Implementing Distributed Approximate Similarity Joins using Locality Sensitive Hashing

Martin Aumüller
IT University of Copenhagen
maau@itu.dk

Matteo Ceccarello*
Free University of Bozen-Bolzano
matteo.ceccarello@unibz.it

ABSTRACT

Similarity joins are a basic primitive in data mining. Given two sets of points, we are interested in reporting all pairs of points whose similarity is above a user-defined threshold. Solving the problem naively entails verifying all possible pairs, which can be infeasible for large inputs. In such contexts, Locality Sensitive Hashing (LSH) is often considered to reduce the number of pairs to verify. However, while it provides subquadratic running time, large input sets make it nevertheless necessary to resort to distributed computing. Hu, Yi, and Tao (PODS'17, TODS'19) proposed a nearly load-optimal LSH-based join algorithm and provided a small-scale experimental study in a distributed setting. This paper provides further analysis of their approach. It shows that the load-minimizing parameter settings by Hu et al. incur too much local work, rendering it impractical.

To remedy this drawback, we propose two approaches: The first distributes work in a data-independent way, while the second adapts to the data distribution using LSH. Both schemes then use LSH to solve subproblems locally. This allows to balance load and the amount of local work.

Through an experimental evaluation, we show that the transition from theory to practice for Hu et al.'s approach is challenging: it is hard to strike a good tradeoff between the load and the amount of local work of each processor, load balancing is itself an issue, and LSH may introduce duplicates in the output. Our extensive experimental evaluation is supported by an efficient open source implementation of all the methods we test.

Our results highlight the need for an holistic approach: only focusing on the load, as tradition in the MPC model, might not make efficient use the available resources and better trade-offs between local work and load are possible.

1 INTRODUCTION

Computing similarity joins is a basic primitive in many data mining tasks. In a similarity join, we are given two sets R and S of objects, typically represented as feature vectors in a high-dimensional space $\mathbb{R}^d$. For a given similarity measure, for example Cosine or Jaccard similarity, we are interested in reporting all pairs $(r, s) \in R \times S$ that are "similar" to each other. In a technical sense, a user provides a threshold value $r \in \mathbb{R}$ and the system is to report all pairs whose similarity is at least r. For example, Sharma et al. [26] carried out a similarity join on the `Twitter` follower graph containing 100s of billion of edges to retrieve similar users for each user in the dataset. Other applications are in recommender systems, for example on music or video streaming platforms. There, carrying out a similarity join retrieves items that should be recommended to a user based on the user's history.

A similarity join can be computed in time $O(|R| \cdot |S|)$ by comparing each element in R to each element in S, a running time prohibitive for large datasets with millions of items. Since similarity joins are a special case of nearest neighbor search, the curse of dimensionality applies also for them. This means that exact solutions to the problem are unlikely to improve much asymptotically on the quadratic time boundary, see [2]. For certain similarity measures, for example sets under Jaccard similarity, techniques like *prefix filtering* [10] can be used to improve the performance of the all-to-all comparison. However, this still does not suffice to provide scalable solutions if the sets are large.

To build efficient solutions to high-dimensional similarity search problems, research has focused on the design of *approximate solutions*. In our context, this means that there is no guarantee of the result being exact, and the quality of the solution can be measured in terms of the *recall* of the solution. In the case of a similarity join, this means that we report on the fraction of similar pairs that were reported by the algorithm compared to the exact solution. Among all techniques for similarity search problems, *locality-sensitive hashing* (LSH) as introduced by Indyk and Motwani in [16] stands out with strong theoretical guarantees paired with a general black-box approach suitable for many different metrics and similarity measures. LSH-based solutions usually provide a probabilistic guarantee on their quality, i.e., each close pair has a chance of at least $1 - \delta$ of being reported, where δ is a failure probability usually set by the user. As we will see later on, using LSH we can compute the similarity join of two sets containing n points in space and time $O(n^{1+\rho})$, where $\rho \leq 1$ is the quality of the LSH function. The parameter ρ depends on the similarity threshold provided by the user and reflects how difficult it is to separate the close pairs from pairs that are below the threshold. For large sets, the time and space requirements might be unrealistic for a single machine, so much of research focused on the design of distributed systems.

