Output-Optimal Massively Parallel Algorithms for Similarity Joins

XIAO HU and KE YI, Hong Kong University of Science and Technology
YUFEI TAO, The Chinese University of Hong Kong

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-case 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. 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 article, 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.

CCS Concepts: • **Theory of computation → Database query processing and optimization (theory)**;

Additional Key Words and Phrases: Parallel computation, similarity joins, output-sensitive algorithms

ACM Reference format:
Xiao Hu, Ke Yi, and Yufei Tao. 2019. Output-Optimal Massively Parallel Algorithms for Similarity Joins. *ACM Trans. Database Syst.* 44, 2, Article 6 (March 2019), 36 pages.
https://doi.org/10.1145/3311967

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 article, we will be mostly interested in the ℓ_1, ℓ_2, and ℓ_∞ distances, although some of our results (the one based on locality-sensitive hashing (LSH)) can be extended to other distance functions as well.

The first two authors were supported by Hong Kong RGC under grants 16200415, 16202317, and 16201318, and by grants from Microsoft and Alibaba. Y. Tao was partially supported by a direct grant (4055079) from The Chinese University of Hong Kong and by a Faculty Research Award from Google.
Authors' addresses: X. Hu and K. Yi, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong, China; emails: {xhuam, yike}@cse.ust.hk; Y. Tao, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong, China; email: taoyf@cse.cuhk.edu.hk.

0362-5915/2019/03-ART6 $15.00
https://doi.org/10.1145/3311967

1.1 The Computation Model

Driven by the rapid development of massively parallel systems such as MapReduce [14], Spark [33], 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 (MPC) model [1–3, 7, 8, 22–24].

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 sends messages to other servers. After all messages have arrived at their destinations, the servers conduct some local computation in parallel and proceed to the next round. The complexity of the algorithm is measured first by the number of rounds, then the *load*, denoted as 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, 23, 24]; however, recently, more interests have been given to multi-round (but still a constant) algorithms [1, 3, 22, 23], as 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.

Note that the MPC model is essentially Valiant's bulk synchronous processing (BSP) model [32] without restricting the outgoing messages. Theoretically, this further simplifies the model. The practical justification is that the routing of outgoing messages overlaps with the computation phase, making it negligible. Furthermore, the incoming message size also indirectly determines the memory requirement and local computation time on each server in the following round, which is why this measure is referred to as "load."

Let N_1 and N_2 be the sizes of the two relations to be joined, and let $\text{IN} = N_1 + N_2$. In this article, we will focus on the case when $1/p \leq N_1/N_2 \leq p$. This is because when N_1/N_2 falls outside this range, the trivial algorithm that broadcasts the smaller relation to all servers would be optimal, as shown in Section 3.2.

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 with load $L = \tilde{O}(\sqrt{N_1 N_2/p})$.[1] This load 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—that is, 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 other works [7, 8, 23].

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

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

However, on most realistic datasets, the join size is nowhere near the worst case. Suppose that the join size is OUT. Applying the same argument as earlier, 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—that is, 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 2-relation join. However, both terms $O(\text{OUT}/\sqrt{p})$ or $O(\text{IN}/\sqrt{p})$ are far from optimal. Beame et al. [8] proposed a randomized algorithm with optimal load $\tilde{O}(\sqrt{\text{OUT}/p} + \text{IN}/p)$, but up to logarithmic factors, due to the use of random hashing. They also assume that each server knows the data statistics in advance.

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 [23] that converts MPC join algorithms into I/O-efficient counterparts under the *enumerate version* [29] 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 [23] is shown to imply an EM algorithm matching the I/O lower bound of Pagh and Silvestri [29] up to a logarithmic factor.

1.3 Our Results

We start with an improved algorithm for computing the equi-join between two relations—for instance, a degenerated similarity join with $r = 0$. We improve upon the algorithm of Beame et al. [8] in the following aspects. First, our algorithm does not assume any prior statistical information about the data, such as the heavy join values and their frequencies. Second, the load of our algorithm is exactly $O(\sqrt{\text{OUT}/p} + \text{IN}/p)$ tuples, without any extra logarithmic factors. Third, 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.

Although the $O(\sqrt{\text{OUT}/p})$ term is optimal by the preceding tuple-based argument, prior work did not show why the input-dependent term $O(\text{IN}/p)$ is necessary. Note that 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 2-relation joins or the AGM bound [6] for multi-way joins). In this case, the preceding simple tuple-based argument already leads to a lower bound higher than $\Omega(\text{IN}/p)$, so this is not an issue. However, when the output size OUT becomes a parameter, these worst-case constructions do not work anymore, and it is not clear why $O(\text{IN}/p)$ load is necessary. Indeed, if $\text{OUT} = 1$, then the preceding tuple-based argument yields a meaningless lower bound of $\Omega(1/p)$. To complete the picture, we provide a lower bound showing that even if $\text{OUT} = O(1)$, computing the equi-join between two relations requires $\Omega(\text{IN}/p)$ load, by resorting to strong results from communication complexity.

The main theoretical results in this article, however, are on similarity joins with $r > 0$. Specifically, we obtain the following results under various distance metrics:

(1) For ℓ_1/ℓ_∞ distance in constant dimensions, we give a deterministic algorithm with load

$$O\left(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{p} \cdot \log^{O(1)} p\right)$$

—that is, the output-dependent term is optimal, whereas the input-dependent term is away from optimality by a polylogarithmic factor, which depends on the dimensionality.

(2) For ℓ_2 distance in d dimensions, we give a randomized algorithm with load

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

Again, the term $O(\sqrt{\frac{\text{OUT}}{p}})$ is output-optimal. The input-dependent term $O(\text{IN}/p^{\frac{d}{2d-1}})$ 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 $O(\text{IN}/\sqrt{p})$, which is the load for computing the full Cartesian product.

(3) In high dimensions, we provide an algorithm based on LSH with load

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

where $\text{OUT}(cr)$ 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 $O(\text{IN}/p^{1/(1+\rho)})$ is always better than that for computing the Cartesian product, although output-optimality here is not only with respect to OUT but also $\text{OUT}(cr)$, due to the approximation nature of LSH.

All the algorithms run in $O(1)$ rounds, under the mild assumption $\text{IN} > p^{1+\epsilon}$, where $\epsilon > 0$ is any small constant. Note that the randomized output-optimal algorithm in Beame et al. [8] for equi-joins has an implicit assumption that $\text{IN} \geq p^2$, as 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, $\text{IN} \geq p^2$ is a very reasonable assumption. Our desire to relax this to $\text{IN} > p^{1+\epsilon}$ 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 [16] has shown that if $\text{IN} = p^{1+o(1)}$, then even computing the "or" of IN bits requires $\omega(1)$ rounds under load $O(\text{IN}/p)$.

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 Afrati et al. [1] mentioned in Section 1.2. However, that term is $O(\text{OUT}/\sqrt{p})$, which is almost quadratically larger than the output-optimal term $O(\sqrt{\text{OUT}/p})$ that we achieved earlier. We show that, unfortunately, such an output-optimal term is not achievable for a simple 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

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

for some constant α, then we must have $\alpha \leq 1/2$, provided $\text{IN} \log^2 \text{IN} = \Omega(p^3)$. However, the algorithm in Beame et al. [23] can already compute any 3-relation chain join with $\tilde{O}(\text{IN}/\sqrt{p})$ load. This means that it is meaningless to introduce the output-dependent term $O(\sqrt{\text{OUT}/p})$.

The conference version of this article [19] is mostly concerned with theoretical optimality. In this extended article, we also look at the practical side of the problem. One of the most serious issues in implementing the theoretically optimal MPC algorithm in practice is the large, although still constant, number of rounds. In the current distributed data processing systems, each round incurs a substantial synchronization overhead, and the benefit of an asymptotic smaller load can be easily offset by the large system overhead. To more precisely model the cost of each synchronous round, we classify them into *heavy rounds* and *light rounds* (Section 8), where a round is *light* if its load is $\tilde{O}(p)$ and *heavy* otherwise. Since $p \ll L$ in practice, it is desirable to minimize the number of heavy rounds while being a bit more tolerant on the light rounds. Then, we design a practical version of our equi-join algorithm and the 1D similarity join algorithm so that both run in a *single* heavy round and a constant number of light rounds. We have implemented these algorithms in Spark and conducted experiments comparing with other techniques. For equi-join, we have also implemented the hash-based output-optimal algorithm of Beame et al. [8], which had not been implemented before. In Section 9, the experimental results suggest that the output-optimal equi-join algorithms can significantly outperform the vanilla join algorithm in Spark, which is based on the simple hash join. We have also conducted experiments for similarity joins on high-dimensional data. The experimental results show that our LSH-based algorithm yields the best performance among a number of alternatives.

2 PRELIMINARIES

In this section, we first define the similarity join problem under different distance metrics, and their reduction to various geometric containment problems, which will be focused upon in the rest of the article. Then we introduce several primitive operations in the MPC model, which will be used by our algorithms as building blocks.

2.1 Similarity Joins

Similarity joins under $r = 0$ are equi-joins, where two points can be joined if and only if they are equal. For $r > 0$, a similarity join is better interpreted geometrically. We model input tuples as points in $\mathbb{R}^d$, and let $R_1, R_2 \subset \mathbb{R}^d$ be two sets of points with $|R_1| = N_1, |R_2| = N_2$. For two points $x = (x_1, x_2, \ldots, x_d) \in R_1, y = (y_1, y_2, \ldots, y_d) \in R_2$, their distance under the ℓ_k norm is

$$\|x - y\|_k = \left(\sum_{i=1}^{d} |x_i - y_i|^k \right)^{1/k}.$$

The most commonly used ℓ_k norms are the ℓ_1-norm, ℓ_2-norm, and ℓ_∞ norm. Note that for $k = \infty$, $\|x - y\|_\infty = \max_{i=1,2,\ldots,d} |x_i - y_i|$.

2.1.1 Similarity Join Under ℓ_1/ℓ_∞ Distance. It is well known that the ℓ_1 metric in d dimensions can be embedded into the ℓ_∞ metric in 2^{d-1} dimensions via the following transformation. Note that for any point $(x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$, we have

$$\sum_{i=1}^{d} |x_i| = \max_{(z_2,\ldots,z_d)\in\{-1,1\}^{d-1}} |x_1 + z_2 x_2 + \cdots + z_d x_d|.$$

Thus two points $(x_1, \ldots, x_d) \in R_1$ and $(y_1, \ldots, y_d) \in R_2$ join under ℓ_1 distance if and only if

$$\max_{(z_2,\ldots,z_d)\in\{-1,1\}^{d-1}} |(x_1 + z_2 x_2 + \cdots + z_d x_d) - (y_1 + z_2 y_2 + \cdots + z_d y_d)| \le r.$$

We map the point $(x_1, x_2, \ldots, x_d)$ to a point in 2^{d-1} dimensions, where each dimension has coordinate $x_1 + z_2 x_2 \cdots + z_d x_d$ corresponding to each $(z_2, \ldots, z_d) \in \{-1, 1\}^{d-1}$. The similar

transformation applies for point $(y_1, y_2, \ldots, y_d)$. These two d-dimensional points join under ℓ_1 distance if and only if the corresponding two 2^{d-1}-dimensional points join under ℓ_∞ distance.

Next, we reduce the similarity join under ℓ_∞ distance to the *rectangles-containing-points problem*: 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 $x \in y$. The reduction is quite straightforward. We map the point $(y_1, y_2, \ldots, y_d) \in R_2$ to a d-dimensional rectangle defined by the Cartesian product of intervals $[y_i - r, y_i + r]$ for $i = 1, 2, \ldots, d$. The points in R_1 remain unchanged. It should be obvious that two points $x \in R_1$ and $y \in R_2$ join under ℓ_∞ distance if and only if x falls inside the rectangle corresponding to y. Note that the reduction will only produce squares, but our solution to the rectangle-containing-points can handle general orthogonal rectangles.

