Output-optimal Parallel Algorithms for Similarity Joins*

Xiao Hu
HKUST

Yufei Tao
University of Queensland

Ke Yi
HKUST

ABSTRACT

Parallel join algorithms have received much attention in recent years, due to the rapid development of massively parallel systems such as MapReduce and Spark. In the database theory community, most efforts have been focused on studying worst-optimal algorithms. However, the worst-case optimality of these join algorithms relies on the hard instances having very large output sizes. In the case of a two-relation join, the hard instance is just a Cartesian product, with an output size that is quadratic in the input size.

In practice, however, the output size is usually much smaller. One recent parallel join algorithm by Beame et al. [8] has achieved *output-optimality*, i.e., its cost is optimal in terms of both the input size and the output size, but their algorithm only works for a 2-relation equi-join, and has some imperfections. In this paper, we first improve their algorithm to true optimality. Then we design output-optimal algorithms for a large class of similarity joins. Finally, we present a lower bound, which essentially eliminates the possibility of having output-optimal algorithms for any join on more than two relations.

Categories and Subject Descriptors

F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems

Keywords

Parallel computation, similarity joins, output-sensitive algorithms

1. INTRODUCTION

The similarity join problem is perhaps one of the most extensively studied problems in the database and data mining literature. Numerous variants exist, depending on the metric space and the distance function used. Let $\text{dist}(\cdot,\cdot)$ be a distance function. Given two point sets R_1 and R_2 and a threshold $r \geq 0$, the similarity join problem asks to find all pairs of points $x \in R_1, y \in R_2$, such that $\text{dist}(x,y) \leq r$. In this paper, we will be mostly interested in the ℓ_1, ℓ_2, and ℓ_∞ distances, although some of our results (the one based on LSH) can be extended to other distance functions as well.

1.1 The computation model

Driven by the rapid development of massively parallel systems such as MapReduce [14], Spark [29], and many other systems that adopt very similar architectures, there have also been resurrected interests in the theoretical computer science community to study algorithms in such massively parallel models. One popular model that has often been used to study join algorithms in particular, is the *massively parallel computation* model (MPC) [1, 2, 3, 7, 8, 20, 21, 22].

In the MPC model, data is initially partitioned arbitrarily across p servers that are connected by a complete network. Computation proceeds in rounds. In each round, each server first receives messages from other servers (sent in a previous round, if there is one), does some local computation, and then sends messages to other servers, which will be received by them at the beginning of the next round. The complexity of the algorithm is measured first by the number of rounds, then the *load*, denoted L, which is the maximum message size received by any server in any round. Initial efforts were mostly spent on understanding what can be done in a single round of computation [2, 7, 8, 21, 22], but recently, more interests have been given to multi-round (but still a constant) algorithms [1, 3, 20, 21], since new main memory based systems, such as Spark, tend to have much lower overhead per round than previous systems like Hadoop. Meanwhile, this puts more emphasis on minimizing the load, to ensure that the local memory at each server is never exceeded.

One thing that we want to point out, which was never explicitly stated in the prior work on the MPC model, is that the MPC model is essentially the same as Valiant's *bulk synchronous processing* model (BSP) [28]. More precisely, it is the same as the CREW version of the BSP [15], where a server may broadcast a message to multiple servers. The incoming message size at each server is still limited in the CREW BSP model, as in the MPC model.

The MPC model, as well as the CREW BSP model, allows broadcasts. This is often justified by the fact that in some systems, broadcasts may indeed be more efficient than sending the message to each destination individually. Very recently, it has been shown [18] that any algorithm in the

*Xiao Hu and Ke Yi are supported by HKRGC under grants GRF-16211614 and GRF-16200415.

PODS'17, May 14-19, 2017, Chicago, IL, USA

© 2017 ACM. ISBN 978-1-4503-4198-1/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3034786.3056110

CREW BSP model can be simulated in the standard BSP model without using any broadcasts by just increasing the number of rounds and the load by a constant factor, provided that $\text{IN} > p^{1+\epsilon}$, where IN is the input size and $\epsilon > 0$ is any small constant. This provides a stronger justification of using broadcasts.

1.2 Previous join algorithms in the MPC model

All prior work on join algorithms in the MPC model has focused on equi-joins, and has mostly been concerned with the worst case. Notably, the *hypercube* algorithm [2] computes the equi-join between two relations of size N_1 and N_2, respectively, with load $L = \tilde{O}(\sqrt{N_1 N_2/p})$[1]. Note that this is optimal in the worst case, as the output size can be as large as $N_1 N_2$, when all tuples share the same join key and the join degenerates into a Cartesian product. Since each server can only produce $O(L^2)$ join results in a round[2] if the load is limited to L, all the p servers can produce at most $O(pL^2)$ join results in a constant number of rounds. Thus, producing $N_1 N_2$ results needs at least a load of $L = \Omega(\sqrt{N_1 N_2/p})$. Note that this lower bound argument is assuming *tuple-based* join algorithms, i.e., the tuples are atomic elements that must be processed and communicated in their entirety. They can be copied but cannot be broken up or manipulated with bit tricks. To produce a join result, all tuples (or their copies) that make up the join result must reside at the same server when the join result is output. However, the server does not have to do any further processing with the result, such as sending it to another server. The same model has also been used in [7, 8, 21].

However, on most realistic data sets, the join size is nowhere near the worst case. Suppose the join size is OUT. Applying the same argument as above, one would hope to get a load of $\tilde{O}(\sqrt{\text{OUT}/p})$. Such a bound would be *output-optimal*. Of course, this is not entirely possible, as OUT can even be zero, so a more reasonable target would be $L = \tilde{O}(\sqrt{\text{OUT}/p} + \text{IN}/p)$, where $\text{IN} = N_1 + N_2$ is the total input size. This is exactly the goal of this work, although in some cases, we have not achieved this ideal input-dependent term $\tilde{O}(\text{IN}/p)$ exactly. Note that we are still doing worst-case analysis, i.e., we do not make any assumptions on the input data and how it is distributed on the p servers initially. We merely use OUT as an additional parameter to measure the complexity of the algorithm.

There are some previous join algorithms that use both IN and OUT to measure the complexity. Afrati et al. [1] gave an algorithm with load $O(\text{IN}^w/\sqrt{p} + \text{OUT}/\sqrt{p})$, where w is the *width* of the join query, which is 1 for any acyclic query, including a two-relation join. However, both terms $O(\text{OUT}/\sqrt{p})$ or $O(\text{IN}/\sqrt{p})$ are far from optimal.

Beame et al. [8] classified the join values into being heavy and light. For a join value v, let $R_i(v)$ be the set of tuples in R_i with join value v. Then a join value v is *heavy* if $|R_1(v)| \geq N_1/p$ or $|R_2(v)| \geq N_2/p$. Then they gave an algorithm with load

$$\tilde{\Theta}\left(\sqrt{\frac{\sum_{\text{heavy } v} |R_1(v)| \cdot |R_2(v)|}{p}} + \frac{\text{IN}}{p}\right). \qquad (1)$$

In fact, this bound can be equivalently written as $\tilde{\Theta}(\sqrt{\text{OUT}/p} + \text{IN}/p)$. Note that

$$\begin{aligned}\text{OUT} &= \sum_v |R_1(v)| \cdot |R_2(v)| \\ &= \sum_{\text{heavy } v} |R_1(v)| \cdot |R_2(v)| + \sum_{\text{light } v} |R_1(v)| \cdot |R_2(v)|,\end{aligned}$$

so (1) is upper bounded by $\tilde{O}(\sqrt{\text{OUT}/p} + \text{IN}/p)$. Meanwhile, it is also lower bounded by $\tilde{\Omega}(\sqrt{\text{OUT}/p} + \text{IN}/p)$: First, it is clearly in $\tilde{\Omega}(\text{IN}/p)$. Second, it is also in $\tilde{\Omega}(\sqrt{\text{OUT}/p})$ since

$$\sum_{\text{light } v} |R_1(v)| \cdot |R_2(v)| \leq \frac{N_1 N_2}{p} \leq \frac{\text{IN}^2}{p},$$

hence

$$\begin{aligned}(1) &= \tilde{\Omega}\left(\sqrt{\frac{\text{OUT} - \text{IN}^2/p}{p}} + \frac{\text{IN}}{p}\right) \\ &= \tilde{\Omega}\left(\sqrt{\frac{\text{OUT} - \text{IN}^2/p}{p} + \frac{\text{IN}^2}{p^2}}\right) = \tilde{\Omega}(\sqrt{\text{OUT}/p}).\end{aligned}$$

Therefore, their algorithm is output-optimal, but up to a polylog factor, due to the use of hashing (the hidden polylog factor is $O(\log^2 p)$). Their analysis relies on the uniform hashing assumption, i.e., the hash function distributes each distinct key to the servers uniformly and independently. It is not clear whether more realistic hash functions, such as universal hashing, could still work. They also assume that each server knows the entire set of heavy join values and their frequencies, namely, all the $|R_i(v)|$'s that are larger than N_i/p, for $i = 1, 2$.