This paper focuses on solving similarity joins of high-dimensional data using LSH in a distributed setting. As a starting point, a recent VLDB survey by Fier et al. [14] reported on the scalability of existing, exact solutions for set similarity joins. Their results were rather discouraging: On a small cluster, many implementations failed to compute the join in a distributed setting using Apache `Hadoop`. This was often an easy task for a single-threaded C++ implementation, as shown in another VLDB survey by Mann et al. [18]. In terms of the COST measure introduced by McSherry et al. in [20], this means that exact solutions in `Hadoop` have a large COST, i.e., require a large cluster to start outperforming a single thread, if they are at all able to outperform such a solution.

Solving Similarity Joins via LSH. Locality-Sensitive Hashing (LSH) [16] is the de-facto standard to provide theoretically sound algorithms for the approximate near neighbor problem. Given a metric space $(\mathcal{X}, \mathcal{M})$ that admits an LSH, we can use an LSH function to hash a point $p \in \mathcal{X}$ to a hash code $\mathbb{R}^k$. The closer two points are, the more likely they are to land in the same bucket.

*Work done in part while at the IT University of Copenhagen.

Table 1: Comparison of performance between configurations minimizing the load (*theoretical*) and the running time (*practical*), for our implementation of the algorithm described in [15]. Times are in minutes, the load is the maximum number of messages received by any machine.

		time (min)	load
Glove	theoretical	107.2	537 104
	practical	52.3	1 117 862
SIFT	theoretical	31.5	290 491
	practical	21.8	938 691
Livejournal	theoretical	45.6	503 309
	practical	14.2	822 931
Orkut	theoretical	33.6	391 267
	practical	3.6	1 601 311

To solve an LSH-based similarity join of two sets containing at most n data points each, one hashes the points using $L = n^\rho$ hash functions $h_1, \ldots, h_L$, where ρ depends on the similarity threshold. If two points have similarity above a threshold $r \in \mathbb{R}$, then with constant probability there exists a hash function under which these points collide. As such, we can just carry out an all-to-all comparison among all colliding vectors and report those that are above the similarity threshold. More details will be given in Section 2.

Hu et al. [15] describe a similarity join algorithm using LSH for distributed similarity joins and give parameter choices that minimize the expected load during the computation. The idea is that each worker computes locally the L hash values for each of its vectors, and then a distributed join is carried out using as keys the hash values paired with the index of the corresponding hash function. Then, each worker solves locally the similarity join using an all-to-all comparison on colliding pairs. The purpose of the present paper is to provide further analyses and experiments on Hu et al.'s approach. We believe this is necessary because their paper focused on the theoretical foundations of designing a load-optimal similarity join algorithm. When going into practice, their focus on the MPC model (Section 2.2), which uses network load as cost metric but hides the cost of local computation, might not provide best parameter choices. Moreover, they do not discuss how to solve problems that arise from data duplication: In particular, how does a worker know if a pair of vectors has already been counted/emitted on a different worker due to collisions under multiple hash functions? We will make such challenges of LSH-based solutions explicit and provide efficient solutions.

A motivating example. We reimplemented the aforementioned algorithm from [15] in the *Timely Dataflow* framework. We used this implementation to compute similarity self-joins on some standard data sets known from surveys on nearest neighbor search [4] and set similarity search [14], at similarity threshold 0.7. Table 1 reports, for each dataset, the configuration minimizing the load (the cost metric in the MPC model) and the one minimizing the running time: the former are up to ten times as slow as the latter.