2.1.2 Similarity Join Under ℓ_2 Distance. We use the *lifting transformation* [13] to reduce the similarity join under ℓ_2 distance to the *halfspaces-containing-points* problem in $d + 1$ dimensions. More precisely, we map each point $(x_1, \ldots, x_d) \in R_1$ to a point in $d + 1$ dimensions as

$$\left(x_1, \ldots, x_d, x_1^2 + \cdots + x_d^2\right)$$

and map a point $(y_1, \ldots, y_d) \in R_2$ to a halfspace in $d + 1$ dimensions (y_i's are the coefficients and z_i's are the variables) as

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

Observe that $(x_1 - y_2)^2 + \cdots + (x_d - y_d)^2 \leq r^2$ can be rewritten as

$$x_1^2 + y_1^2 + \cdots + x_d^2 + y_d^2 - 2x_1y_1 - \cdots - 2x_dy_d - r^2 \geq 0.$$

Thus, two points $x \in R_1$ and $y \in R_2$ join in the original d-dimensional space under ℓ_2 distance if and only if the lifted x falls inside the halfspace corresponding to y in the $(d + 1)$-dimensional space. Thus, it is sufficient to solve the halfspaces-containing-points problem: given a set of N_1 points and a set of N_2 halfspaces, report all the (point, halfspace) pairs such that the point is inside the halfspace.

2.2 MPC Primitives

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

2.2.1 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, whereas 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 [16]. His algorithm has load $L = \Theta(\text{IN}/p)$ and runs in $O(\log_L \text{IN}) = O(\log_L(pL)) = O(\log_L p)$ rounds. When $\text{IN} > p^{1+\epsilon}$, this is $O(1)$ rounds.

2.2.2 Multi-Numbering. Suppose that each tuple carries a key, and that there are n_k tuples for key k. The goal of the *multi-numbering* problem is to, for each key k, assign consecutive numbers $1, 2, \ldots, n_k$ to the n_k tuples with key k, respectively.

We solve this problem by reducing it to the *all prefix-sums* problem: given an array of elements $A[1], \ldots, A[\text{IN}]$, compute $S[i] = A[1] \oplus \cdots \oplus A[i]$ for all $i = 1, \ldots, \text{IN}$, where $\oplus$ is any associative operator. Goodrich et al. [17] gave an algorithm in the BSP model for this problem that uses $O(\text{IN}/p)$ 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 $(x_1, y_1) \oplus (x_2, y_2) = (x_1 x_2, y)$, where $y = y_1 + y_2$ if $x_2 = 1$ and otherwise $y = y_2$.

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, whereas 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.

2.2.3 Sum-by-Key. Suppose that 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 earlier, 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 preceding algorithm 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.

2.2.4 Multi-Search. The *multi-search* problem is defined as follows. Let U be a universe with a total order, and let $S, Q \subseteq U$ be two subsets of elements from U. Let $|S| = N_1$, $|Q| = N_2$, and $\text{IN} = N_1 + N_2$. The elements in S are called *keys*, and the elements in Q are called *queries*. The problem asks us to, for each query $q \in Q$, find its predecessor in S (i.e., the largest key that is no larger than q).

The multi-search algorithm given by Goodrich et al. [17] is randomized, with a small probability exceeding $O(\text{IN}/p)$ load. In fact, this problem can also be solved using all prefix-sums, which results in a deterministic algorithm with load $O(\text{IN}/p)$. We first sort all the keys and queries together; in case of ties, we put the keys before the queries. Then for each key k, define its corresponding $A[i]$ as itself; for each query, define its $A[i] = -\infty$; define $\oplus = \max$. Then each query has its $S[i] = \max_{j \le i} A[j]$, which is the largest key among those smaller than the query itself (i.e., its predecessor in the keys).

2.2.5 Cartesian Product. The Cartesian product of two sets of size N_1 and N_2, respectively, can be reported using a degenerated version of the hypercube algorithm [2, 8], incurring a load of $O((\sqrt{N_1 N_2/p} + \text{IN}/p) \log p)$ with probability $1 - 1/p^{\Omega(1)}$. 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 that $N_1 \le N_2$. As in the standard hypercube algorithm, we arrange the p servers into a $d_1 \times d_2$ grid such that $d_1 d_2 = p$. We first use multi-numbering to assign consecutive numbers to all tuples in R_1 and R_2, respectively. 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 $d_1 = \sqrt{\frac{pN_1}{N_2}}, d_2 = \sqrt{\frac{pN_2}{N_1}}$, the load is $O(\sqrt{N_1N_2/p} + \text{IN}/p)$.

2.2.6 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 $\text{IN} \ge p^2$, 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 Beame et al. [8]. When we only have $\text{IN} \ge p^{1+\epsilon}$, some more work is needed to ensure $O(\text{IN}/p)$ 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 to 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 (natural join) problem between two relations, $R_1(A, B) \bowtie R_2(B, C)$. Let N_1 and N_2 be the sizes of R_1 and R_2, respectively; set $\text{IN} = N_1 + N_2$.

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 $|R_1(v)| \ge N_1/p$ or $|R_2(v)| \ge N_2/p$, and *light* otherwise. Then they gave an algorithm with load

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

We observe that this bound is asymptotically the same as $\tilde{\Theta}(\sqrt{\text{OUT}/p} + \text{IN}/p)$, because

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

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 $\tilde{\Omega}(\sqrt{\text{OUT}/p - \text{IN}^2/p^2})$ since

$$\sum_{\text{light } v} |R_1(v)| \cdot |R_2(v)| \le \frac{N_1N_2}{p} \le \frac{\text{IN}^2}{p}.$$

Thus, we have

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

Therefore, their algorithm is output-optimal, but up to a logarithmic factor. Furthermore, 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 $i = 1, 2$. In the following, we describe a deterministic algorithm that achieves the result in Theorem 3.1.

3.1 The Algorithm

Our algorithm can be seen as an MPC version of the classical sort-merge-join algorithm. The high-level procedure is given in Algorithm 1. It decomposes the equi-join into a set of Cartesian products, one for each distinct join value in the domain of B, and runs the hypercube algorithm to compute each Cartesian product in parallel. To achieve output-optimality, it needs to compute OUT first and allocate the appropriate number of servers to each subproblem. The details in each step are described in the following.

ALGORITHM 1: EQUI-JOIN $R_1(A, B) \bowtie R_2(B, C)$

1 Collect data statistics and compute OUT;
2 $p_v \leftarrow \max\left\{\left\lceil p \cdot \frac{N_1(v)+N_2(v)}{\text{IN}}\right\rceil, \left\lceil p \cdot \frac{N_1(v)N_2(v)}{\text{OUT}}\right\rceil\right\}$ for each $v \in \text{dom}(B)$ that has tuples on at least 2 servers;
3 **for** *each such* v **do in parallel**
4 Compute $R_1(v) \times R_2(v)$ with p_v servers;

Step (1): Computing OUT. Consider each distinct value v of the join attribute B. Let $R_i(v) = \sigma_{B=v}R_i$, and 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 in $R_i(v)$ 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 attribute B. Consider each distinct value v in B. 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 two or more servers. There are at most p such v's. For each such v, we allocate

$$p_v = \max\left\{\left\lceil p \cdot \frac{N_1(v) + N_2(v)}{\text{IN}}\right\rceil, \left\lceil p \cdot \frac{N_1(v)N_2(v)}{\text{OUT}}\right\rceil\right\} \tag{2}$$

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 tuples 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 $O(\max_v\{\sqrt{\frac{N_1(v)N_2(v)}{p_v}} + \frac{N_1(v)}{p_v} + \frac{N_2(v)}{p_v}\}) = O(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{p})$.

THEOREM 3.1. *There is a deterministic algorithm that computes the equi-join between two relations in $O(1)$ rounds with load $O(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{p})$. It does not assume any prior statistical information about the data.*

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. In the following, we show an input-dependent lower bound of $\Omega(\min\{N_1, N_2, \text{IN}/p\})$ in terms of the number of bits even when $\text{OUT} = O(1)$. Note that when $1/p \le N_1/N_2 \le p$, this lower bound becomes $\Omega(\text{IN}/p)$, matching the upper bound in Theorem 3.1 up to a logarithmic factor. When $N_1/N_2 < 1/p$ or $N_1/N_2 > p$, the lower bound becomes $\Omega(\min\{N_1, N_2\})$, which can be matched up to a logarithmic factor by the trivial algorithm that simply broadcasts the smaller relation to all servers.

Theorem 3.2. *Any randomized algorithm that computes the equi-join between two 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 $\le n$ elements and Bob has $\le m$ 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 [30]. This holds even for randomized algorithms with a success probability larger than $3/4$. We also note that in the hard instances used in Pătraşcu [30], the intersection size of Alice's and Bob's sets is either 0 or 1.

The reduction works as follows. Assuming that $N_1 \le N_2$, we will show in the following an $\Omega(\min(N_1, \text{IN}/p))$ lower bound; symmetrically, if $N_2 \le N_1$, we can show an $\Omega(\min(N_2, \text{IN}/p))$ lower bound. Combining the two cases proves the theorem. 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, whereas 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 $N_2 \le p \cdot N_1$, 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_∞

Recall that in Section 2.1, we reduced similarity joins to various geometric containment problems. In each geometric containment problem, we are given a set of points R_1 and a set of ranges R_2, and the goal is to find all (point, range) pairs such that the point is contained in the range.

All our algorithms for solving different containment problems are based on the following basic ideas. Each range is first decomposed into non-overlapping cells. We assign a join key for each cell, as well as for each point, and then invoke the equi-join algorithm to find all the (point, cell) pairs that share the same join key. We can easily assign join keys to ensure completeness (i.e., all (point, cell) pairs where the point is contained in the cell are assigned the same join key), and hence must be captured by the equi-join algorithm. In fact, a trivial method for completeness is just to assign the same join key to all points and all cells. However, this is essentially computing the full Cartesian product. To achieve output-optimality, we need to more carefully assign the join keys so that the

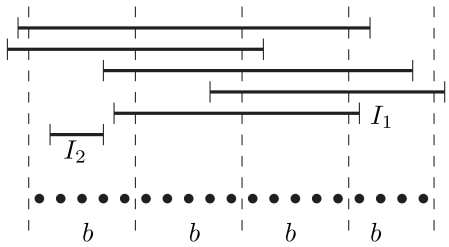

Fig. 1. Partially covered and fully covered slabs.

equi-join algorithm does not report too many potential join results. For asymptotic optimality, the equi-join algorithm should return at most a constant times more than the actual join results (i.e., $O(\text{OUT})$) while guaranteeing completeness. The exact way in which the join keys are assigned are different depending on the particular distance metric under consideration. The ℓ_1/ℓ_∞ metric (rectangles-containing-points) will be discussed in Section 4 and ℓ_2 metric (halfspaces-containing-points) in Section 5.

In addition, knowing the value of OUT is crucial, which will be used to allocate servers for distinct join keys. Although for some problems OUT can be computed easily, for other problems computing OUT exactly would be difficult. In these cases, we compute an estimate of OUT that is accurate enough for asymptotic optimality.

In this section and the next, we will assume constant dimensions, which means that it is sufficient to solve the various geometric containment problems in constant dimensions by the reductions in Section 2.1. We deal with the high-dimensional case in Section 6.

4.1 One Dimension

We start by considering the 1D case—that is, 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 is to report all (point, interval) pairs such that the point is inside the interval. In the following, we describe how to solve this problem in $O(\sqrt{\text{OUT}/p} + \text{IN}/p)$ 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 of these counts to get OUT (special case of sum-by-key).

Step (2): Sorting into slabs. 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. There are at most p slabs, which are labeled as $1, 2, 3, \ldots$, in sorted order. 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), whereas 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.