Note that equi-join is a special case of similarity joins with $r = 0$. There are previously no algorithms in the MPC model for similarity joins with $r > 0$, except computing the full Cartesian product of the two relations with load $O(\sqrt{N_1 N_2/p})$, which is not output-optimal.

As a remark, there exists a general reduction [21] that converts MPC join algorithms into I/O-efficient counterparts under the *enumerate version* [26] of the external memory model [4], where each result tuple only needs to be seen in memory, as opposed to being reported in the disk. A nice application of the reduction has been demonstrated for the *triangle enumeration problem*, where an MPC algorithm [21] is shown to imply an EM algorithm matching the I/O lower bound of [26] up to a logarithmic factor.

1.3 Our results

We start with an improved algorithm for computing the equi-join between two relations, i.e., a degenerated similarity join with $r = 0$. We improve upon the algorithm of Beame et al. [8] in the following aspects: (1) Our algorithm does not assume any prior statistical information about the data, such as the heavy join values and their frequencies. (2) The load of our algorithm is exactly $O(\sqrt{\text{OUT}/p} + \text{IN}/p)$ tuples, without any extra logarithmic factors. (3) Our algorithm is deterministic. The only price we pay is that the number of rounds increases from 1 to $O(1)$. This algorithm is described in Section 3.

While the $O(\sqrt{\text{OUT}/p})$ term is optimal by the tuple-based argument above, prior work did not show why the input-dependent term $O(\text{IN}/p)$ is necessary. In fact, the load

[1] The $\tilde{O}$ notation suppresses polylogarithmic factors.

[2] Technically, this is true under the condition $L = \Omega(\frac{N_1+N_2}{p})$, but as will be proved in this paper, the condition indeed holds even just to decide whether the join result is empty.

has often been written in form of $O(\mathrm{IN}/p^{1-\delta})$ for $\delta \in [0,1]$, implicitly assuming that $O(\mathrm{IN}/p)$ is the best load possible, i.e., every tuple has to be communicated at least once. Indeed, if OUT is not a parameter, the worst-case input is always when the output size is maximized, i.e., a full Cartesian product for two-relation joins, or the AGM bound [6] for multi-way joins. In this case, the simple tuple-based argument above already leads to a lower bound higher than $\Omega(\mathrm{IN}/p)$, so this is not an issue. However, when the output size is restrained to be a parameter OUT, these worst-case constructions do not work anymore; and it is not clear why $O(\mathrm{IN}/p)$ load is necessary. Indeed, if $\mathrm{OUT} = 1$, then the tuple-based argument above yields a meaningless lower bound of $\Omega(1/p)$. To complete the picture, we provide a lower bound showing that even if $\mathrm{OUT} = O(1)$, computing the equi-join between two relations requires $\Omega(\mathrm{IN}/p)$ load, by resorting to strong results from communication complexity.

The main results in this paper, however, are on similarity joins with $r > 0$. In this regard, We achieve the following results under various distance functions.

1. For ℓ_1/ℓ_∞ distance in constant dimensions, we give a deterministic algorithm with load
$$O\left(\sqrt{\frac{\mathrm{OUT}}{p}} + \frac{\mathrm{IN}}{p} \cdot \log^{O(1)} p\right),$$
i.e., the output-dependent term is optimal, while the input-dependent term is away from optimal by a polylogarithmic factor, which depends on the dimensionality.

2. For ℓ_2 distance in d dimensions, we give a randomized algorithm with load
$$O\left(\sqrt{\frac{\mathrm{OUT}}{p}} + \mathrm{IN}/p^{\frac{d}{2d-1}} + p^{\frac{d}{2d-1}} \log p\right).$$
Again, the term $O\left(\sqrt{\frac{\mathrm{OUT}}{p}}\right)$ is output-optimal. The input-dependent term $O(\mathrm{IN}/p^{\frac{d}{2d-1}})$ is worse than the ℓ_1/ℓ_∞ case, due to the non-orthogonal nature of the ℓ_2 metric, but it is always better than $O(\mathrm{IN}/\sqrt{p})$, which is the load for computing the full Cartesian product.

3. In high dimensions, we provide an LSH-based algorithm with load
$$O\left(\sqrt{\frac{\mathrm{OUT}}{p^{1/(1+\rho)}}} + \sqrt{\frac{\mathrm{OUT}(cr)}{p}} + \frac{\mathrm{IN}}{p^{1/(1+\rho)}}\right),$$
where $\mathrm{OUT}(cr)$ is the output size if the distance threshold is enlarged to cr for some constant $c > 1$, and $0 < \rho < 1$ is the quality measure of the hash function used, which depends only on c and the distance function. Similarly, the term $O(\mathrm{IN}/p^{1/(1+\rho)})$ is always better than that for computing the Cartesian product, although output-optimality here is only with respect to $\mathrm{OUT}(cr)$ instead of OUT, due to the approximation nature of LSH.

All the algorithms run in $O(1)$ rounds, under the mild assumption $\mathrm{IN} > p^{1+\epsilon}$, where $\epsilon > 0$ is any small constant. Note that the randomized output-optimal algorithm in [8] for equi-joins has an implicit assumption that $\mathrm{IN} \geq p^2$, since there are $\Theta(p)$ heavy join values, so each server has load at least $\Omega(p)$ to store these values and their frequencies. We acknowledge that in practice, $\mathrm{IN} \geq p^2$ is a very reasonable assumption. Our desire to relax this to $\mathrm{IN} > p^{1+\epsilon}$ is more from a theoretical point of view, namely, achieving the minimum requirement for solving these problem in $O(1)$ rounds and optimal load. Indeed, Goodrich [15] has shown that, if $\mathrm{IN} = p^{1+o(1)}$, then even computing the "or" of IN bits requires $\omega(1)$ rounds under load $O(\mathrm{IN}/p)$.

Finally, we turn to multi-way joins. The only known multi-way equi-join algorithm in the MPC model that has a term related to OUT is the algorithm in [1] mentioned in Section 1.2. However, that term is $O(\mathrm{OUT}/\sqrt{p})$, which is almost quadratically larger than the output-optimal term $O(\sqrt{\mathrm{OUT}/p})$ we have achieved above. We show that, unfortunately, such an output-optimal term is not achievable, even for the simplest multi-way equi-join, a 3-relation chain join $R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D)$. More precisely, in Section 7 we show that if any tuple-based algorithm computing this join has a load in the form of
$$L = O\left(\frac{\mathrm{IN}}{p^\alpha} + \sqrt{\frac{\mathrm{OUT}}{p}}\right),$$
for some constant α, then we must have $\alpha \leq 1/2$, provided $\mathrm{IN} \log^2 \mathrm{IN} = \Omega(p^3)$. On the other hand, the algorithm in [21] can already compute any 3-relation chain join with $\tilde{O}(\mathrm{IN}/\sqrt{p})$ load. This means that it is meaningless to introduce the output-dependent term $O(\sqrt{\mathrm{OUT}/p})$.

2. PRELIMINARIES

We first introduce the following primitives in the MPC model. We will assume $\mathrm{IN} > p^{1+\epsilon}$ where $\epsilon > 0$ is any small constant.

2.1 Sorting

The sorting problem in the MPC model is defined as follows. Initially, IN elements are distributed arbitrarily on p servers, which are labeled $1, 2, \ldots, p$. The goal is to redistribute the elements so that each server has IN/p elements in the end, while any element at server i is smaller than or equal to any element at server j, for any $i < j$. By realizing that the MPC model is the same as the BSP model, we can directly invoke Goodrich's optimal BSP sorting algorithm [15]. His algorithm has load $L = \Theta(\mathrm{IN}/p)$ and runs in $O(\log_L \mathrm{IN}) = O(\log_L(pL)) = O(\log_L p)$ rounds. When $\mathrm{IN} > p^{1+\epsilon}$, this is $O(1)$ rounds.

2.2 Multi-numbering

Suppose each tuple has a key. The goal of the *multi-numbering* problem is, for each key, assign consecutive numbers $1, 2, 3, \ldots$ to all the tuples with the same key.

We solve this problem by reducing it to the *all prefix-sums* problem: Given an array of elements $A[1], \ldots, A[\mathrm{IN}]$, compute $S[i] = A[1] \oplus \cdots \oplus A[i]$ for all $i = 1, \ldots, \mathrm{IN}$, where $\oplus$ is any associative operator. Goodrich et al. [16] gave an algorithm in the BSP model for this problem that uses $O(\mathrm{IN}/p)$ load and $O(1)$ rounds.

