, we can bound the size of $R(B, \mathcal{V}_B \cap \mathbf{y})$ by $N \cdot \sqrt{\text{OUT}}$ for each light attribute $B \in \mathcal{S} \cap \bar{\mathbf{y}}$, since there are at most N values in $\text{dom}(B)$ and each one can be joined with at most $\sqrt{\text{OUT}}$ distinct combinations over $\mathcal{V}_B \cap \mathbf{y}$. So the Yannakakis algorithm has a load of $O\left(\frac{N+N'}{p} + \frac{N \cdot \sqrt{\text{OUT}}}{p}\right) = O(L)$. On the residual query, by plugging $N, \max\{N', N \cdot \sqrt{\text{OUT}}\}$ to hypothesis induction, it can be easily checked that its load is still bounded by $O(L)$.

This completes the induction proof for Lemma 14. By setting $N' = N$, we obtain the following result.

Theorem 6. *For an arbitrary tree join-aggregate query with input size N and output size* OUT*, there is an algorithm computing it in $O(1)$ rounds with load $\widetilde{O}\left(\frac{N \cdot \text{OUT}^{2/3}}{p} + \frac{N+\text{OUT}}{p}\right)$ w.h.p.*

8 OPEN PROBLEMS

While our matrix multiplication algorithm is optimal, we do not have any lower bounds for more general queries. The major difficulty in proving lower bounds for queries that involve more than two relations is that it is not clear which "elementary products" must be computed. For these queries, one can use the distributive law of the semiring to save computation and communication costs, as our algorithms have done. In fact, it is due to the same reason that there is no lower bound for such queries even in the (sequential) RAM model. Note that the distributive law does not help with matrix multiplication, and one can argue that all the elementary products must be computed.

Other open problems include join-aggregate and join-project queries over relations of arity more than two, or when the query is cyclic.

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] 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.

[3] 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.

[4] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. *Randomization and Approximation Techniques in Computer Science*, pages 952–952, 2002.

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

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

[7] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2007.

[8] 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.

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

[10] 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.

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

[12] J.-W. Hong and H.-T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM, 1981.

[13] X. Hu, Y. Tao, and K. Yi. Output-optimal parallel algorithms for similarity joins. In *Proc. ACM Symposium on Principles of Database Systems*, 2017.

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

[15] 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.

[16] 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.

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

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

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

[20] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303. ACM, 2014.

[21] R. Pagh and M. Stöckel. The input/output complexity of sparse matrix multiplication. In *Proc. European Symposium on Algorithms*, 2014.

[22] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

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

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

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