Step (3): Partially 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 that it contains

$P(i)$ endpoints. We allocate $\lceil p \cdot \frac{P(i)}{N_2} \rceil$ 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 (4): 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 the prefix-sums algorithm as follows. If an interval fully covers slabs $i, \ldots, j$, we generate two (key, value) pairs $(i, 1)$ and $(j + 1, -1)$. For each slab k, we generate a (key, value) pair $(k + 0.5, 0)$. Then we sort all these pairs by key and compute the prefix-sums on the value. Consider the prefix-sum at key $k + 0.5$. Observe that an interval fully covering slabs $i, \ldots, j$ contributes 1 to this prefix-sum if $i \leq k \leq j$ and 0 otherwise: if $i > k$, the keys of the two pairs generated by the interval are both after $k + 0.5$; if $j < k$, the values of the two pairs cancel out. Therefore, the prefix-sum computed at key $k + 0.5$ is exactly $F(i)$.

Now, the full slabs can be dealt with using essentially the same algorithm. We allocate $p_i = \lceil p \cdot \frac{bF(i)}{\text{OUT}} \rceil$ 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 $\sum_i bF(i) \leq \text{OUT}$, 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

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

Theorem 4.1. *There is a deterministic algorithm for the intervals-containing-points problem that runs in $O(1)$ rounds with $O(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{p})$ load.*

4.2 Two and Higher Dimensions

Next, we consider the rectangle-containing-points problem in two dimensions. Here we are given a set of N_1 points in 2D and a set of N_2 rectangles. Set $\text{IN} = N_1 + N_2$. The goal is to report all (point, rectangle) pairs such that the point is inside the rectangle. The basic idea is to impose a canonical decomposition on the x-axis to break down the problem into many instances of the 1D problem. Although canonical decomposition is a well-known technique, applying it in the MPC model has some technical challenges. For simplicity, we will assume that p is a power of 2, which does not affect the asymptotic results.

Step (1): Sorting into atomic slabs and canonical slabs. We sort all the x-coordinates, including those of the points, as well as the left and right x-coordinates of the rectangles. After the sorting, each server has IN/p x-coordinates, which divides the whole x-axis into p slabs, which are labeled as $1, 2, \ldots, p$ in increasing order along the x-axis. We call these slabs *atomic slabs*. We impose a binary tree over these p atomic slabs in the standard fashion, where each node in the binary tree corresponds to a *canonical slab* (thus, an atomic slab is also a canonical slab), as shown in Figure 2. We decompose each rectangle into $O(\log p)$ disjoint pieces: a left piece and a right piece that partially cover an atomic slab, and at most $O(\log p)$ middle pieces, each spanning one canonical slab as large as possible. For example, σ_1 in Figure 2 is decomposed into four pieces: a left piece that falls inside slab 1, a right piece that falls inside slab 7, and 3 middle pieces that span canonical slabs 2, 3–4, and 5–6, respectively.

All the join results between a point and a left/right piece can be found easily. Each server holds $O(\text{IN}/p)$ x-coordinates, which correspond to at most $O(\text{IN}/p)$ points and at most $O(\text{IN}/p)$ left/right

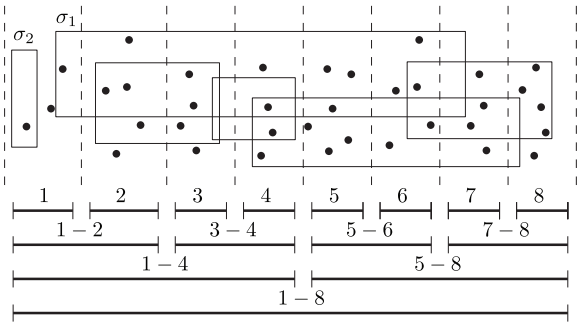

Fig. 2. Rectangles-joining-points. In this example, the atomic slabs are $\{1, 2, 3, 4, 5, 6, 7, 8\}$ and the canonical slabs are $\{1, 2, 3, 4, 5, 6, 7, 8, 1-2, 3-4, 5-6, 7-8, 1-4, 5-8, 1-8\}$.

pieces whose left/right x-coordinates are stored at this server. Thus, all of these join results can be found locally after the sorting.

Step (2): Middle pieces—Reduction to the 1D case. We are left with finding the join results between a point and a middle piece of a rectangle. Note that a middle piece of a rectangle is not just a canonical slab on the x-axis, but an implicit rectangle induced by this canonical slab and the y-projection of the original rectangle. The idea is to "collect," for each canonical slab s, all the middle pieces that span s, as well as all the points that fall inside s. Because the x-coordinates of all these points must fall in the x-projection of these middle pieces (which is just the canonical slab), the problem will reduce to the 1D problem, where we need to find all the (point, middle piece) pairs where the y-coordinate of the point falls inside the y-projection of the middle piece. Then we can invoke the 1D algorithm to solve all of these 1D instances in parallel. This is exactly the classical technique of using a canonical decomposition to reduce a 2D problem to many 1D instances. However, in the MPC model, we face the following difficulties (which are trivial in a centralized model): (1) how to collect the inputs of each 1D problem? and (2) how to allocate the p servers to these 1D instances so as to ensure a balanced load? We address each in turn.

Step (2.1): Collecting inputs for each canonical slab. In the MPC model, it is not possible to actually collect all the inputs of a canonical slab to one server because there are too many (e.g., the largest canonical slab has all the points as its inputs). By "collect," we mean that for each canonical slab s, we will attach its id to all its inputs so that when a number of servers are allocated later to solve the 1D problem associated with s, its inputs can be sent to these servers, by just looking at the canonical slab id's attached to the points and middle pieces.

For each point, it is easy to figure out the $O(\log p)$ canonical slabs it falls into, by simply looking at the atomic slab it belongs to. For each interval, we first find the two atomic slabs containing its two endpoints. Note that these two pieces of information are held by two different servers, so we need to bring them together by another sorting step (say, sorting by the id of the interval). Then from the two atomic slabs, we have enough information to break the interval into $O(\log p)$ middle pieces, each corresponding to a canonical slab.

Step (2.2): Computing IN(s) and OUT(s). Suppose that canonical slab s has $N_1(s)$ points and $N_2(s)$ middle pieces. Its input size is thus $\text{IN}(s) = N_1(s) + N_2(s)$. Define $\text{OUT}(s)$ as the output size of the 1D instance on s. The allocation of servers will depend on $\text{IN}(s)$ and $\text{OUT}(s)$. As each point belongs to $O(\log p)$ canonical slabs and each interval has $O(\log p)$ middle pieces, the total input size of all instances is $\sum_s \text{IN}(s) = O(\text{IN} \cdot \log p)$. Since each rectangle is broken up into disjoint middle pieces, we have $\sum_s \text{OUT}(s) \leq \text{OUT}$.

All the IN(s)'s can be computed by sum-by-key, using the canonical slab as the key. To count the OUT(s)'s, we invoke an instance of step (1) of the 1D algorithm for each canonical slab s. However, the load of step (1) of the 1D algorithm depends on its input size. Thus, to ensure a uniform load, we allocate $p_s = \lceil p \cdot \frac{\text{IN}(s)}{\text{IN} \log p} \rceil$ servers for canonical slab s (this uses $O(p)$ servers in total), with each server having load $O(\text{IN}(s)/p_s) = O(\frac{\text{IN}}{p} \log p)$.

Step (2.3): Solving 1D instances. Finally, we allocate servers to the $O(p)$ canonical slabs to solve the 1D problem associated to each. Since the load of the 1D algorithm depends on both the input and output size, we need to take both into account when allocating the servers. More precisely, we allocate

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

servers for a canonical slab s and invoke steps (2) and (3) of the 1D algorithm for all the canonical slabs in parallel. Plugging in the result of Theorem 4.1 yields the following result.

Theorem 4.2. *There is a deterministic algorithm for the rectangles-containing-points problem in 2D that runs in $O(1)$ rounds with $O(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{p} \log p)$ load.*

The algorithm can be extended to higher dimensions using the same idea presented earlier. More precisely, we can reduce a d-dimensional problem to many $(d-1)$-dimensional instances. The algorithm is the same as the preceding 2D algorithm. We build a canonical decomposition on the first dimension, break up each rectangle into a left piece, a right piece, and $O(\log p)$ middle pieces. The join results on the left/right pieces can be found easily with $O(\text{IN}/p)$ load, whereas we invoke $O(p)$ instances of the $(d-1)$-dimensional algorithm to find the join results of the middle pieces. Since the total input size of the $(d-1)$-dimensional instances is $O(\text{IN} \log p)$ while the total output size remains the same, we incur an extra $O(\log p)$ factor in the input-dependent term every time the dimensionality is reduced by one. Therefore, we obtain the following.

Theorem 4.3. *There is a deterministic algorithm for the rectangles-containing-points problem in d dimensions that runs in $O(1)$ rounds with $O(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{p} \log^{d-1} p)$ load.*

5 SIMILARITY JOIN UNDER ℓ_2

As mentioned in Section 2.1, we will study the halfspaces-containing-points problem that generalizes similarity joins under ℓ_2 distance. Compared to the ℓ_1/ℓ_∞ case, a key challenge in this problem 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 OUT = 1 or 0 by sampling requires us to essentially sample the whole dataset), but in that case, the input-dependent term will dominate the load, and we do not need to know the value of OUT anyway.

5.1 Useful Tools From Computational Geometry

5.1.1 Sampling With Threshold Approximation. We first mention the *θ-thresholded approximation*. A θ-thresholded approximation of x is an estimate $\hat{x}$ such that (1) if $x \geq \theta$, then $\frac{x}{2} < \hat{x} < 2x$; (2) if $x < \theta$, then $\hat{x} < 2\theta$. It captures our need of sampling when the output size is large enough. The following result first relates thresholded approximations with random sampling.

Theorem 5.1 ([18, 25]). *For any $q > 1$, let S be a random sample with replacement 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[3] Δ.*

5.1.2 Partition Tree. 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 simplex $\Delta(v)$, which encloses all the points stored at the leaves below v. For any v, the simplexes of its children do not overlap. In particular, this implies that all the leaf simplexes are disjoint. Chan [11] presented an algorithm to construct a b-partial partition tree with the following properties in Theorem 5.2. Their construction of a partition tree is a recursive process of cutting the space into disjoint simplexes.

Theorem 5.2 ([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 each as a simplex such that any hyperplane intersects $O((n/b)^{1-1/d})$ simplexes of the tree.*

In Chan's construction, all the leaf simplexes are disjoint and their union is the whole space. It only guarantees that each leaf simplex contains at most b points but offers no lower bound, whereas we will need each leaf to have $\Theta(b)$ points. This can be easily achieved, however. Suppose that Chan's b-partial partition tree has at most $c \cdot n/b$ leaves for some constant c. A leaf simplex is *big* if it contains at least $b/2c$ points; otherwise, it is *small*. Observe that there must be at least $n/2b$ big leaf simplexes; otherwise, all the leaves together would contain less than $n/2b \cdot b + cn/b \cdot b/2c = n$ points. Then, we merge each big simplex with an equal number of small simplexes, and denote this union of simplexes as a *cell* so that each cell has $\Theta(b)$ points (Figure 3). Note that the small simplexes can be allocated to the big simplexes arbitrarily, and each cell is not necessarily connected. Since each cell consists of one big simplex and at most $\frac{cn/b}{n/2b} = 2c$ small simplexes, which is a constant, we say that such a cell has constant *description size*. Note that on any cell with constant description size, any standard geometric operation (e.g., testing if a point falls inside the cell or if a hyperplane intersects the cell) takes $O(1)$ time. The following corollary summarizes the properties we need from this modified b-partial partition tree. In particular, we will not need the internal nodes of the tree.

Corollary 5.3. *Given n points in $\mathbb{R}^d$ and a parameter $b < n/\log^{\omega(1)} n$, we can find $O(n/b)$ disjoint cells such that (1) each cell has constant description size, (2) each cell contains $\Theta(b)$ points, and (3) any hyperplane intersects $O((n/b)^{1-1/d})$ cells.*

5.2 The Algorithm

Let q be a parameter such that $1 < q < p$, where p is the number of servers. The value of q will be determined later.

[3]Simplex generalizes the notion of triangle in 2D space (e.g., in 3D space, a simplex is a tetrahedron). Although the precise definition of a d-dimensional simplex is somewhat technical, for the following discussion, the reader can simply regard it as a polytope with a constant number of sides.