To see how the multi-numbering problem reduces to the all prefix-sums problem, we first sort all tuples by their keys; ties are broken arbitrarily. The i-th tuple in the sorted order will produce a pair (x, y), which will act as $A[i]$. For each tuple that is the first of its key in the sorted order, we produce the pair $(0, 1)$; otherwise, we produce $(1, 1)$. Note that we need another round of communication to determine whether each tuple is the first of its key, in case that its predecessor resides on another server.

Then we define the operator $\oplus$ as

$$(x_1, y_1) \oplus (x_2, y_2) = (x_1x_2, y),$$

where

$$y = \begin{cases} y_1 + y_2, & \text{if } x_2 = 1; \\ y_2, & \text{if } x_2 = 0. \end{cases}$$

Consider any $(x, y) = A[i] \oplus \cdots \oplus A[j]$. Intuitively, $x = 0$ indicates that $A[i], \ldots, A[j]$ contain at least one tuple that is the first of its key, while y counts the number of tuples in $A[i], \ldots, A[j]$ whose key is the same as that of $A[j]$. It is an easy exercise to check that $\oplus$ is associative, and after solving the all prefix-sums problem, $S[i]$ is exactly the number of tuples in front of the i-th tuple that has the same key (including the i-th tuple itself), which solves the multi-numbering problem as desired.

2.3 Sum-by-key

Suppose each tuple is associated with a key and a weight. The goal of the *sum-by-key* problem is to compute, for each key, the total weight of all the tuples with the same key.

This problem can be solved using essentially the same approach as for the multi-numbering problem. First sort all the N tuples by their keys. As above, each tuple will produce a pair (x, y). Now, x still indicates whether this tuple is the first of its key, but we just set y to be the weight associated with the tuple. After we have solved the all prefix-sums problem on these pairs, the last tuple of each key has the total weight for this key. Again, we need another round to identify the last tuple of each key, by checking each tuple's successor.

After the algorithm above finishes, for each key, exactly one tuple knows the total weight for the key, i.e., the last one in the sorted order. In some cases, we also need every tuple to know the total weight for the tuple's own key. To do so, we invoke the multi-numbering algorithm, so that the last tuple of each key also knows the number of tuples with that key. From this number, we can compute exactly the range of servers that hold all the tuples with this key. Then we broadcast the total weight to these servers.

2.4 Multi-search

The *multi-search* problem is defined as follows. Given N_1 distinct keys and N_2 queries, where $\mathrm{IN} = N_1 + N_2$, for each query, find its predecessor, i.e., the largest key that is no larger than the query. The multi-search algorithm given by Goodrich et al. [16] is randomized, with a small probability exceeding $O(\mathrm{IN}/p)$ load. In fact, this problem can also be solved using all prefix-sums, which results in a deterministic algorithm with load $O(\mathrm{IN}/p)$: We first sort all the keys and queries together. Then for each key k, define its corresponding $A[i]$ as itself; for each query, define its $A[i] = -\infty$; define $\oplus = \max$. Then it should be obvious that $S[i]$ is the predecessor of the corresponding query.

2.5 Cartesian product

The hypercube algorithm [2, 8] is a randomized algorithm that computes the Cartesian product of two sets R_1 and R_2. Suppose the two sets have size N_1 and N_2, respectively. This algorithm has a load of $O((\sqrt{N_1N_2/p} + \mathrm{IN}/p) \log^2 p)$ with probability $1 - 1/p^{O(1)}$. The extra log factors are due to the use of hashing. We observe that if the elements in each set are numbered as $1, 2, 3, \ldots,$ then we can achieve deterministic and perfect load balancing.

Without loss of generality, assume $N_1 \le N_2$. As in the standard hypercube algorithm, we arrange the p servers into a $d_1 \times d_2$ grid such that $d_1d_2 = p$. If an element in R_1 gets assigned a number x, then we send it to all servers in the ($x \mod d_1$)-th row of the grid; for an element in R_2, we send it to all servers in the ($x \mod d_2$)-th column of the grid. Each server then produces all pairs of elements received. By setting (1) $d_1 = \sqrt{\frac{pN_1}{N_2}}, d_2 = \sqrt{\frac{pN_2}{N_1}}$, if $N_2 \le pN_1$; or (2) $d_1 = 1$, $d_2 = p$, if $N_2 > pN_1$, the load is $O(\sqrt{N_1N_2/p} + \mathrm{IN}/p)$.

2.6 Server allocation

In many of our algorithms, we decompose the problem into up to p subproblems, and allocate the p servers appropriately, with subproblem j having $p(j)$ servers, where $\sum_j p(j) \le p$. Thus, each subproblem needs to know which servers have been allocated to it. This is trivial if $\mathrm{IN} \ge p^2$, as we can collect all the $p(j)$'s to one server, do a central allocation, and broadcast the allocation results to all servers, as is done in [8]. When we only have $\mathrm{IN} \ge p^{1+\epsilon}$, some more work is needed to ensure $O(\mathrm{IN}/p)$ load.

More formally, in the *server allocation* problem, each tuple has a subproblem id j, which identifies the subproblem it belongs to (the j's do not have to be consecutive), and $p(j)$, which is the number of servers allocated to subproblem j. The goal is to attach each tuple a range $[p_1(j), p_2(j)]$, such that the ranges of different subproblems are disjoint and $\max_j p_2(j) \le p$.

We again resort to all prefix-sums. First sort all tuples by their subproblem id. For each tuple, define its corresponding $A[i] = p(j)$ if it is the first tuple of subproblem j, and 0 otherwise. After running all prefix-sums, for each tuple, we set its $p_2(j) = S[i]$, and $p_1(j) = S[i] - p(j) + 1$.

3. EQUI-JOIN

We start by revisiting the equi-join problem between 2 relations, $R_1 \bowtie R_2$. Let N_1 and N_2 be the sizes of R_1 and R_2, respectively; set $\mathrm{IN} = N_1 + N_2$. First, if $N_1 > pN_2$ or $N_2 > pN_1$, the problem can be trivially solved by broadcasting the smaller relation to all servers, incurring a load of $O(\min\{N_1, N_2\})$. Below, we assume $N_1 \le N_2 \le pN_1$, and describe an algorithm that achieves the following result.

THEOREM 1. *There is a deterministic algorithm that computes the equi-join between 2 relations in $O(1)$ rounds with load $O\left(\sqrt{\frac{\mathrm{OUT}}{p}} + \frac{\mathrm{IN}}{p}\right)$. It does not assume any prior statistical information about the data.*

3.1 The algorithm

Our algorithm can be seen as an MPC version of sort-merge-join.

Step (1) Computing OUT

Consider each distinct join value v. Let $R_i(v)$ be the set of tuples in R_i with join value v; let $N_i(v) = |R_i(v)|$. Note that

$\text{OUT} = \sum_v N_1(v)N_2(v)$. We first use the sum-by-key algorithm to compute all the $N_i(v)$'s, i.e., each tuple in $R_i(v)$ is considered to have key v and weight 1. Recall that after the sum-by-key algorithm, for each v, exactly one tuple knows $N_i(v)$. We sort all such tuples by the key v. Then we add up all the $N_1(v)N_2(v)$'s, which can also be done by sum-by-key (just that the key is the same for all tuples).

Step (2) Computing $R_1 \bowtie R_2$

Next, we compute the join, i.e., the Cartesian products $R_1(v) \times R_2(v)$ for all v. Sort all tuples in both R_1 and R_2 by the join value. Consider each join value v. If all tuples in $R_1(v) \cup R_2(v)$ land on the same server, their join results can be emitted directly, so we only need to deal with the case when they land on 2 or more servers. There are at most p such v's. For each such v, we allocate

$$p_v = \left\lceil p \cdot \frac{N_1(v)}{N_1} + p \cdot \frac{N_2(v)}{N_2} + p \cdot \frac{N_1(v)N_2(v)}{\text{OUT}} \right\rceil$$

servers and compute the Cartesian product $R_1(v) \times R_2(v)$. Note that we need a total of $O(p)$ servers; scaling down the initial p can ensure that at most p servers are needed. Here, we also need the server allocation primitive to allocate servers to these subproblems accordingly. Finally, to be able to use the deterministic version of the hypercube algorithm, the elements in each $R_i(v)$ need to be assigned consecutive numbers, which can be achieved by running the multi-numbering algorithm, treating each distinct join value v as a key. It can be easily verified that the load is $O\left(\sqrt{\frac{N_1(v)N_2(v)}{p_v}} + \frac{N_1(v)}{p_v} + \frac{N_2(v)}{p_v}\right) = O\left(\sqrt{\frac{\text{OUT}}{p}} + \frac{N_1}{p} + \frac{N_2}{p}\right)$.

3.2 A matching lower bound

As argued in Section 1.2, the term $O(\sqrt{\text{OUT}/p})$ is optimal for any tuple-based algorithm. Below we show that the term $O(\text{IN}/p)$ is also necessary, even when $\text{OUT} = O(1)$.