As we will discuss in detail in Section 2.4, the main reason for this behavior is that local computation in the context of similarity search is not free, and might dominate the running time. In more detail, in [15], the sets R and S are distributed using LSH (putting similar elements onto the same worker, but retaining a constant fraction of far pairs as well); each server receiving subsets R' and S' will compute the similarity join of R' and S' using the naïve all-to-all comparison approach that runs in time $O(|R'||S'|)$. As shown in Hu et al. [15], to achieve minimal load in expectation, the LSH parameter only depends on the number of servers p and the "quality of the LSH" in terms of its ρ-value. This means that R' and S' are still large, and the cost of local computation will dominate the cost of the join. A particular problem with LSH is that the subproblems defined by applying a single hash function do not have good load balancing properties, e.g., they differ widely in size. This has to be taken into account when computing the join on the hash values.

Our contribution. With this paper, we bridge the gap between the theoretical considerations by Hu et al. in [15] and an efficient distributed similarity join algorithm based on LSH in practice. We outline the design space of LSH-based algorithms and include (i) *sketching techniques* to speed up candidate verification, (ii) *hash function tensoring* (i.e., reusing hash values) to decrease the number of LSH function evaluations, and (iii) propose an *efficient, local duplicate check* that works in a distributed setting. One main contribution of the paper is an efficient, low-overhead implementation of LSH-based similarity join in a distributed setting as open source.

Motivated by the shortcomings of disregarding the influence of local computation, we will explore the design space of LSH-based similarity join algorithms in a distributed setting. As described above, Hu et al.'s algorithm distributes data in data-dependent way to workers using LSH, but then uses all-to-all comparisons to solve the local subproblems. Splitting up the abstract view of a similarity join into generating subproblems and solving them locally gives rise to the following other variants. We could generate subproblems naïvely (similar to computing a cartesian product) and solve subproblems locally using LSH, or use LSH on both levels, both for data distribution and computing solutions for the generated subproblems. A detailed overview over these algorithms and their theoretical guarantees in terms of expected load and their cost of local computation is described in Section 3. In that section, we will also discuss how we can use techniques described in [11] to check for duplicated pairs in the output of the join, and we will use sketching techniques from the very same paper to speed up candidate checking. In Section 4, we report on experimental results of the algorithms discussed.

In a nutshell, from a theoretical side using LSH to both generate subproblems and solve them merges the best of both worlds: low expected load, and small local work. Empirically, we will see that for the small cluster employed in our study, the most efficient solution is often provided by creating subproblems data-independently, and then using LSH locally to compute the similarity join. However, data-dependent subproblem generation using LSH provides smaller load and can be favorable in situations where this is a concern. Adding sketches to the computation is of tremendous importance: not only does it speed up the verification process of a pair, but it also makes the LSH application much more robust with respect to its internal parameters. Lastly, Section 3.4 shows that the tensoring approach from [11] can be used for an efficient duplication check to avoid a final round of communication to deduplicate the pairs. In essence, it is not slower than verifying the exact similarity.

Related Work. Fier et al. survey exact approaches to set similarity joins in MapReduce in [14]. An overview over such approaches in a non-distributed setting is given by Mann et al. in [18]. Christiani et al. describe in [13] an approximate set similarity join algorithm in the sequential setting based on the algorithm for set similarity search developed by Christiani and Pagh in [12]. They compare their approach to some other LSH-based approaches and we refer the reader to [13] for the details. However, none of these approximate, centralized techniques are straight-forward to translate into a distributed setting.

With regard to approximate, distributed similarity join under Cosine Similarity, Sharma et al. present in [26] a distributed system using a sampling approach called wedge sampling to generate candidate vectors. They use SimHash-based sketches [9] for quick estimation of the Cosine Similarity. In very recent work, Rashtchian et al. describe in [24] another algorithm in this scenario that makes use of a locality-sensitive filtering step to generate subproblems on which to perform the naive join. While it would be interesting to provide a detailed comparison between LSF and LSH joins, we stress that this paper focuses on providing further experiments and analyses on the black-box LSH-based join by Hu et al. [15]. It is future work to compare these different approaches to similarity joins to each other, and we hope that this paper provides a solid starting ground for LSH-based implementations.