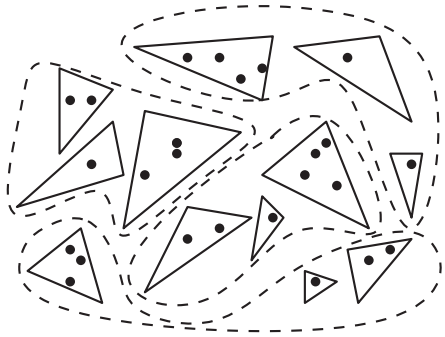

Fig. 3. The triangles are the leaf simplexes of the original partition tree, where each simplex contains at most four points but can contain as few as one point. Suppose thatsimplexes with at least three points are big simplexes and other simplexes are small simplexes. We merge each big simplex with an equal number of smaller simplexes, as shown by the dashed regions.

Step (1): Constructing a partition tree. We ask all servers to randomly sample $\Theta(q \log p)$ points in total and send all the sampled points to one server. The server builds the modified $\Theta(\log p)$-partial partition tree on the sampled points. By Theorem 5.1 and Corollary 5.3, we obtain $O(q)$ disjoint cells of constant description size. Note that the set of cells is a subdivision of the whole space, so it contains all points in R_1. With probability $1 - 1/p^{\Omega(1)}$, every cell contains $O(N_1/q)$ points of the original dataset. Any hyperplane always intersects $O(q^{1-\frac{1}{d}})$ cells due to the property of the partition tree. We broadcast these cells to all servers. This step incurs a load of $O(q \log p)$.

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 intersected by its bounding halfplane. Note that there are $O(q^{1-\frac{1}{d}})$ such cells by Corollary 5.3. Note that this can be done locally, as we have broadcast all cells to all servers. Then for each cell Δ, we compute the number of halfspaces whose bounding halfplane intersects Δ, denoted as $P(\Delta)$. This is a sum-by-key problem on a total of $\sum_\Delta P(\Delta) = O(N_2 \cdot q^{1-\frac{1}{d}})$ key-value pairs, and thus the load is

$$O\left(\frac{N_2}{p} \cdot q^{1-\frac{1}{d}}\right). \tag{3}$$

For each cell Δ, we allocate $p_\Delta = \lceil p \cdot \frac{P(\Delta)}{\sum_\Delta P(\Delta)} \rceil$ servers to compute the join between the $\Theta(\frac{N_1}{q})$ points in the cell and these $P(\Delta)$ halfspaces partially covering Δ. The total number of servers needed is $O(p)$. Invoking the hypercube algorithm to compute their Cartesian product incurs a load of

$$O\left(\sqrt{\frac{\frac{N_1}{q} \cdot P(\Delta)}{p_\Delta}} + \frac{\frac{N_1}{q} + P(\Delta)}{p_\Delta}\right) = O\left(\sqrt{\frac{N_1 N_2}{p q^{\frac{1}{d}}}} + \frac{N_1}{q} + \frac{N_2 q^{1-\frac{1}{d}}}{p}\right). \tag{4}$$

Choosing $q = p^{\frac{d}{2d-1}}$ balances the terms in (3) and (4), and the load becomes $O(\frac{\mathrm{IN}}{q}) = O(\frac{\mathrm{IN}}{p^{d/(2d-1)}})$.

Step (3): Fully covered cells. In the intervals-containing-points algorithm, fully covered intervals are dealt with in a way similar to the partially covered intervals, as we can compute OUT exactly and set the right slab size. In this case, we may have used a cell size (i.e., N_1/q) that is too small in relation to OUT. This would result in too many replicated halfspaces to be distributed, 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, denoted as $\widehat{F}(\Delta)$. 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. Consider any particular Δ. By the Chernoff inequality, with probability at least $1 - 1/p^{O(1)}$, we get an $(\frac{N_2}{q})$-thresholded approximation of $F(\Delta)$. Then applying a union bound on the $O(q)$ cells, we get an $(\frac{N_2}{q})$-thresholded approximation for every $F(\Delta)$ with probability at least $1 - q/p^{O(1)} = 1 - 1/p^{O(1)}$, as long as the hidden constant in the $O(q \log p)$ sample size is sufficiently large. We use these approximate $F(\Delta)$'s to compute $\widehat{K}$ (i.e., $\widehat{K} = \sum_{\Delta} \widehat{F}(\Delta)$), which is then an N_2-thresholded approximation of the true value of K.

Step (3.2): If $\widehat{K} < \frac{\text{IN} \cdot p}{q}$. As $\widehat{K}$ is an N_2-thresholded approximation of K, $\widehat{K} < \frac{\text{IN} \cdot p}{q}$ would imply that $K = O(\frac{\text{IN} \cdot p}{q})$ for $q = o(p)$. 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 and 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, as the halfspaces cover too many cells. This means we have used a cell size too small, and we need to restart the whole algorithm with a new q'. Note that if $\widehat{K} > \frac{\text{IN} \cdot p}{q}$, then $\widehat{K}$ must be a constant-factor approximation of K. Let $\widehat{\text{OUT}} = \widehat{K} \cdot \frac{\text{IN}}{q}$, and $\widehat{\text{OUT}}$ is also a constant-factor approximation of the remaining output size, and thus $\widehat{\text{OUT}} = O(\text{OUT})$. We set $q' = \text{IN}/\sqrt{\frac{\widehat{\text{OUT}}}{p}}$ where $q' < q$.

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. With the new q', step (1) has load $O(q' \log p) = O(q \log p)$, and step (2) has load $O(\frac{\text{IN}}{q'}) = O(\sqrt{\frac{\text{OUT}}{p}})$. In the step (3.1), 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 $K' = O(Kq'/q)$, as 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, $\widehat{K}' = O(\frac{\text{IN} \cdot p}{q'})$ always holds by the following fact:

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

Then it always reaches step (3.2), with load complexity $O(\sqrt{\frac{\text{OUT}}{p}} + \frac{\text{IN}}{q'}) = O(\sqrt{\frac{\text{OUT}}{p}})$. Therefore, the re-execution, if it takes place, must have load $O(\sqrt{\frac{\text{OUT}}{p}})$. Combining with the load of the first execution, we obtain the following result.

Theorem 5.4. *There is a randomized algorithm that solves the halfspaces-containing-points problem in $O(1)$ rounds and load*

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

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 $O(\text{IN}/p^{\frac{d}{2d-1}})$ approaches $O(\text{IN}/\sqrt{p})$ 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 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 OUT(cr), which is the output size when the distance threshold of the similarity join is made c times larger, for some constant $c > 1$.

6.1 Locality Sensitive Hashing

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 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 [28].

Let $\text{dist}(\cdot, \cdot)$ 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 $\text{dist}(x, y) \leq r$; and (2) $\Pr[h(x) = h(y)] \leq p_2$ if $\text{dist}(x, y) \geq cr$. In addition, we require $\mathcal{H}$ to be *monotone*—that is, for a randomly chosen $h \in \mathcal{H}$, $\Pr[h(x) = h(y)]$ is a non-increasing function of $\text{dist}(x, y)$. This requirement is not in the standard definition of LSH, but the LSH constructions for most metric spaces satisfy this property, include Hamming [20], ℓ_1 [12], ℓ_2 [5, 12], Jaccard [9], and so forth. Meanwhile, all of these LHS hash functions can be represented efficiently in $O(d)$ space. Since in this section each tuple is a point with d coordinates and we measure the load in terms of tuples, we can consider each LSH function to have size the same as $O(1)$ tuples.

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 [5, 12, 20]. 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, whereas $\rho = \frac{\log p_1}{\log p_2}$ is kept fixed, or equivalently, $p_2 = p_1^{1/\rho}$.

6.2 The Algorithm

In the description of our algorithm that follows, we leave p_1, p_2 unspecified, which will be later determined in the analysis. The algorithm proceeds in the following three steps:

(1) Choose $q = 3 \cdot 1/p_1 \cdot \ln \mathrm{IN}$ hash functions $h_1, \ldots, h_q \in \mathcal{H}$ randomly and independently, and broadcast them to all servers.
(2) For each tuple x, make q copies, and attach the pair $(i, h_i(x))$ to each of these copies, for $i = 1, \ldots, q$.
(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 $\mathrm{dist}(x, y) \le r$.

Theorem 6.1. *Assume that 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*

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

The algorithm reports all join results with probability at least $1 - 1/\mathrm{IN}$.

Proof. Correctness of the algorithm follows from standard LSH analysis: for any two tuples x, y with $\mathrm{dist}(x, y) \le r$, the probability that they join on one h_i is at least p_1. Across q independently chosen hash functions, the probability that they do not collide on any one of hash function is at most $(1 - p_1)^{3 \cdot 1/p_1 \cdot \ln \mathrm{IN}} \le e^{-3 \cdot \ln \mathrm{IN}} \le 1/\mathrm{IN}^3$. As there are at most IN^2 pairs of tuples with their distance smaller than r, the probability that any one of them is not reported is at most $1/\mathrm{IN}$ by the union bound. Then we have probability of at least $1 - 1/\mathrm{IN}$ to report all join results.

In the following, we analyze the load. Step (1) has load $\widetilde{O}(1/p_1)$, and step (2) is local computation. Thus, we only need to analyze step (3).

The total number of tuples generated in step (2) is $\widetilde{O}(\mathrm{IN}/p_1)$, which is the input size to the equi-join, denoted as IN_{LSH}. The output size, denoted as OUT_{LSH}, has its expectation bounded by

$$E[\mathrm{OUT}_{LSH}] = \widetilde{O}\left(\mathrm{OUT}/p_1 + \mathrm{OUT}(cr) + \mathrm{IN}^2/p_1^{1-1/\rho}\right).$$

The first term is for all pairs (x, y) such that $\mathrm{dist}(x, y) \le r$. They could join on every h_i. The second term is for (x, y)'s such that $r < \mathrm{dist}(x, y) \le cr$. 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 IN^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 3.1, and using Jensen's inequality $E[\sqrt{X}] \le \sqrt{E[X]}$, the expected load can be bounded by (the $\widetilde{O}$ of)

$$E\left[\sqrt{\frac{\mathrm{OUT}_{LSH}}{p}} + \frac{\mathrm{IN}_{LSH}}{p}\right] \le \frac{\sqrt{E\left[\mathrm{OUT}_{LSH}\right]}}{\sqrt{p}} + \frac{\mathrm{IN}_{LSH}}{p} \le \frac{\sqrt{\mathrm{OUT}/p_1 + \mathrm{OUT}(cr) + \mathrm{IN}^2/p_1^{1-1/\rho}}}{\sqrt{p}} + \frac{\mathrm{IN}}{pp_1}$$
$$\le \sqrt{\frac{\mathrm{OUT}}{pp_1}} + \sqrt{\frac{\mathrm{OUT}(cr)}{p}} + \mathrm{IN}\sqrt{\frac{1}{pp_1^{1-1/\rho}}} + \frac{\mathrm{IN}}{pp_1}.$$

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

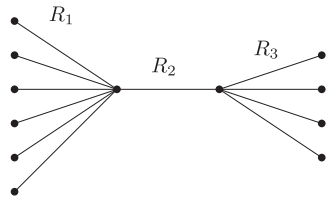

Fig. 4. An instance of a 3-relation chain join.

Remark 1. Note that since $0 < \rho < 1$, the input-dependent term is always better than performing a full Cartesian product. The output-term $O(\sqrt{\frac{\text{OUT}(cr)}{p}})$ 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 is smaller than r or slightly above r. A worst-case scenario is all the 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 OUT(cr) pairs to make sure that it does not miss any true join results. Finally, the term $O(\sqrt{\frac{\text{OUT}}{p^{1/(1+\rho)}}})$ is also worse than the bound $O(\sqrt{\frac{\text{OUT}}{p}})$ we achieved in earlier sections. This is perhaps the best one can hope for as well, if $O(1)$ rounds are required: 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 present a negative result, showing that this is not possible for the 3-relation equi-join $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$. This means that one cannot hope to achieve output-optimality for arbitrary multi-way joins. Precisely characterizing the class of joins for which output-optimal algorithms exist seems an interesting and challenging direction of future work.