THEOREM 2. *Any randomized algorithm that computes the equi-join between 2 relations in $O(1)$ rounds with a success probability more than $3/4$ must incur a load of at least $\Omega(\min\{N_1, N_2, \frac{\text{IN}}{p}\})$ bits.*

PROOF. We use a reduction from the *lopsided set disjointness* problem studied in communication complexity: Alice has $\leq n$ elements and Bob has $\leq m$ elements with $m > n$, both from a universe of size m, and the goal is to decide whether they have an element in common. It has been proved that in any multi-round communication protocol, either Alice has to send $\Omega(n)$ bits to Bob, or Bob has to send $\Omega(m)$ bits to Alice [27]. This holds even for randomized algorithms with a success probability larger than 3/4. We also note that in the hard instances used in [27], the intersection size of Alice's and Bob's sets is either 0 or 1.

The reduction works as follows. Without loss of generality, we assume $N_1 \leq N_2$. Given a hard instance of lopsided set disjointness, we create R_1 with $N_1 = n$ tuples, whose join values are the elements of Alice's set; create R_2 with $N_2 = m$ tuples, whose join values are the elements of Bob's set. Then solving the join problem also determines whether the two sets intersect or not, while OUT can only be 1 or 0.

Recall that in the MPC model, the adversary can allocate the input arbitrarily. We allocate R_1 and R_2 to the p servers as follows.

If $N_2 \leq p \cdot N_1$, we allocate Alice's set to $\frac{pN_2}{\text{IN}}$ servers and Bob's set to $\frac{pN_1}{\text{IN}}$ servers. Then Alice's servers must send $\Omega(N_1)$ bits to Bob's servers, which incurs a total load (across all rounds) of $\Omega(\text{IN}/p)$ bits per server, or Bob's servers must send $\Omega(N_2)$ bits to Alice's servers, also incurring a total load of $\Omega(\text{IN}/p)$ bits per server.

If $N_2 > p \cdot N_1$, then we allocate Bob's set to one server, and Alice's set to the other $p-1$ servers. Then Alice's servers will send $\Omega(N_1)$ bits to Bob's server, or receive $\Omega(N_2)$ bits, so the load is $\Omega(\min(N_1, N_2/p)) = \Omega(N_1)$. □

4. SIMILARITY JOIN UNDER ℓ_1/ℓ_∞

In this section, we study similarity joins under the ℓ_1 or the ℓ_∞ metric. We will actually study a more general problem, namely the *rectangles-containing-points* problem. Here we are given a set R_1 of N_1 points and a set R_2 of N_2 orthogonal rectangles. The goal is to return all pairs $(x, y) \in R_1 \times R_2$ such that $x \in y$. Note that a similarity join with ℓ_∞ metric is equivalent to a rectangles-containing-points problem where each side of the rectangles has length $2r$. A similarity join with ℓ_1 metric in d dimensions can be reduced to a similarity join with ℓ_∞ metric in 2^{d-1} dimensions, by noticing that for any vector $(x_1, \ldots, x_d) \in \mathbb{R}^d$,

$$\sum_{i=1}^{d} |x_i| = \max_{(z_2,\ldots,z_d)\in\{-1,1\}^{d-1}} |x_1 + z_2x_2 + \cdots + z_dx_d|.$$

In this section and the next, we will assume constant dimensions, so that 2^{d-1} is still a constant. We deal with the high-dimensional case in Section 6.

4.1 One dimension

We start by considering the one-dimensional case, i.e., the *intervals-containing-points* problem. We are given a set of N_1 points and a set of N_2 intervals. Set $\text{IN} = N_1 + N_2$. The goal to report all (point, interval) pairs such that the point is inside the interval. Below we describe how to solve this problem in $O(\sqrt{\text{OUT}/p} + \text{IN}/p)$ load. Note that as with the equi-join case, if $N_1 > p \cdot N_2$ or $N_2 > p \cdot N_1$, then the problem can be trivially and optimally solved with $O(\min(N_1, N_2))$ load.

Step (1) Computing OUT

As with the equi-join algorithm, we start by computing the value of OUT. First, we sort all the points and number them consecutively in the sorted order. Then, for each interval $I = [x, y]$, we find the predecessor points of x and y (multi-search). Taking the difference of the numbers assigned to the two predecessors will give us the number of points inside I. Finally, we add up all these counts to get OUT (special case of sum-by-key).

Step (2) Partially covered slabs

By setting $b = \sqrt{\text{OUT}/p} + \text{IN}/p$, we will ensure that the load of the remaining steps is $O(b)$. We sort all the points and divide them into *slabs* of size b. Note that there are at most p slabs. Consider each interval I in R_2. All the points inside I can be classified into two cases: (1) points that fall in a slab partially covered by I, and (2) points that fall in a slab fully covered by I. For example, in Figure 1, the join between I_1 and the points in the leftmost and the rightmost slab is considered under case (1), while the join between I_1 and the points in the two middle slabs is considered under case (2). Note that if an interval falls inside a slab completely, its join with the points in that slab is also considered under case (1), such as I_2 in Figure 1.

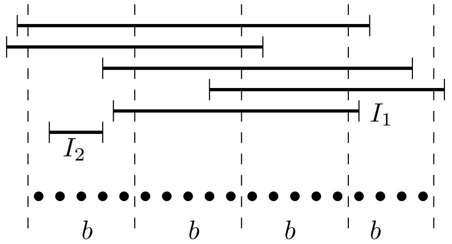

Figure 1: Partially covered and fully covered slabs.

In this step, we deal with the partially covered slabs. For each interval endpoint, we find which slab it falls into (multi-search). Then, for each slab, we compute the number of endpoints falling inside (sum-by-key). Consider each slab i. Suppose it contains $P(i)$ endpoints. We allocate $\left\lceil p \cdot \frac{P(i)}{N_2} \right\rceil$ servers to compute the join between the b points in the slab and the intervals with these $P(i)$ endpoints (note that we need $O(p)$ servers). We simply evenly allocate the $P(i)$ intervals to these servers (use multi-numbering to ensure balance), and broadcast all the b points to them. The load is thus

$$O\left(b + \frac{P(i)}{pP(i)/N_2}\right) = O(b).$$

Step (3) Fully covered slabs

Let $F(i)$ be the number of intervals fully covering a slab i. We can compute all the $F(i)$'s using all prefix-sums algorithm, as follows. If an interval fully covers slabs $i, \ldots, j$, we generate two pairs $(i, 1)$ and $(j + 1, -1)$. For each slab i, we generate a pair $(i + 0.5, 0)$. Then we sort all these (i, v) pairs by i and compute the prefix-sums on the v's.

Now, the full slabs can be dealt with using essentially the same algorithm. We allocate $p_i = \lceil p \cdot \frac{bF(i)}{\mathrm{OUT}} \rceil$ servers to compute the join (full Cartesian product) of the b points in slab i and the $F(i)$ intervals fully covering the slab. Since $\sum_i bF(i) \leq \mathrm{OUT}$, this requires at most $O(p)$ servers. We simply evenly allocate the $F(i)$ intervals to these servers and broadcast all the b points to them. The load is thus

$$O\left(b + \frac{F(i)}{pbF(i)/\mathrm{OUT}}\right) = O\left(b + \frac{\mathrm{OUT}}{pb}\right) = O(b).$$

THEOREM 3. *There is a deterministic algorithm for the intervals-containing-points problem that runs in $O(1)$ rounds with $O\left(\sqrt{\frac{\mathrm{OUT}}{p}} + \frac{\mathrm{IN}}{p}\right)$ load.*

4.2 Two and higher dimensions

Next, we generalize the algorithm above to two dimensions. Here we are given a set of N_1 points in 2D and a set of N_2 rectangles. Set $\mathrm{IN} = N_1 + N_2$. The goal is to report all (point, rectangle) pairs such that the point is inside the rectangle.

Step (1) Computing OUT

The first step is still to compute the output size OUT. We first sort all the x-coordinates, including those of the points and those of the left and right sides of the rectangles. Then each server defines a vertical slab, containing $O(\mathrm{IN}/p)$ x-coordinates. All joining (point, rectangle) pairs that land on the same server can be counted and output easily. For example, in Figure 2, the join results between σ_1 and all the points in slab 1 and slab 7 can be found by those two servers easily. This also includes rectangles that completely fall inside a slab, such as σ_2.

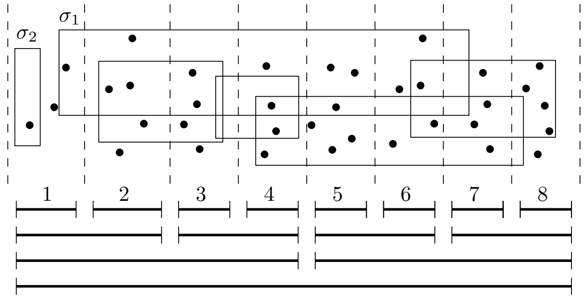

Figure 2: Rectangles-joining-points.