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, and 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.

However, consider the instance shown in Figure 4, 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 $O(\sqrt{\text{OUT}/p})$. In the following, we show that this is not possible either, assuming any meaningful input-dependent term.

Theorem 7.1. *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.

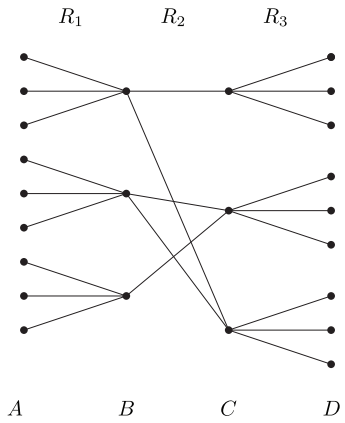

Fig. 5. A randomly constructed hard instance.

Note that there is an algorithm for the 3-relation chain join with load $\tilde{O}(\text{IN}/\sqrt{p})$ [23], without any output-dependent term. This means that it is meaningless to introduce the output-dependent term $O(\sqrt{\text{OUT}/p})$.

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

The construction is illustrated in Figure 5. Let $N = \frac{\text{IN}}{3}$. More precisely, attribute A and D each have N distinct values. Each distinct value of A appears in one tuple in R_1, and each distinct value of D appears in one tuple in R_3. 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, whereas R_2 is probabilistic with N tuples in expectation, so the expected input size is IN. The output size is expected to be NL. By the Chernoff inequality, the probability that the input size or output size 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. Furthermore, we allow a server in each round to retrieve L tuples from each of R_1 and R_3, for a total of $2L$ tuples, which actually exceeds the load constraint by a factor of 2. 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 to maximize its output size. Here, a group in R_1 (respectively, R_2) means all tuples sharing the same value on B (respectively, C). Suppose that two groups in R_1, say, g_1 and g_2, are not loaded in full by a server: $x_1 < \sqrt{L}$ tuples of g_1 and $x_2 < \sqrt{L}$ tuples of g_2 have been loaded. Suppose that they respectively join with y_1 and y_2 tuples in R_3 that are loaded by the server. Note that they will produce $x_1y_1 + x_2y_2$ join results. Without loss of generality, assume that $y_1 \geq y_2$. Now consider the alternative where the server loads $x_1 + 1$ tuples of g_1 and $x_2 - 1$ tuples of g_2. Then this would produce $(x_1 + 1)y_1 + (x_2 - 1)y_2 = x_1y_1 + x_2y_2 + y_1 - y_2 \geq x_1y_1 + x_2y_2$ tuples. This means that by moving one tuple from g_2 to g_1, the server can only get more join results (at least not less). We can move tuples from one group to

another as long as there are two non-full groups. Eventually, we arrive at a configuration where all groups of R_1 are loaded by the server in full, without decreasing the output size. Next, we apply the same transformation to the groups of R_3 to make all its groups full as well.

The preceding argument implies that we can assume each server in each round loads $\sqrt{L}$ full groups from R_1 and $\sqrt{L}$ full groups from R_3, with $2L$ tuples in total. In the following, we show that on a random instance constructed as earlier, 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 $O((\frac{N}{\sqrt{L}})^{2\sqrt{L}})$ different choices of $\sqrt{L}$ groups from R_1 and $\sqrt{L}$ groups from R_3. Thus, 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 3.2, 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 $N/\log^2 N > c_3 \cdot p^3$ for some sufficiently large constant c_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 $O(\frac{L^3 p}{N})$ results in a constant number of rounds. Thus, we have

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

in other words,

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

Suppose that 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 PRACTICAL SIMILARITY JOIN ALGORITHMS

The algorithms designed in previous sections have achieved output-optimality in theory, but at the expense of using a large, although still constant, number of rounds. In large-scale distributed systems, each round incurs a substantial synchronization overhead, and the benefit of an asymptotic smaller load can be easily offset by the large system overhead. In this section, we describe a practical version of the equi-join algorithm from Section 3 and the one-dimensional similarity join algorithm from Section 4.1. Specifically, the practical versions of the algorithms need just one *heavy round* and a constant number of *light rounds*, where a round is *light* if its load is $\tilde{O}(p)$ and *heavy* otherwise. Theoretically speaking, however, a light round might be even heavier than a heavy round if $p > L$, where L is the optimal load of the algorithm. Thus, the practical version of the algorithm is theoretically optimal only when $p < L$, but this is always the case in practice. In fact, in most massively parallel systems in practice, p is usually on the order of hundreds, whereas L is at least 10^6, and thus the benefit of using multiple light rounds to avoid a heavy round will be obvious. Indeed, in our experiments, we observe that the total time spent in all the light rounds accounts for less than 10% of the wall-clock time. This means the heavy round dominates the cost, and it is important to restrict the algorithms to using only one heavy round.

8.1 Primitive Operations

We first show how to implement the primitive operations from Section 2.2 in $O(1)$ light rounds and at most one heavy round.

Sorting. The theoretically optimal BSP sorting algorithm by Goodrich is very complicated and not practical. In practice, a sampling-based algorithm, such as TeraSort [27, 31], is often used instead, which operates in the following two steps:

(1) Take a random sample of $O(p \log p)$ elements, and collect them to a master server. The master server sorts this sample, takes the $p - 1$ splitters, denoted as $s_1, \ldots, s_{p-1}$, that split this sample evenly into p partitions, and broadcasts these splitters to all the servers. This step can be implemented in two light rounds.
(2) Upon receiving the $p - 1$ splitters, each server scans its own elements. For each element x, the server finds the two consecutive splitters s_i and s_{i+1} (define $s_0 = -\infty, s_p = \infty$) such that $s_i \leq x < s_{i+1}$ and sends x to server i. This step requires one heavy round. After all the shuffling, each server locally sorts all elements that it has received.

This sorting algorithm will be used in our sort-merge-join–based equi-join algorithm. Thus, strictly speaking, the practical version of our algorithm is still randomized. However, as shown in Tao et al. [31], the number of elements received by any server is at most $O(\text{IN}/p)$ with high probability $1 - 1/p^{\Omega(1)}$, where the exponent depends on the hidden constant in the sample size. In our implementation, we choose a sample size of $10p \log p$, which makes this probability very close to 1.

All prefix-sums. Instead of the full BSP algorithm for all prefix-sums [17], we use the following simple variant, which can be done in two light rounds. Recall that in the all prefix-sums problem,

a large array A is stored on p servers in a consecutive manner, and the goal is to compute $S[j] = A[1] \oplus \cdots \oplus A[j]$ for every j, where $\oplus$ is an associative operator:

(1) Each server i computes the partial sum on its local sub-array, denoted $B[i]$, and sends it to a master server.
(2) The master server computes the prefix-sums on the partial sums elements received—that is, $C[i] = B[1] \oplus \cdots \oplus B[i-1]$ and sends it to server i, for $i = 2, \ldots, p$. Define $C[1] = 0$. Then server i computes the prefix-sums for its local sub-array sequentially starting from $C[i]$.

Multi-Numbering. The multi-numbering problem was solved in Section 2.2.2 by sorting all elements and then running all prefix-sums, requiring one heavy round and $O(1)$ light rounds. We observe that if the number of distinct keys is $O(p)$, then we do not have to sort all tuples, thus avoiding the heavy round. To do so, we first ask each server i to compute a partial-count $N(i, v)$, which is the number of elements with key v at server i. Then we sort these partial-counts by v. Since there are only $O(p^2)$ partial-counts, the load in the "heavy round" of TeraSort is only $O(p^2/p) = O(p)$, making it a light round. Then we run the all prefix-sums algorithm on these $O(p^2)$ partial-counts, using the same definition of $\oplus$ as in Section 2.2.2. This will give us, for each distinct v, the prefix-sums $C(i, v) = N(1, v) + \cdots + N(1, i-1)$, $i = 2, \ldots, p$. Finally, we send the $C(i, v)$'s to server i for all v, which then assigns consecutive numbers to all tuples with key v, starting from $C(i, v)$ (define $C(1, v) = 0$).

Therefore, when the number of distinct keys is $O(p)$, multi-numbering can be solved with $O(1)$ light rounds and no heavy round.

Sum-by-Key. Similar to multi-numbering, the sum-by-key problem can be also solved with $O(1)$ light rounds and no heavy round when the number of distinct keys is $O(p)$. We first ask each server i to compute the partial-sum $S(i, v)$ of all elements with key v, for each distinct key v. Then we sort these $O(p^2)$ partial-sums by v and run all prefix-sums using the same definition of $\oplus$ as in Section 2.2.2.

8.2 A Practical Equi-Join Algorithm

The practical equi-join algorithm for computing $R_1(A, B) \bowtie R_2(B, C)$ is described in the following. It follows the TeraSort framework but replaces certain steps appropriately. As in Section 3, define $R_i(v) = \sigma_{B=v} R_i$ and $N_i(v) = |R_i(v)|$:

(1) The first step is the same as in TeraSort—that is, we take a sample of $O(p \log p)$ tuples from $R_1 \cup R_2$ and collect them to one server. The server sorts the sample on attribute B, breaking ties using other attributes of the tuples (we assume that no two tuples are the same on all attributes). The server then finds the $p-1$ splitters, denoted $s_1, \ldots, s_{p-1}$, from the sample and broadcasts them to all servers. This step requires two light rounds. Let $C = \pi_B\{s_1, \ldots, s_{p-1}\}$ be the set of distinct B values of the splitters.
(2) Upon receiving the $p-1$ splitters, each server scans its own tuples. For each tuple t, the server finds the two consecutive splitters, say s_i, and s_{i+1}, such that $s_i \le x < s_{i+1}$. Ties are broken using the same tie breaker as in step (1). If $x.B = s_i.B$ or $x.B = s_{i+1}.B$, x is said to be a *crossing tuple* and *non-crossing* otherwise. One simplification we have made is not to compute OUT exactly, which would require a heavy round. Instead, we only compute the join size on the crossing tuples—that is, we compute $\text{IN}_c = \sum_{v \in C} N_1(v) + N_2(v)$, $\text{OUT}_c = \sum_{v \in C} N_1(v) N_2(v)$ and use IN_c, OUT_c in place of IN, OUT, respectively, in formula (2) for allocating the servers. Note that OUT_c can be computed in $O(1)$ light rounds by running the sum-by-key algorithm with the $\le p-1$ distinct values in C as keys. We then run the

multi-numbering algorithm to assign consecutive numbers to the tuples for each distinct value $v \in C$, which also takes $O(1)$ light rounds as described earlier.

(3) With high probability, the number of non-crossing tuples between each pair of consecutive splitters s_i, s_{i+1} is $O(\text{IN}/p)$, so we send these tuples to server i, which will compute their join results. For each $v \in C$, we allocate

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

servers and invoke the deterministic Cartesian product algorithm. Note that all crossing tuples for each distinct value $c \in C$ have been numbered as well. Thus, all the Cartesian products and the join of non-crossing tuples can be computed in parallel in one heavy round.

Theorem 8.1. *The preceding equi-join algorithm runs in $O(1)$ light rounds and one heavy round, where each light round has a load of $O(p \log p)$, whereas the heavy round has load $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$. These bounds hold with probability at least $1 - 1/p^{O(1)}$.*

Proof. The $O(p \log p)$ load in the light rounds is due to the first step of collecting the sample to the master server; all other light rounds actually have load only $O(p)$. For the heavy round, each server receives $O(\frac{\text{IN}}{p})$ non-crossing tuples and $O(\frac{\text{IN}_c}{p} + \sqrt{\frac{\text{OUT}_c}{p}}) = O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$ crossing tuples. □

8.3 A Practical Intervals-Containing-Points Algorithm

Recall that in the intervals-containing-points problem, we are given N_1 points and N_2 intervals, and the goal is to report all the (point, interval) pairs such that the interval contains the point. Note that the ℓ_1/ℓ_∞ similarity join problem reduces to the special case of this problem where all intervals have length $2r$:

(1) The first step is the same as that in the equi-join algorithm—that is, we take a sample of $O(p \log p)$ points and collect them to one server. This server finds $p - 1$ splitters as $\{s_1, s_2, \ldots, s_{p-1}\}$ (define $s_0 = -\infty, s_p = \infty$) and broadcasts them to all servers. Each consecutive pair of splitters defines a slab. Note that there are p slabs labeled as $1, 2, \ldots, p$ (i.e., (s_{i-1}, s_i) defines slab i). With high probability, the number of points falling into each slab is $O(\text{IN}/p)$. Same as before, this step requires two light rounds.