So we are left to count the joining pairs such that the x-projection of the rectangle fully spans the slab containing the point. We impose a binary hierarchy on the p slabs, and decompose each rectangle into $O(\log p)$ canonical slabs. For example, σ_1 in Figure 2 fully spans slabs 2–6, and it is decomposed into 3 canonical slabs: 2, 3–4, 5–6.

This results in a total of $O(N_2 \log p)$ canonical rectangles, each of which corresponds to one canonical slab. For each canonical slab s, we count its output size $\mathrm{OUT}(s)$, using step (1) of the 1D algorithm. To run all these instances in parallel, we allocate the servers as follows. For a canonical slab s that has $N_2(s)$ canonical rectangles and consists of $k(s)$ atomic slabs, we allocate $p_s = \left\lceil p \cdot \frac{k(s)\mathrm{IN}/p + N_2(s)}{\mathrm{IN} \log p} \right\rceil$ servers. As each atomic slab is covered by $O(\log p)$ canonical slabs, we have $\sum_s k(s) = O(p \log p)$. This step uses $O(p)$ servers, with each server having load $O(\frac{k(s)\mathrm{IN}/p + N_2(s)}{p_s}) = O(\frac{\mathrm{IN}}{p} \log p)$. To count the $N_2(s)$'s, we first count $F(i)$, the number of rectangles fully covering an atomic slab i. This is the same as the $F(i)$'s in the 1D case, can be counted with $O(\mathrm{IN}/p)$ load. Then for each atomic slab i, we produce $O(\log p)$ pairs $(s, F(i))$, one for each canonical slab s that contains i. This generates $O(p \log p)$ such pairs. Finally, we run sum-by-key on these pairs (using s as the key) to compute $N_2(s)$, with load $O(p \log p/p) = O(\log p) = O(\mathrm{IN}/p)$.

Note that this step always has load $O(\frac{\mathrm{IN}}{p} \log p)$ regardless of how large OUT is.

Step (2) Reduction to the 1D case

In this step, we compute, for each rectangle σ, all the result pairs produced by σ and the points in the slabs fully spanned by σ. We follow the same approach as in step (1), but will also need to take into account the output size of each canonical slab, i.e., $\mathrm{OUT}(s)$, when allocating the servers. More precisely, for a canonical slab s that has $N_2(s)$ canonical rectangles and consists of $k(s)$ contiguous slabs, we allocate

$$p_s = \left\lceil p \cdot \frac{\mathrm{OUT}(s)}{\mathrm{OUT}} + p \cdot \frac{k(s)\mathrm{IN}/p + N_2(s)}{\mathrm{IN} \log p} \right\rceil$$

servers, and invoke step (2) and (3) of the 1D algorithm (since $\mathrm{OUT}(s)$ is already computed) for all the canonical slabs in parallel. Note that a point knows which canonical slabs it falls into, hence which 1D instances it should participate in, from just its own slab number. For each canonical slab s, denote the size of its derived instance as $\text{IN}(s) = k(s)\text{IN}/p + N_2(s)$. Plugging $\text{OUT} = \text{OUT}(s), \text{IN} = \text{IN}(s), p = p_s$ into Theorem 3 yields the following result.

Theorem 4. *There is a deterministic algorithm for the rectangles-containing-points problem in 2D that runs in $O(1)$ rounds with $O\left(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{p}\log p\right)$ load.*

The algorithm can also be extended to higher dimensions using similar ideas, with an extra $O(\log p)$ factor for each dimension higher. We give the following result without proof.

Theorem 5. *There is a deterministic algorithm for the rectangles-containing-points problem in d dimensions that runs in $O(1)$ rounds with $O\left(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{p}\log^{d-1} p\right)$ load.*

5. SIMILARITY JOIN UNDER ℓ_2

In this section, we consider the similarity join between two point sets R_1 and R_2 under the ℓ_2 distance in d dimensions. We first use the *lifting transformation* [13] to convert the problem to the *halfspaces-containing-points* problem in $d+1$ dimensions. Consider any two points $(x_1, \dots, x_d) \in R_1$ and $(y_1, \dots, y_d) \in R_2$. The two points join under the ℓ_2 distance if

$$(x_1 - y_2)^2 + \dots + (x_d - y_d)^2 \le r^2,$$

or

$$x_1^2 + y_1^2 + \dots + x_d^2 + y_d^2 - 2x_1y_1 - \dots - 2x_dy_d - r^2 \ge 0.$$

We map the point $(x_1, \dots, x_d)$ to a point in $d+1$ dimensions: $(x_1, \dots, x_d, x_1^2 + \dots + x_d^2)$, and the point $(y_1, \dots, y_d)$ to a halfspace in $d+1$ dimensions (z_i's are the parameters):

$$-2y_1z_1 - \dots - 2y_dz_d + z_{d+1} + y_1^2 + \dots + y_d^2 - r^2 \ge 0.$$

We see that the two d-dimensional points join if and only if the corresponding $(d+1)$-dimensional halfspace contains the $(d+1)$-dimensional point.

Thus, in the following, we will study the halfspaces-containing-points problem. Given a set of N_1 points and a set of N_2 halfspaces in d dimensions, report all the (point, halfspace) pairs such that the point is inside the halfspace.

A key challenge in this problem, compared with the ℓ_1/ℓ_∞ case, is that there is no easy way to compute OUT, due to the non-orthogonal nature of the problem. Knowing the value of OUT is crucial in the previous algorithms, which is used to determine the right slab size, which in turn decides the load.

Our way to get around this problem is based on the observation that the load is determined by the output-dependent term only when OUT is sufficiently large. But in this case, a constant-factor approximation of OUT suffices to guarantee the optimal load asymptotically, and random sampling can be used to estimate OUT. Random sampling will not be effective when OUT is small (it is known that to decide whether $\text{OUT} = 1$ or 0 by sampling requires us to essentially sample the whole data set), but in that case, the input-dependent term will dominate the load, and we do not need to know the value of OUT anyway.

The following notion of a *θ-thresholded approximation* captures our needs, which will be used in the development of the algorithm.

Definition 1. *A θ-thresholded approximation of x is an estimate $\hat{x}$ such that: (1) if $x \ge \theta$, then $\frac{x}{2} < \hat{x} < 2x$; (2) if $x < \theta$, then $\hat{x} < 2\theta$.*

5.1 Useful tools from computational geometry

We will need the following tools from computational geometry. The first relates thresholded approximations with random sampling.

Theorem 6 ([23, 17]). *For any $q > 1$, let S be a random sample from a set P of n points with $|S| = O(q\log(q/\delta))$. Then with probability at least $1-\delta$, $n \cdot \frac{|\Delta \cap S|}{|S|}$ is a $\frac{n}{q}$-thresholded approximation of $|\Delta \cap P|$ for every simplex Δ.*

Next, we introduce the *partition tree*. In particular, we make use of the *b-partial partition tree* of Chan [11].

A b-partial partition tree on a set of points is a tree T with constant fanout, where each leaf stores at most b points, and each point is stored in exactly one leaf. Each node $v \in T$ (both internal nodes and leaf nodes) stores a cell $\Delta(v)$, which is a simplex that encloses all the points stored at the leaves below v. For any v, the cells of its children do not overlap. In particular, this implies that all the leaf cells are disjoint. Chan [11] presented an algorithm to construct a b-partial partition tree with the following properties.

Theorem 7 ([11]). *Given n points in $\mathbb{R}^d$ and a parameter $b < n/\log^{\omega(1)} n$, we can build a b-partial partition tree with $O(n/b)$ nodes, such that any hyperplane intersects $O((n/b)^{1-1/d})$ cells of the tree.*

Chan's construction only guarantees that each leaf cell contains at most b points but offers no lower bound, while we will need each leaf cell to have $\Theta(b)$ points. This can be easily achieved, though. Suppose Chan's b-partial partition tree has at most $c \cdot n/b$ leaves for some constant c. A leaf cell is *big* if it contains at least $b/2c$ points; otherwise *small*. Observe that there must be at least $n/2b$ big leaf cells. Then, we simply combine $O(1)$ small leaf cells with each of the big leaf cells. This will eliminate all small leaf cells, while each merged cell consists of a constant number of the original cells.

5.2 The algorithm

Let q be a parameter such that $1 < q < p$. The value of q will be determined later.

Step (1) Constructing a partition tree

Randomly sample $\Theta(q\log p)$ points, and send all the sampled points to one server. The server builds a $\Theta(\log p)$-partial partition tree on the sampled points. By Theorem 6 and 7, this tree has $O(q)$ nodes, and with probability $1 - 1/p^{O(1)}$, every leaf cell contains $O(N_1/q)$ points in the original data set, and any hyperplane intersects $O(q^{1-\frac{1}{d}})$ cells. Then we perform the merging process described after Theorem 7. Since each merged cell consists of $O(1)$ leaf cells, each merged cell has constant description complexity and still contains $O(N_1/q)$ points. Note that the merging process does not increase the total cell count and the number of cells intersected by any hyperplane.

We broadcast these merged cells to all servers. This step incurs a load of $O(q\log p)$. Henceforth, a "cell" will refer to such a merged cell.

Similar to the intervals-containing-points algorithm, we consider the following two cases for all points inside a halfspace: (1) those in cells partially covered by the halfspace, and (2) those in cells fully covered by the halfspace.

Step (2) Partially covered cells

For each halfspace, we find all the cells Δ such that its bounding halfplane intersects Δ. There are $O(q^{1-\frac{1}{d}})$ such intersecting cells, by Theorem 7. For each cell Δ, we compute the number of halfspaces whose bounding halfplane intersects Δ, denoted as $P(\Delta)$. Note that $\sum_{\Delta} P(\Delta) = O(N_2 \cdot q^{1-\frac{1}{d}})$, so this is a sum-by-key problem on a total of $O(N_2 \cdot q^{1-\frac{1}{d}})$ key-value pairs. The load is thus

$$O\left(\frac{N_2}{p} \cdot q^{1-\frac{1}{d}}\right). \tag{2}$$

For each cell Δ, we allocate $p_{\Delta} = \left\lceil p \cdot \frac{P(\Delta)}{N_2 \cdot q^{1-\frac{1}{d}}} \right\rceil$ servers to compute the join between the $\Theta(N_1/q)$ points in the cell and these $P(\Delta)$ halfspaces that partially cover Δ. Note that the total number of servers needed is $\sum_{\Delta} p_{\Delta} = O(q+p) = O(p)$ as there are $O(q)$ cells in total. Invoking the hypercube algorithm to compute their Cartesian product incurs a load of

$$\begin{aligned} &O\left(\sqrt{\frac{\frac{N_1}{q} \cdot P(\Delta)}{p_{\Delta}}} + \frac{\frac{N_1}{q} + P(\Delta)}{p_{\Delta}}\right) \\ =&O\left(\sqrt{\frac{N_1 N_2}{p q^{\frac{1}{d}}}} + \frac{N_1}{q} + \frac{N_2 q^{1-\frac{1}{d}}}{p}\right). \end{aligned} \tag{3}$$

Choosing $q = p^{\frac{d}{2d-1}}$ balances the terms in (2) and (3), and the load becomes

$$O\left(\frac{\text{IN}}{q}\right) = O\left(\frac{\text{IN}}{p^{d/(2d-1)}}\right).$$

Step (3): Fully covered cells

In the intervals-containing-points algorithm, fully covered cells are dealt in a way similar to the partially covered cells, but that is because we can compute OUT and set the right slab size. In this algorithm, we may have used a cell size (i.e., IN/q) that is too small in relation to OUT. This would result in too many join results to be produced for the fully covered cells, exceeding the load target. Our strategy is thus to first estimate the join size for the fully covered cells (which is easier than computing OUT), and then rectify the mistake by restarting the whole algorithm with the right cell size, if needed.

Step (3.1): Join size estimation

For each cell Δ, let $F(\Delta)$ be the number of halfspaces fully covering it, and let $K = \sum_{\triangle} F(\triangle)$. Since every point inside Δ joins with every halfspace fully covering Δ, $K \cdot N_1/q$ is (a constant-factor approximation of) the remaining output size, and we will be able to estimate K easily.

We first compute an $(\frac{N_2}{q})$-thresholded approximation of $F(\Delta)$ for each Δ. This can be done by sampling $O(q \log p)$ halfspaces and collecting them on one server. For each cell Δ, we count the number of sampled halfspaces fully covering it, and scale up appropriately. Standard Chernoff type of analysis shows that with probability $1 - 1/p^{O(1)}$, we get an $(\frac{N_2}{q})$-thresholded approximation for every $F(\Delta)$. We use these approximate $F(\Delta)$'s to compute $\widehat{K}$, which is then an N_2-thresholded approximation of the true value of K.

Step (3.2): If $\widehat{K} < \frac{\text{IN} \cdot p}{q}$

Since we have chosen $q = o(p)$, if $\widehat{K} < \frac{\text{IN} \cdot p}{q}$ and $\widehat{K}$ is an N_2-thresholded approximation of K, then we must have $K = O(\frac{\text{IN} \cdot p}{q})$. In this case, we just break up each halfspace that fully covers k cells into k small pieces, which results in a total of K pieces. Now every piece covers exactly one cell, thus joins with all the points in that cell. The problem now reduces to an equi-join on two relations of size N_1 and K. Invoking the hypercube algorithm, the load is

$$O\left(\sqrt{\frac{\text{OUT}}{p}} + \frac{K + N_1}{p}\right) = O\left(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{q}\right).$$

Step (3.3): If $\widehat{K} > \frac{\text{IN} \cdot p}{q}$

In this case, we cannot afford to reduce the problem to an equi-join, since the halfspaces cover too many cells. This means we have used a cell size too small. Now, we restart the whole algorithm, but with a new $q' = \sqrt{\frac{\text{IN} \cdot pq}{\widehat{K}}} < q$. In the re-execution of the algorithm, we further merge every $O(q/q')$ cells into a bigger cell containing $\Theta(N_1/q')$ points. Now, each newly merged cell has non-constant description complexity, but since there are only a total of $O(q)$ cells, the entire mapping from these cells to the newly merged cells can be broadcast to all servers. Each server can still identify, for each of its points, which newly merged cell contains it.

Meanwhile, note that if $\widehat{K}$ is an N_2-thresholded approximation and $\widehat{K} > \frac{\text{IN} \cdot p}{q}$, then $\widehat{K}$ must be a constant-factor approximation of K and we have $q' = \Theta\left(\sqrt{\frac{\text{IN} \cdot pq}{K}}\right)$.

With the new q', step (1) has load $O(q' \log p) = O(q \log p)$; step (2) has load

$$O\left(\frac{\text{IN}}{q'}\right) = O\left(\sqrt{\frac{\text{IN} \cdot K}{pq}}\right) = O\left(\sqrt{\frac{\text{OUT}}{p}}\right),$$

where the second step uses the fact that $KN_1/q \le K \cdot \text{IN}/q$ is the output size for the fully covered cells in the first attempt of the algorithm, so must be no larger than OUT.

In the re-execution of the algorithm, let $F'(\Delta)$ be the number of halfspaces covering a newly merged cell Δ, and let $K' = \sum_{\Delta} F(\Delta)$. Observe that each newly merged cell consists of $\Theta(q/q')$ old cells. This means that we have $K' = O(Kq'/q)$, since any halfspace fully covering one newly merged cell must cover $\Theta(q/q')$ old cells (but not vice versa).

We argue that in the re-execution, we will always have $\widehat{K}' = O(\frac{\text{IN} \cdot p}{q'})$, thus always reaching step (3.2), whose load is $O\left(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{q'}\right) = O\left(\sqrt{\frac{\text{OUT}}{p}}\right)$. Indeed, we have

$$\begin{aligned} \widehat{K}' &= O(K' + \text{IN}) \\ &\qquad (\widehat{K}' \text{ is a } N_2\text{-thresholded approximation of } K') \\ &= O\left(K \cdot \frac{q'}{q} + \text{IN}\right) \\ &= O\left(\frac{\text{IN} \cdot pq}{(q')^2} \cdot \frac{q'}{q} + \text{IN}\right) = O\left(\frac{\text{IN} \cdot p}{q'}\right). \end{aligned}$$

Therefore, the re-execution, if it takes place, must have load $O\left(\sqrt{\frac{\text{OUT}}{p}}\right)$. Combining with the load of the first execution, we obtain the following result.

THEOREM 8. *There is a randomized algorithm that solves the halfspaces-containing-points problem in $O(1)$ rounds and* *load*

$$O\left(\sqrt{\frac{\mathrm{OUT}}{p}}+\mathrm{IN}/p^{\frac{d}{2d-1}}+p^{\frac{d}{2d-1}}\log p\right).$$

The algorithm succeeds with probability at least $1-1/p^{O(1)}$.

6. SIMILARITY JOIN IN HIGH DIMENSIONS

So far we have assumed that the dimensionality d is a constant. The load for both the ℓ_1/ℓ_∞ algorithm and the ℓ_2 algorithm hides constant factors that depend on d exponentially in the big-Oh notation. For the ℓ_2 algorithm, even for constant d, the term $O(\mathrm{IN}/p^{\frac{d}{2d-1}})$ approaches $O(\mathrm{IN}/\sqrt{p})$ as d grows, which is the load for computing the full Cartesian product.