(2) Upon receiving the $p - 1$ splitters, each server scans its own data. Similarly, we do not compute OUT exactly. Instead, we compute the join size of points and intervals fully covering slabs—that is, $\text{OUT}_f = \sum_i P(i) \cdot F(i)$, where $P(i)$ is the number of points in slab i and $F(i)$ is the number of intervals fully covering slab i. For each point, as well as each endpoint of intervals, the servers find which slab it falls into (i.e., a pair of consecutive splitters s_i, s_{i+1} for x such that $s_i \le x < s_{i+1}$). We first run the sum-by-key algorithm to compute all $P(i)$'s, where each point x with $s_{i-1} \le x < s_i$ is considered to have key i and weight 1. An interval $[x, y]$ with $s_{i-1} \le x < s_i$ and $s_j \le y \le s_{j+1}$ fully covers slabs $i, i+1, \ldots, j$. We ask each server to compute p partial-counts in term of (i, j) for $i = 1, 2, \ldots, p$, indicating that there are j intervals fully covering slab j in its local data. There would be p^2 such pairs in total. Then we just run the sum-by-key algorithm to compute all $F(i)$'s—that is, each pair (i, j) is considered to have key i and weight j. All $P(i)$'s and $F(i)$'s will be sent to the master server to compute OUT_f. Note that this step only requires $O(1)$ light rounds.

(3) When OUT_f is larger than IN^2/p, we need to merge slabs into larger ones such that each slab has size $b' = \max\{\frac{\text{IN}}{p}, \sqrt{\frac{\text{OUT}_f}{p}}\}$. If this happens, the master server just broadcasts new splitters to all servers and redefines all slabs. For each slab s, we count the number of intervals partially and fully covering it, denoted as $F(s)$ and $G(s)$, respectively. All $F(s)$'s and $G(s)$'s can be computed similarly in $O(1)$ light rounds using the sum-by-key algorithm. Note that $\sum_s F(s) \leq 2 \cdot \text{IN}$ and $\sum_s b'G(s) \leq \text{OUT}_f$. We then run the multi-numbering algorithm to assign consecutive numbers to the intervals covering s for each slab s, which also takes $O(1)$ light rounds as described earlier.

(4) For each slab s, we allocate $p_s = \lceil \frac{F(s)+G(s)}{b'} \rceil$ servers and invoke the deterministic Cartesian product algorithm. Note that all intervals covering s for each slab s have been numbered as well. All the Cartesian products can be computed in parallel in one heavy round. This step only uses $O(p)$ servers since $\sum_s \lceil \frac{F(s)+G(s)}{b'} \rceil \leq p + \sum_s \frac{F(s)}{b'} + \sum_s \frac{G(s)}{b'} \leq p + \frac{2 \cdot \text{IN}}{b'} + \frac{\text{OUT}_f}{b'^2} \leq 4p$.

Theorem 8.2. *The preceding intervals-containing-points algorithm runs in $O(1)$ light rounds and one heavy round, where each light round has a load of $O(p \log p)$, whereas the heavy round has load $O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$. These bounds hold with probability at least $1 - 1/p^{O(1)}$.*

Proof. The $O(p \log p)$ load in the light rounds is due to the first step of collecting the sample to the master server; all other light rounds actually have load only $O(p)$. For the heavy round, each server receives $O(b') = O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}_f}{p}}) = O(\frac{\text{IN}}{p} + \sqrt{\frac{\text{OUT}}{p}})$ points and intervals. □

Unfortunately, we do not know if the rectangles-containing-points algorithm can be made to run in one heavy round and $O(1)$ light rounds. Furthermore, the logarithmic factor will cause a quick degradation of the algorithm in higher dimensions. Similarly, the ℓ_2 similarity join algorithm is unlikely to be competitive in practice, due to its complexity and the large hidden constants. Thus, these results are mostly of theoretical interest only.

8.4 Implementation Details

We have implemented the algorithms in Spark [33]. The main abstraction Spark provides is a resilient distributed dataset (RDD) that enables efficient data reuse in a broad range of applications. In Spark, each RDD is partitioned into a number of partitions. An operation on an RDD is performed by launching multiple tasks that each run on a partition in parallel across a cluster of computing nodes. Thus, each partition/task can be naturally modeled as a "server" in the MPC model. In our implementation, the two input relations R_1 and R_2 are stored as two RDDs, respectively. Note that the number of partitions/tasks, also called the *level of parallelism*, does not have to be equal to the number of physical processors in the system; in fact, the official Spark documentation recommends setting the level of parallelism to be two to three times the number of CPU cores in the cluster to allow the system to perform some dynamic load balancing at runtime. Thus, we set p to be three times the number of CPU cores in the cluster for steps (1) and (2) in both algorithms from Sections 8.2 and 8.3. Since the p_v servers allocated to each join key $v \in C$ in step (3) of the equi-join case, and the p_s servers allocated to each slab s in step (4) of the intervals-containing-points case, should be arranged in a grid with dimensions specified as in Section 2.2.5, which have to be integers, some rounding has to be done, and the total number of allocated servers might range from one to two times of value p. In this way, each of p servers participates in one or two Cartesian product computations.

The vanilla join operator provided by Spark performs a simple hash join. More precisely, all tuples with join value v are sent to server v mod p, which then computes the join of all tuples received. This requires a complicated and expensive *shuffle* operation that involves construction of hash tables, data serialization, and network I/O. We have implemented four algorithms: (1) the practical equi-join algorithm from Section 8.2 and the hash-based equi-join algorithm of Beame et al. [8], (2) the practical one-dimensional similarity join algorithm from Section 8.3, and (3) the LSH-based similarity join algorithm from Section 6.2. We did not implement our algorithms from scratch but leverage on Spark's own join operator.

In the practical equi-join algorithm, the idea is to "repackage" the join keys in a way such that the vanilla hash join will work just as how our algorithm would perform the join. More precisely, when our algorithm wants to send a tuple with join value v to server i, we turn it into a tuple with key $pv + i$. In the case of crossing tuples, which need to be sent to multiple servers, we use Spark's `flatMap` method to generate multiple tuples with different repackaged keys. It can be easily verified that the results from the vanilla hash join on the repackaged keys are exactly the same as those from the original join. Through this trick, we can implement our algorithm completely in the user space without modifying any existing code of Spark. The benefit of this approach is that our algorithm can automatically enjoy the improvements of any future updates to Spark's shuffle operation. Meanwhile, we point out that if one were to implement the algorithm inside the Spark core, we could have some (very) small savings by avoiding generating the intermediate repackaged tuples. The multi-numbering and sum-by-key algorithms described earlier use a constant number of light rounds, and the load in each round is $O(p)$. Because p is at most a few hundred in our experimental setting, we simply collect all the $O(p^2)$ partial-counts or partial-sums to the driver node, which then computes the prefix-sums and broadcasts them back to all workers using Spark's broadcast variables.

The hash-based equi-join algorithm has been implemented as a counterpart of the practical equi-join algorithm, with the following differences. First, instead of using crossing and non-crossing tuples, this algorithm uses information about the heavy hitters in the join values, where a join value is heavy if it appears more than IN/p times. The heavy hitters can be found by first finding the "local" heavy hitters on each server (i.e., those that appear more than IN/p^2 times on one server) and then collecting the local heavy hitters to the driver node, which then finds the global heavy hitters. Second, instead of computing the destination server i from the splitters, this algorithm uses a random hash function h to assign light hitter v to server $h(v)$. Similarly, we generate a repackaged tuple with join key $pv + h(v)$. Third, instead of assigning tuples for a heavy hitter deterministically to a row or a column in the grid, this algorithm uses a hash function on the A and C attributes of the two relations. Similarly, we use `flatMap` to generate multiple repackaged tuples for each such tuple.

The implementation of practical one-dimensional similarity join algorithm follows the same approach. The difference from the equi-join algorithms is that we need to assign join keys for points as well as intervals before invoking the Cartesian product algorithm. In our algorithm, the one-dimensional space is decomposed into a set of disjoint slabs in sorted order, whose ids are used as the join keys. Each slab induces one instance of Cartesian product. Note that each point participates in one Cartesian product, whereas each interval may participate in multiple Cartesian products. Implied by the algorithm from Section 2.2.5, servers allocated for each slab are arranged into one row such that points are broadcasted to the servers and interval are evenly distributed across the servers. Similarly, we use `flatMap` to generate multiple repackaged tuples for each point and interval. Note that for each pair of (point, interval) found by the Cartesian product, we report it if the point is indeed inside the interval.

Incorporating the LSH technique to the preceding equi-join algorithms, we are able to implement the LSH-based similarity join algorithm. The construction of the LSH family follows that in Gionis et al. [15], each hash function consists of multiple candidate hash keys, where each hash key is a set of bits of the input tuple, randomly sampled from the unary representation of the coordinates (real numbers are scaled up and rounded to integers). Given a similarity threshold, we first apply the LSH function to each tuple and use `flatMap` to generate multiple pairs for each candidate hash key. Then we use the hash key as the join key to perform a self-join by calling the equi-join algorithm as implemented earlier. For each pair of tuples in the join results, we compute their actual distance and report the result if it is indeed smaller than the similarity threshold. Thus, the algorithm will not return false positives but may miss a small number of true join results.

9 EXPERIMENTS

9.1 Experimental Setup

All experiments have been performed on a Microsoft Azure HDInsight cluster running Spark 2.1. By default, the cluster is set up with six worker nodes each having eight CPU cores, whereas we vary the number of worker nodes from one to six in studying the scalability of algorithms. Each worker node has 56GB of RAM, which is sufficient to keep all the data (raw and intermediate) so that even if all tuples are sent to one task, no data has to be swapped out to disk. Note that this is actually the ideal setting for the vanilla hash join algorithm, which may incur unbalanced loads when the data is highly skewed. If the worker nodes have smaller RAM, unbalanced loads can make the vanilla hash join algorithm even worse due to garbage collection and disk I/Os.

We have evaluated three equi-join algorithms: the vanilla hash join algorithm provided by Spark, later referred to as *Spark join*; the hash-based output-optimal join algorithm of Beame et al. [8], referred to as *hash join*; and our sort-based output-optimal join algorithm, referred to as *sort join*.

For similarity join in one dimension, we evaluated three algorithms: the deterministic Cartesian product algorithm in Section 2.2.5, referred to as *full join*; the LSH-based similarity join algorithm, referred to as *LSH join*; and our intervals-containing-points algorithm, referred to as *interval join*. In higher dimensions, we evaluated the *full join* and the *LSH join* using different equi-join algorithms, referred to as *LSH-Spark join*, *LSH-Hash join*, and *LSH-Sort join*.

As described earlier, we set p to be three times the number of CPU cores, namely 144 for our cluster with 48 cores, when evaluating all algorithms.

9.2 Datasets

We used both synthetic data and real data to test the performance of these algorithms.

For equi-join algorithms, we generated two RDDs of (key, value) pairs and performed their equi-join on the key. The values are randomly generated strings of length 8. The keys are generated according to some distribution. We tested two distributions. The first is the zipf distribution, where the frequency of the i-th distinct key is proportional to $i^{-\alpha}$. The parameter $\alpha \geq 1$ controls the skewness of the distribution: larger α means larger skew. The second distribution is simply uniform, but we vary the number of distinct keys.