In this section, we present an algorithm for high-dimensional similarity joins based on *locality sensitive hashing* (LSH), where d is not considered a constant. The nice thing about the LSH-based algorithm is that its load is independent of d (we still measure the load in terms of tuples; if measured in words, then there will be an extra factor of d). The downside is that its output-dependent term will not depend on OUT exactly; instead, it will depend on $\mathrm{OUT}(cr)$, which is the output size when the distance threshold of the similarity join is made c times larger, for some constant $c > 1$. LSH is known to be an approximate solution for nearest neighbor search, as it may return a neighbor whose distance is c times larger than the true nearest neighbor. In the case of similarity joins, all answers returned are truly within a distance of r (since this can be easily verified), but its cost will depend on $\mathrm{OUT}(cr)$ instead of OUT. It is also an approximate solution, in the sense that it approximates the optimal cost. The same notion of approximation has also been used for LSH-based similarity joins in the external memory model [25].

Let $\mathrm{dist}(\cdot,\cdot)$ be a distance function. For $c > 1, p_1 > p_2$, recall that a family $\mathcal{H}$ of hash functions is (r, cr, p_1, p_2)-sensitive, if for any uniformly chosen hash function $h \in \mathcal{H}$, and any two tuples x, y, we have (1) $\Pr[h(x) = h(y)] \geq p_1$ if $\mathrm{dist}(x, y) \leq r$; and (2) $\Pr[h(x) = h(y)] \leq p_2$ if $\mathrm{dist}(x, y) \geq cr$. In addition, we require $\mathcal{H}$ to be *monotone*, i.e., for a randomly chosen $h \in \mathcal{H}$, $\Pr[h(x) = h(y)]$ is a non-increasing function of $\mathrm{dist}(x, y)$. This requirement is not in the standard definition of LSH, but the LSH constructions for most metric spaces satisfy this property, include Hamming [19], ℓ_1 [12], ℓ_2 [5, 12], Jaccard [9], etc.

The quality of a hash function family is measured by $\rho = \frac{\log p_1}{\log p_2} < 1$, which is bounded by a constant that depends only on c, but not the dimensionality, and $\rho \approx 1/c$ for many common distance functions [19, 12, 5]. In a standard hash family $\mathcal{H}$, p_1 and p_2 are both constants, but by concatenating multiple hash functions independently chosen from $\mathcal{H}$, we can make p_1 and p_2 arbitrarily small, while $\rho = \frac{\log p_1}{\log p_2}$ is kept fixed, or equivalently, $p_2 = p_1^{1/\rho}$.

In the description of our algorithm below, we leave p_1, p_2 unspecified, which will be later determined in the analysis.

The algorithm proceeds in the following 3 steps:

(1) Choose $1/p_1$ hash functions $h_1, \ldots, h_{1/p_1} \in \mathcal{H}$ randomly and independently, and broadcast them to all servers.

(2) For each tuple x, make $1/p_1$ copies, and attach the pair $(i, h_i(x))$ to each of these copies, for $i = 1, \ldots, 1/p_1$.

(3) Perform an equi-join on all the copies of tuples, treating the pair $(i, h_i(x))$ as the join value, i.e., two tuples x, y join if $h_i(x) = h_i(y)$ for some i. For two joined tuples x, y, output them if $\mathrm{dist}(x, y) \leq r$.

THEOREM 9. *Assume there is a monotone (r, cr, p_1, p_2)-sensitive LSH family with $\rho = \frac{\log p_1}{\log p_2}$. Then there is a randomized similarity join algorithm that runs in $O(1)$ rounds and with expected load*

$$O\left(\sqrt{\frac{\mathrm{OUT}}{p^{1/(1+\rho)}}}+\sqrt{\frac{\mathrm{OUT}(cr)}{p}}+\frac{\mathrm{IN}}{p^{1/(1+\rho)}}\right).$$

The algorithm reports each join result with at least constant probability.

PROOF. Correctness of the algorithm follows from standard LSH analysis: For any two tuples x, y with $\mathrm{dist}(x, y) \leq r$, the probability that they join on one h_i is at least p_1. Across $1/p_1$ independently hash functions, we have constant probability that they join on at least one of them.

Below we analyze the load. Step (1) has load $O(1/p_1)$; step (2) is local computation. So we only need to analyze step (3).

The total number of tuples generated in step (2) is $O(N/p_1)$, which is the input size to the equi-join. The expected output size is at most

$$\mathrm{OUT}/p_1 + \mathrm{OUT}(cr) + \mathrm{IN}^2/p_1^{1-1/\rho}.$$

The first term is for all pairs (x, y) such that $\mathrm{dist}(x, y) \leq r$. They could join on every h_i. The second term is for (x, y)'s such that $r < \mathrm{dist}(x, y) \leq cr$. There are $\mathrm{OUT}(cr)$ such pairs, and each pair has probability at most p_1 to join on each h_i, so each pair joins exactly once in expectation. The last term is for all (x, y)'s such that $\mathrm{dist}(x, y) > cr$. There are N^2 such pairs, and each pair joins with probability at most p_2 on each h_i, so they contribute the term $\mathrm{IN}^2 p_2/p_1 = \mathrm{IN}^2/p_1^{1-\rho}$ in expectation.

Plugging these into Theorem 1, and using Jensen's inequality $E[\sqrt{X}] \leq \sqrt{E[X]}$, the expected load can be bounded by (the big-Oh of)

$$\frac{\sqrt{\mathrm{OUT}/p_1+\mathrm{OUT}(cr)+\mathrm{IN}^2/p_1^{1-1/\rho}}}{\sqrt{p}}+\frac{\mathrm{IN}}{pp_1}$$
$$\leq\sqrt{\frac{\mathrm{OUT}}{pp_1}}+\sqrt{\frac{\mathrm{OUT}(cr)}{p}}+\mathrm{IN}\sqrt{\frac{1}{pp_1^{1-1/\rho}}}+\frac{\mathrm{IN}}{pp_1}.$$

Setting $p_1 = 1/p^{\frac{\rho}{1+\rho}}$ balances the last two terms, and we obtain the claimed bound in the theorem. □

Remark. Note that since $0 < \rho < 1$, the input-dependent term is always better than performing a full Cartesian product. The output-term $O\left(\sqrt{\frac{\mathrm{OUT}(cr)}{p}}\right)$ is also the best we can achieve for any LSH-based algorithm, by the following intuitive argument: Due to its approximation nature, LSH cannot tell whether the distance between two tuples are smaller than r or slightly above r. A worst-case scenario is all the $\mathrm{OUT}(cr)$ pairs of tuples have distance slightly above r but none of them actually joins. Unfortunately, since the hash functions cannot distinguish the two cases, any LSH-based algorithm will have to check all the $\mathrm{OUT}(cr)$ pairs to make sure that it does not miss any true join results. Finally, the term $O\left(\sqrt{\frac{\text{OUT}}{p^{1/(1+\rho)}}}\right)$ is also worse than the bound $O\left(\sqrt{\frac{\text{OUT}}{p}}\right)$ we achieved in earlier sections. This is perhaps the best one can hope for as well, if $O(1)$ rounds are required: In order to capture all joining pairs, $1/p_1$ repetitions are necessary, and two very close tuples may join in all these repetitions, introducing the extra $1/p_1$ factor in the output size. If we want to perform all of them in parallel, there seems to be no way to eliminate the redundancy beforehand. Of course, this is just an intuitive argument, not a formal proof.

7. A LOWER BOUND ON 3-RELATION CHAIN JOIN

In this section, we consider the possibility of designing output-optimal algorithms for multi-way joins. We show that, unfortunately, this is not possible, even for the simplest multi-way join, a 3-relation equi-join, $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$.

The first question is how an output-optimal term would look like for a 3-relation join. Applying the tuple-based argument in Section 1.2, each server can potentially produce $O(L^3)$ join results in a single round, hence $O(pL^3)$ results over all p servers in a constant number of round. Thus, an $O((\text{OUT}/p)^{1/3})$ term is definitely output-optimal.

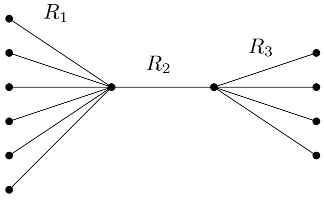

Figure 3: An instance of a 3-relation chain join.

However, consider the instance shown in Figure 3, where we use vertices to represent attribute values, and edges for tuples. On such an instance, the 3-relation join degenerates into the Cartesian product of R_1 and R_3. Each server can produce at most $O(L^2)$ pairs of tuples in one round, one from R_1 and one from R_3, so we must have $O(pL^2) = \Omega(\text{OUT})$, or $L = \Omega(\sqrt{\text{OUT}/p})$. This means that the best possible output-dependent term is still $O(\sqrt{\text{OUT}/p})$. Below we show that this is not possible, either, assuming any meaningful input-dependent term.