For similarity join in one dimension, we generated two RDDs of point sets, one floating-point number for each point. The numbers are uniformally randomly generated from $[0, 1]$. In fact, we also tested the Gaussian distribution, and the results are similar. In high dimensions, we use the same COREL dataset as in Gionis et al. [15], which is a set of 15,000 64-dimensional points. Each point corresponds to the histogram of one distinct color image taken from the COREL library. Note that we use the same LSH family, so all three equi-join algorithms have the same input size and give exactly the same results but only differ in running time.

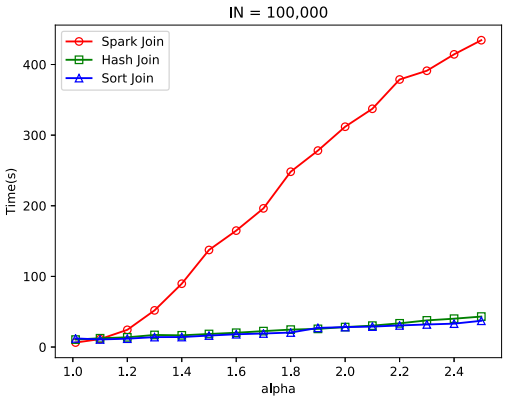

Fig. 6. Running times with different α.

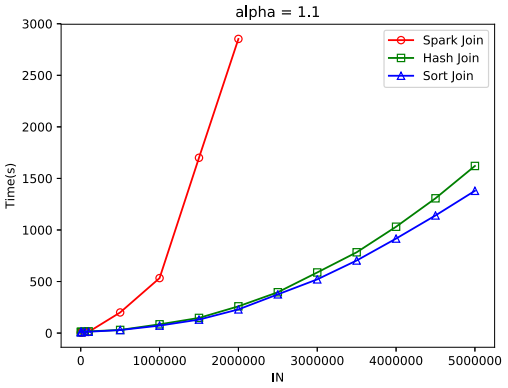

Fig. 7. Running times with different input size.

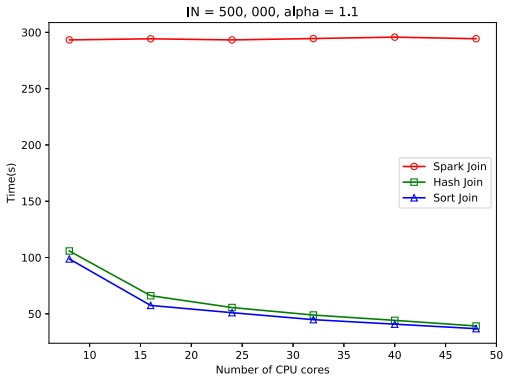

Fig. 8. Running times with different numbers of CPU cores on the zipf distribution.

9.3 Experimental Results

Results of equi-join on the zipf distribution. We have performed two sets of experiments, in which we vary α and data size, respectively. In the first set of experiments, the number of tuples in each relation is fixed at 100,000 and α increases from 1.0 to 2.5. From the results shown in Figure 6, we see an increasing gap in the running time between the Spark join and the two output-optimal algorithms. This is because the vanilla hash join algorithm used in Spark is bottlenecked by the heaviest key, as all tuples of the same key must be sent to the same server. As α increases, the skewness increases. When the data size is fixed, increasing α will increase the frequency of the heaviest key. However, the two output-optimal algorithms will allocate servers appropriately according to the frequencies, so they can much better balance the heavy keys with the light keys. Similarly, when we fix α and increase IN, the results in Figure 7 show a similar trend. This can also be explained by the same reason that increasing IN while keeping α fixed also increases the frequency of the heaviest key. We also tested the scalability of these algorithms by varying the number of worker nodes, and hence the total number of CPU cores in the cluster, and the result is shown in Figure 8. Both the two output-optimal algorithms will benefit from more CPU cores, whereas the Spark join almost stays the same, as all tuples of the heaviest key must be handled by the same CPU core, which is the bottleneck no matter how many workers are available.

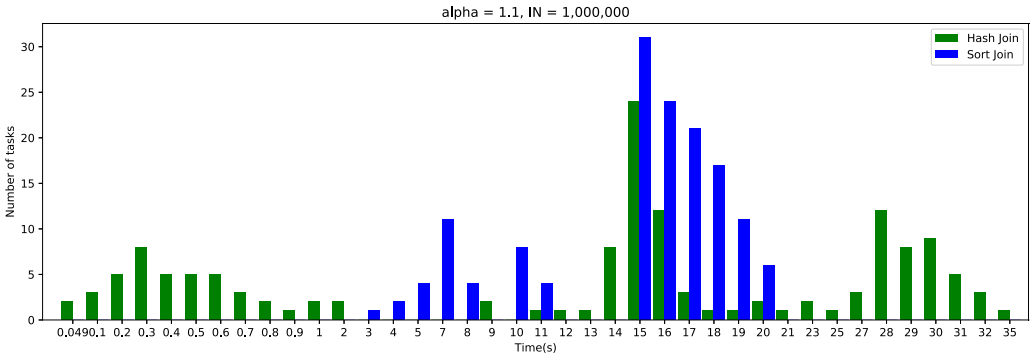

Fig. 9. Running time distribution of tasks on the zipf distribution.

However, the difference between the two output-optimal algorithms is small, with sort join slightly better. Although, theoretically speaking, the randomized hash join algorithm of Beame et al. [23] is sub-optimal by a logarithmic factor, the actual difference is much smaller. There are two explanations for this phenomenon. First, the analysis in Beame et al. [23] is assuming the worst input. On the zipf distribution, the maximum load is determined by the hypercube algorithm on the heavy hitters. Since each grid of servers only has a small number of rows and columns while a large number of tuples are hashed to the rows and columns, a random allocation can already achieve good balance. Indeed, from the classical balls-in-bins analysis, we know that if sufficiently many balls are thrown into a small number of bins, then with high probability the bin sizes are within a constant factor of each other.

The second reason is more system related. To see how the two algorithms allocate work to the servers, we took a closer look at the running time distribution of the tasks, which is shown in Figure 9. We see that the sort join algorithm indeed yields a more concentrated distribution (i.e., more balanced tasks) than the hash join algorithm. Recall that we used a parallelism of 144 and the 144 tasks are dynamically allocated to the 48 CPU cores at runtime by Spark's scheduler, which uses sophisticated heuristics based on data locality for making scheduling decisions. This has the effect of "smoothing" things out since multiple small tasks can be allocated to the same CPU core. However, this dynamic load balancing incurs extra cost for data migration and increased overhead for the scheduler.

Since both the hash join algorithm and the sort join algorithm are randomized, we have also examined their stability over multiple repetitions. We repeated both algorithms multiple times with $\alpha = 1.1$ and IN = 1,000,000. For the sort join algorithm, a different sample is drawn each time. For the hash join algorithm, we use a different hash function $h(x) = ax + b$, where a and b are chosen randomly each time. Figure 10 shows the results of the running times of the algorithms on 80 repetitions. We see that both algorithms are quite stable. Note that due to system perturbation, even a completely deterministic algorithm will have some small variations from time to time.

Results of equi-join on uniform distribution. We note that the uniform distribution actually presents the best-case scenario for the vanilla Spark join, as the load is naturally balanced. Meanwhile, unless the number of distinct keys is much less than p, no key is heavy and the hypercube algorithm will not be needed. Therefore, the hash join algorithm essentially reduces to the vanilla Spark join. This can be observed from the results shown in Figures 11 and 12, where we vary the number of distinct keys and the input size, respectively. Since the hash join algorithm still needs to first collect data statistics, then it just realizes that all keys are light hitters. After that, it performs

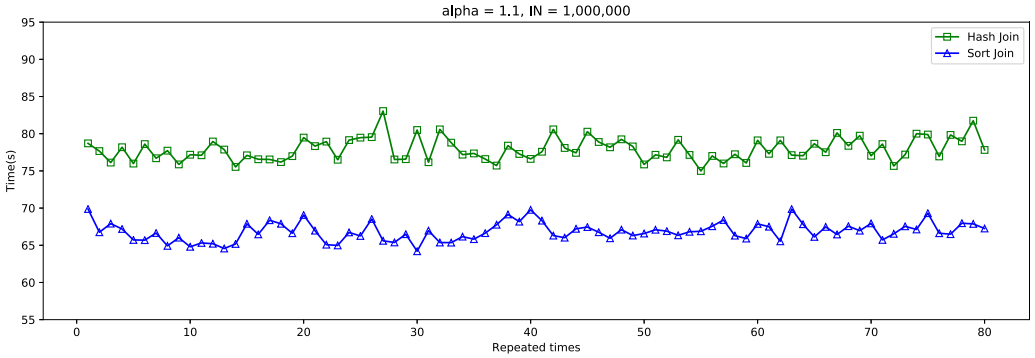

Fig. 10. Stability of the two output-optimal algorithms on the zipf distribution.

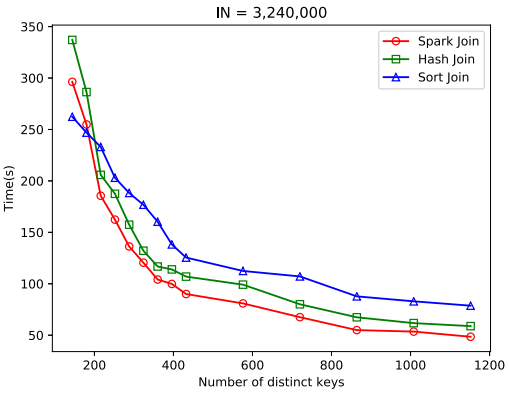

Fig. 11. Running time with different numbers of keys.

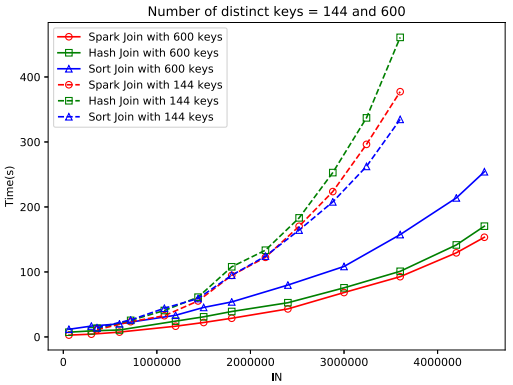

Fig. 12. Running time with different input size.

exactly the same shuffle operation as Spark join, so we always see a small gap, proportional to IN, between the two algorithms.

The comparison between the hash join and the sort join algorithm is more interesting. Note that the sort join algorithm achieves almost perfect load balance on the uniform distribution. Suppose that there are n distinct keys, each with frequency IN/n. Then the number of non-crossing keys between every two consecutive splitters is almost always $n/p - 1$, as long as the splitters do not "drift away" by more than a distance of IN/n. There are exactly $p - 1$ crossing keys. But since each key's frequency is not high enough, the "grid" allocated to each crossing key degenerates to a 1×1 grid. Therefore, it is almost always the case that exactly n/p distinct keys are assigned to each server.

However, the behavior of the hash join algorithm is characterized by the classical balls-in-bins problem [26], where we throw n balls into p bins randomly and study the number of balls landing in the largest bin. It is known that (1) if $n = \Theta(p)$, then with high probability the largest bin receives $\Theta(\log p / \log \log p)$ balls, and (2) when $n = \Omega(p \log p)$, then with high probability the largest bin receives $\Theta(n/p)$ balls. Thus, when n is large, we expect the hash join algorithm to perform well, whereas for small n, it can be sub-optimal by an $\Theta(\log p / \log \log p)$ factor. This is can be verified in Figure 12. When there are 144 distinct keys, each server in the sort join algorithm is allocated with exactly 1 key, wherease the largest server in the hash join gets approximately 3 keys. With the number of distinct keys increasing (e.g., 600 keys), the advantage of sort join will disappear since

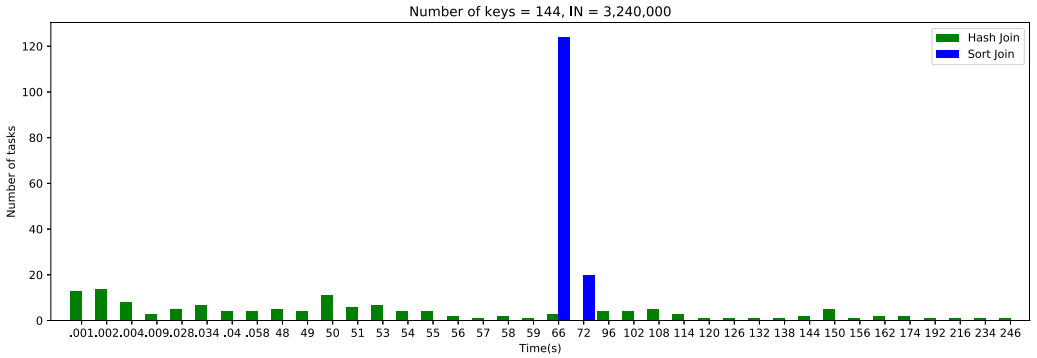

Fig. 13. Running time distribution of tasks in the last stage on the uniform distribution.

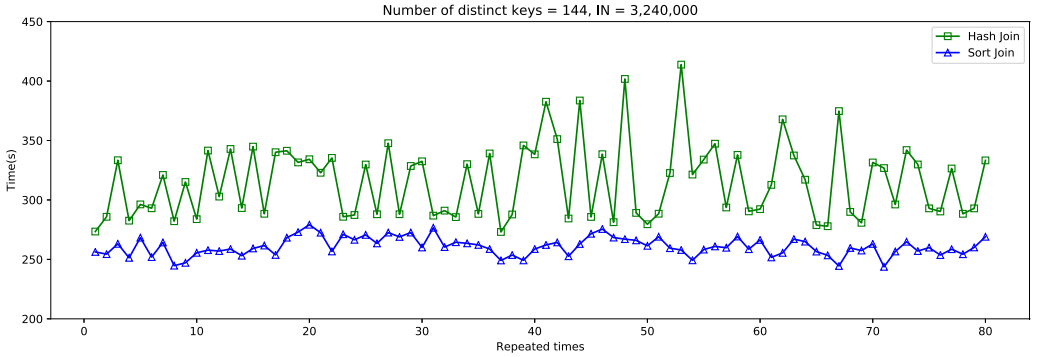

Fig. 14. Stability of two output-optimal algorithms on uniform distribution.

the hash join would achieve almost balanced load, whereas sort join always has extra overhead for server allocation. This can be further verified in Figure 13, where we plot the running time distribution of the 144 tasks of the two algorithms. Comparing Figures 13 and 9, we see that on the uniform dataset with 144 distinct keys, the tasks' running times are more unbalanced for the hash join algorithm, whereas the sort join algorithm produces tasks of almost equal running times.

As before, we also tested the stability of the two algorithms over repeated runs using different samples and random hash functions, using a uniform data set with 144 distinct keys. The results are shown in Figure 14. Compared to the results in Figure 10 on the zipf distribution, we see that the sort join algorithm is still very stable across different runs, but the hash join algorithm is much less stable. This is because on the zipf distribution, the largest load is determined by the heavy hitters, and a random allocation can deal with this case well because the hash function is applied on the other attributes of the tuples. However, on the uniform distribution, we hash on the join key and the maximum load is determined by the server receiving the most distinct join keys, which tends to be much less stable.

The scalability results of all algorithms are shown in Figure 15. As expected, all algorithms will benefit from more worker nodes because on the uniform distribution, load is always balanced.

Results of similarity joins in one dimension. We have performed two sets of experiments in which we vary the similarity threshold and the data size, respectively. In the first set of experiments, the number of points in each set is fixed at 100,000 and r varies from 0.1 to 0.5, with the

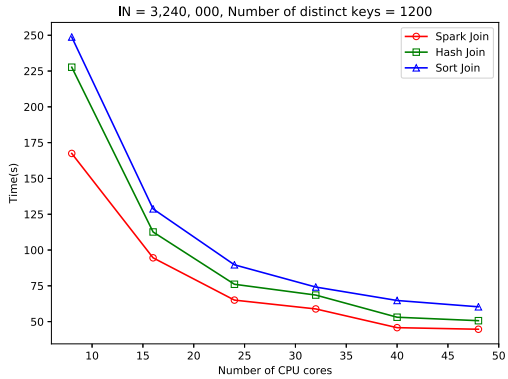

Fig. 15. Running times with different numbers of CPU cores on the uniform distribution.

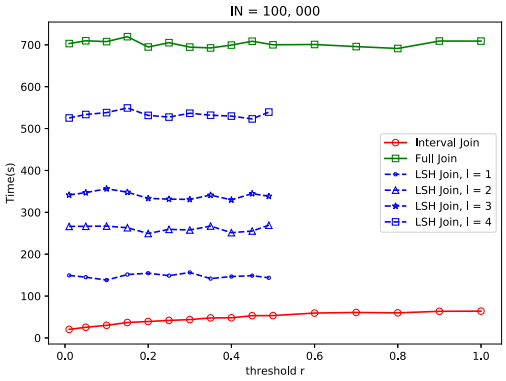

Fig. 16. Running times with different thresholds.

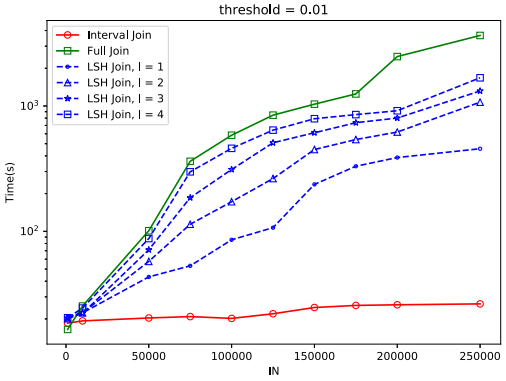

Fig. 17. Running times with different input sizes.

results shown in Figure 16. In the second set of experiments, we fix the similarity threshold at 0.01 and increase IN, and the results are shown in Figure 17.

From both sets of experiments, we see that, as expected, the full join has highest cost, whereas the interval join algorithm has the lowest cost. In both figures, the cost of the interval join grows, as its load has both an input-dependent term and an output-dependent term. Note that increasing r will increase the output size. The cost of the full join algorithm is not affected by r, as it always enumerates all the IN^2 pairs of tuples.

For the LSH join, we fixed the approximation ratio at 2 but varies l, the number of hash keys per hash function from 1 to 4. Note that when fixing l, the input size of the equi-join is also fixed, thus the running time is not affected by r. But a larger r implies larger output size, the ratio of true results that can be recalled would be decreased, as shown in Table 1. Note that a larger l implies larger input size of the equi-join and thus larger running times. Meanwhile, a larger l increases the accuracy of the hash functions, resulting in a higher recall rate, as shown in Table 2. We see that the LSH join is outperformed by interval join, despite being an approximation algorithm. Essentially, LSH join is designed for the high-dimensional case. For one-dimensional similarity joins, a specialized algorithm like interval join is much more competitive.

Results of similarity joins in high dimensions. In the last set of experiments, we perform self-similarity joins on the COREL dataset using Hamming distance. Given a distance threshold r, we first construct an $(r, 10r, 0.9, 0.1)$-sensitive hash family such that any pair of points with distance

Table 1. $k = 10, c = 2, l = 1$

r	0.01	0.1	0.2	0.3	0.4
p_1	0.9043	0.3487	0.1074	0.0282	0.0060
p_2	0.8171	0.1074	0.0060	0.0001	1e-7
ρ	0.4975	0.4722	0.4368	0.3893	0.3174
Recall	0.9496	0. 5863	0.3344	0.2548	0.2067

Table 2. $k = 10, c = 2, r = 0.01$

l	1	2	3	4
p_1	0.9043	0.9909	0.9991	0.9999
p_2	0.8171	0.9665	0.9939	0.9989
ρ	0.4975	0.2699	0.1424	0.0746
Recall	0.9496	0.9949	0.9968	0.9999

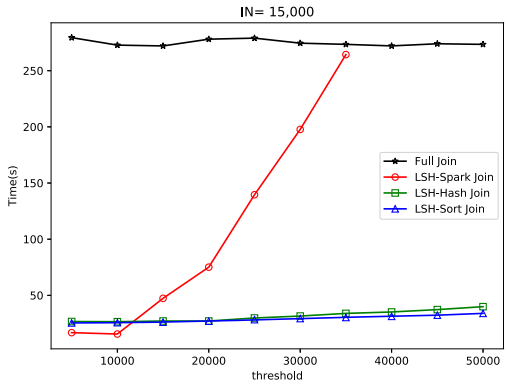

Fig. 18. Running time with different thresholds.

smaller than r has probability at least 0.9 to be hashed to the same bucket, whereas any pair of points with distance greater than $10r$ has probability no more than 0.1 to collide. In this set of experiments, we fix the number of candidate hash keys as 2 and vary the number bits per hash keys. The larger r is, the less bits the hash keys will have, and more points will be hashed to the same bucket. The join size will also increase as well.

Figure 18 shows the running times of the three algorithms over different similarity thresholds. The overall trend is similar to that on the zipf distribution (Figure 6), which is as expected, as the COREL dataset has an uneven distribution. After hashed into buckets by an LSH function, the distribution of the buckets is also highly skewed.

A small difference between Figures 18 and 6 is that when r is very small, the running times of all algorithms also increase. This in fact is due to the length of the join keys being large for small r, which makes the tuples larger, although the amount of work in terms of tuple count should still decrease.

10 CONCLUDING REMARKS

In this article, we have studied various similarity joins in the MPC model. The main difference between this and prior work is that we consider OUT, the output size of the join, as an additional parameter in characterizing the complexity of the algorithms. We first proposed a deterministic equi-join algorithm that improves the hypercube algorithm by a polylogarithmic factor and removes the assumption on knowing data statistics. Then we designed output-optimal algorithms for similarity joins under $\ell_1/\ell_2/\ell_\infty$ distances in constant dimensions. We also gave an approximation algorithm based on LSH for the high-dimensional case. Finally, we gave practical versions of some of these algorithms and performed an experimental evaluation.

This article is mainly concerned with 2-relation joins, but we also made an initial step towards multi-way joins by presenting a lower bound regarding output-optimality for the 3-relation chain

join. However, it remains an open problem to more precisely characterize the class of multi-way joins that can be solved with output-optimal MPC algorithms.

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. Although 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, 21], and so forth.

REFERENCES

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

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

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

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

[5] A. Andoni and P. Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM* 51, 1 (2008), 117.

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

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

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

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

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

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

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

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

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

[15] A. Gionis, P. Indyk, and R. Motwani. 1999. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases*.

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

[17] M. T. Goodrich, N. Sitchinava, and Q. Zhang. 2011. Sorting, searching and simulation in the MapReduce framework. In *Proceedings of the International Symposium on Algorithms and Computation*.

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

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

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

[21] M. Joglekar and C. Ré. 2016. It's all a matter of degree: Using degree information to optimize multiway joins. In *Proceedings of the International Conference on Database Theory*.

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

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

[24] P. Koutris and D. Suciu. 2011. Parallel evaluation of conjunctive queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*.

[25] Y. Li, P. M. Long, and A. Srinivasan. 2001. Improved bounds on the sample complexity of learning. *Journal of Computer and System Sciences* 62, 3 (2001), 516–527.
[26] M. Mitzenmacher and E. Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press.
[27] O. O'Malley. 2008. *Terabyte Sort on Apache Hadoop*. Technical Report. Yahoo!
[28] R. Pagh, N. Pham, F. Silvestri, and M. Stöckel. 2015. I/O-efficient similarity join. In *Proceedings of the European Symposium on Algorithms*.
[29] R. Pagh and F. Silvestri. 2014. The input/output complexity of triangle enumeration. In *Proceedings of the ACM Symposium on Principles of Database Systems*.
[30] M. Pătraşcu. 2011. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing* 40, 3 (2011), 827–847.
[31] Y. Tao, W. Lin, and X. Xiao. 2013. Minimal MapReduce algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
[32] L. G. Valiant. 1990. A bridging model for parallel computation. *Communications of the ACM* 33, 8 (1990), 103–111.
[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*.

Received December 2017; revised November 2018; accepted January 2019