THEOREM 10. *For any tuple-based algorithm computing a 3-relation chain join, if its load is in the form of*

$$L = O\left(\frac{\text{IN}}{p^\alpha} + \sqrt{\frac{\text{OUT}}{p}}\right),$$

for some constant α, then we must have $\alpha \le 1/2$, provided $\text{IN}/\log^2 \text{IN} > cp^3$ for some sufficiently large constant c.

Note that there is an algorithm for the 3-relation chain join with load $\tilde{O}(\text{IN}/\sqrt{p})$ [21], without any output-dependent term. This means that it is meaningless to introduce the output-dependent term $O(\sqrt{\text{OUT}/p})$.

PROOF. Suppose there is an algorithm with a claimed load L in the form stated above. We will construct a hard instance on which we must have $\alpha \le 1/2$. Our construction is probabilistic, and we will show that with high probability, the constructed instance satisfies our needs.

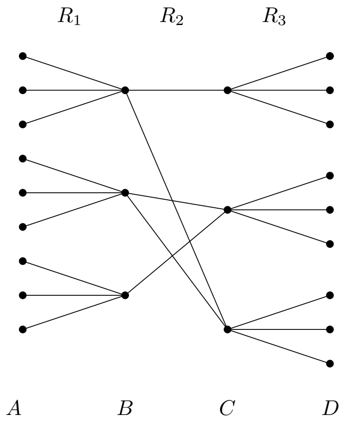

Figure 4: A randomly constructed hard instance.

The construction is illustrated in Figure 4. More precisely, attributes B and C each have $\frac{N}{\sqrt{L}}$ distinct values. Each distinct value of B appears in $\sqrt{L}$ tuples in R_1, and each distinct value in C appears in $\sqrt{L}$ tuples in R_3. Each distinct value of B and each distinct value of C have a probability of $\frac{L}{N}$ to form a tuple in R_2. Note that R_1 and R_3 are deterministic and always have N tuples, while R_2 is probabilistic with N tuples in expectation, so $E[\text{IN}] = 3N$. The output size is expected to be $E[\text{OUT}] = NL$. By the Chernoff inequality, the probability that IN or OUT deviates from their expectations by more than a constant fraction is $\exp(-\Omega(N))$.

We allow all servers to access R_2 for free, and only charge the algorithm for receiving tuples from R_1 and R_3. More precisely, we bound the maximum number of join results a server can produce in a round, if it only receives L tuples from R_1 and L tuples from R_3. Then we multiply this number by p, which must be larger than OUT. Note that this is purely a counting argument; if the same join result is produced at two or more servers, it is counted multiple times.

First we argue that a server should load R_1 and R_3 in whole groups in order to maximize its output size. Here, a group means all tuples sharing the same value on B (or C). Without loss of generality, suppose two groups in R_1 are not loaded in full by a server, say, g_1 and g_2. If g_1 joins more tuples from R_3 that have been loaded by the same server than g_2, then we can always shift tuples from g_2 to g_1 so as to produce more (at least not less) join results.

Thus, each server in each round loads $\sqrt{L}$ groups from R_1 and $\sqrt{L}$ groups from R_3. Below we show that, on a random instance constructed as above, with high probability, not many pairs of groups can join, no matter which subset of $2\sqrt{L}$ groups are loaded. Consider any subset of $\sqrt{L}$ groups from R_1 and any subset of $\sqrt{L}$ groups from R_3. There are L possible pairs of groups, and each pair has probability $\frac{L}{N}$ to join, so we expect to see $\frac{L^2}{N}$ pairs to join. By Chernoff bound, the probability that more than $2\frac{L^2}{N}$ pairs join is at most $\exp(-\Omega(\frac{L^2}{N}))$. There are $O\left(\left(N/\sqrt{L}\right)^{2\sqrt{L}}\right)$ different choices of $\sqrt{L}$ groups from R_1 and $\sqrt{L}$ groups from R_3.

So, the probability that one of them yields more than $2\frac{L^2}{N}$ joining groups is at most

$$O\left(\left(\frac{N}{\sqrt{L}}\right)^{2\sqrt{L}}\right) \cdot \exp\left(-\Omega\left(\frac{L^2}{N}\right)\right)$$
$$= \exp\left(-\Omega\left(\frac{L^2}{N}\right) - O\left(\sqrt{L}\cdot\log N\right)\right).$$

This probability is exponentially small if

$$\frac{L^2}{N} > c_1\sqrt{L}\cdot\log N,$$

for some sufficiently large constant c_1. Rearranging, we get

$$N\log N < \frac{1}{c_1}\cdot L^{\frac{3}{2}}.$$

By Theorem 2, we always have $L = \Omega(N/p)$, so this is true as long as

$$N\log N < \frac{1}{c_2}\cdot\left(\frac{N}{p}\right)^{\frac{3}{2}},$$

for some sufficiently large constant c_2, or $N/\log^2 N > c_2p^3$.

By a union bound, we conclude that with high probability, a randomly constructed instance has IN $= \Theta(N)$, OUT $= \Theta(NL)$, and on this instance, no matter which groups are chosen, no more than $\frac{2L^2}{N}$ pairs of groups can join. Since each pair of joining groups produces L results, the p servers in total produce $O\left(\frac{L^3p}{N}\right)$ results in a constant number of rounds. So we have

$$\frac{L^3p}{N} = \Omega(NL),$$

i.e.,

$$L = \Omega\left(\frac{N}{\sqrt{p}}\right).$$

Suppose an algorithm has a load in the form as stated in the theorem, then with OUT $= \Theta(NL)$, we have

$$\frac{N}{p^\alpha} + \sqrt{\frac{NL}{p}} = \Omega\left(\frac{N}{\sqrt{p}}\right).$$

If $\alpha > 1/2$, we must have

$$\sqrt{\frac{NL}{p}} = \Omega\left(\frac{N}{\sqrt{p}}\right),$$

or $L = \Omega(N)$, which is an even higher lower bound. Thus we must have $\alpha \leq 1/2$. □

8. CONCLUDING REMARKS

Our negative result has ruled out the possibility of having output-optimal algorithms for any join on 3 or more relations. However, there is still hope if we sacrifice the output optimality slightly. For example, what can be done if the output-dependent term is to be $\tilde{O}(\sqrt{\text{OUT}/p^{1-\delta}})$ for some small δ?

More broadly, using OUT as a parameter to measure the complexity falls under the realm of parameterized complexity, or beyond-worst-case analysis in general. This type of analyses often yields more insights for problems where worst-case scenarios are rare in practice, such as joins. While OUT is considered the most natural additional parameter to introduce, other possibilities exist, such as assuming that the data follows certain parameterized distributions, or the degree (i.e., maximum number of tuples a tuple can join) is bounded [10, 24], etc.

9. REFERENCES

[1] F. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. Gym: A multiround join algorithm in mapreduce. *arXiv preprint arXiv:1410.4156*, 2014.

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

[3] P. K. Agarwal, K. Fox, K. Munagala, and A. Nath. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Proc. ACM Symposium on Principles of Database Systems*, 2016.

[4] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proc. IEEE Symposium on Foundations of Computer Science*, 2006.

[6] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 739–748, 2008.

[7] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proc. ACM Symposium on Principles of Database Systems*, 2013.

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

[9] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[10] Y. Cao, W. Fan, T. Wo, and W. Yu. Bounded conjunctive queries. In *Proc. International Conference on Very Large Data Bases*, 2014.

[11] T. M. Chan. Optimal partition trees. *Discrete & Computational Geometry*, 47(4):661–690, 2012.

[12] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality sensitive hashing scheme based on p-stable distributions. In *Proc. Annual Symposium on Computational Geometry*, 2004.

[13] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. Symposium on Operating Systems Design and Implementation*, 2004.

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

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

[17] S. Har-Peled and M. Sharir. Relative (p, ε)-approximations in geometry. *Discrete & Computational Geometry*, 45(3):462–496, 2011.

[18] X. Hu, P. Koutris, and K. Yi. The relationships among coarse-grained parallel models. Technical report, HKUST, 2016.

[19] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. ACM Symposium on Theory of Computing*, 1998.

[20] 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.

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

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

[23] Y. Li, P. M. Long, and A. Srinivasan. Improved bounds on the sample complexity of learning. *Journal of Computer and System Sciences*, 62(3):516–527, 2001.

[24] C. R. Manas Joglekar. It's all a matter of degree: Using degree information to optimize multiway joins. In *Proc. International Conference on Database Theory*, 2016.

[25] R. Pagh, N. Pham, F. Silvestri, and M. Stöckel. I/O-efficient similarity join. In *Proc. European Symposium on Algorithms*, 2015.

[26] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. In *Proc. ACM Symposium on Principles of Database Systems*, 2014.

[27] M. Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing*, 40(3), 2011.

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

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX conference on Networked Systems Design and Implementation*, 2012.