2 PRELIMINARIES

2.1 Problem definition

Consider a metric space $(\mathcal{X}, \text{dist})$, and let $r > 0$ be an input parameter. The *similarity join* of radius r of two sets $R, S \subseteq \mathcal{X}$ is defined as the set

$$R \bowtie_{\leq r} S = \{(x, y) \in R \times S : \text{dist}(x, y) \leq r\}$$

For notational simplicity, we define $\text{OUT} = |R \bowtie_{\leq r} S|$. We remark that this distance-based join is usually referred to as a *distance range join* [27]. Since the distance measure in a metric space is often a measure of dissimilarity, we will use the more general term *similarity join* instead. Additionally, for simplicity we will only discuss *self-joins* of the form $R \bowtie_{\leq r} R$ in this paper. The general case $R \bowtie_{\leq r} S$ can be solved by merging both sets into a set R', carrying out the self-join, and filtering output pairs. As tradition, we will denote with IN the size of R.

In this paper we present approximate randomized algorithms for this problem. The quality of the returned solution is measured in terms of *recall*, defined as the fraction of pairs in $R \bowtie_{\leq r} R$ reported by the algorithm. Note that all the algorithms we describe verify their output pairs against the similarity threshold, therefore there are no false positives, i.e., the *precision* is always 1.

2.2 Model of Computation

We study our algorithms in the *massively parallel computation* model (MPC), which is a standard model to study distributed join algorithms, see for example the references in [15]. In this model, p servers are connected with a complete network. The input data is arbitrarily partitioned across the p servers in the beginning. In each round, each server receives messages from the other servers, does some local computation, and then sends messages to other servers. These messages will be received by these servers in the start of the next round. The complexity of an algorithm in this model is described by the number of rounds it performs, and by the load, which is the maximum message size received by any server in any round. For simplicity, we state results on the load in terms of the number of messages received (instead of the number of bits). Furthermore, we state all of our results assuming that workers have enough memory to (i) store all received messages and (ii) carry out the local similarity join computation[1].

The MPC model disregards the cost of local computation. To highlight differences between algorithms, we will take the cost of local computation into account, which we will measure by the maximum number of local operations carried out in any round of the computation by all workers. In general, we will use the number of similarity computations carried out between pairs of points as a proxy for the local work.

2.3 Locality Sensitive Hashing

Definition 2.1 (Locality Sensitive Hashing [16]). Let $(\mathcal{X}, \text{dist})$ be a metric space, let T be a set, and let $\mathcal{H}$ be a family of functions $h: \mathcal{X} \to T$. For positive reals r_1, r_2, q_1, q_2, with $q_1 > q_2$, $\mathcal{H}$ is (r_1, r_2, q_1, q_2)-sensitive if for $x, y \in \mathcal{X}$ and h sampled uniformly at random from $\mathcal{H}$ we have that:

- $\text{dist}(x, y) \leq r_1 \Rightarrow \Pr[h(x) = h(y)] \geq q_1$
- $\text{dist}(x, y) \geq r_2 \Rightarrow \Pr[h(x) = h(y)] \leq q_2$

In this paper we focus mainly on families of functions where $T = \{0, 1\}$. For the d-dimensional unit sphere under inner product similarity (or cosine similarity), such a family is random hyperplane hashing from [9]. As another example, 1-bit MinHash described by Li and König in [17] is such a family for set similarity under Jaccard similarity. As a technical detail, we assume that the LSH is *monotonic*, i.e., its collision probability function is decreasing with the distance. Many LSH families have this property, see the discussion in [15] as well. Since we use LSH functions as a black-box, our results hold for all LSH families that have this property.

Given the set R and two distance thresholds $r_1 < r_2$, for notational simplicity we denote as *near pairs* the pairs of points in OUT. The pairs of points in $R \times R$ at distance greater than r_1 but less than r_2 are called r_2*-near* pairs; their number is denoted by OUT_{r_2}. Pairs of points further away than r_2 are called *far*.

A similarity self-join of the set R with distance threshold r_1 can be approached as follows: First, select a hash function $h \in \mathcal{H}$ uniformly at random and compute each hash value of points in R using h. All pairs $(x, y) \in R \times R$ with $h(x) = h(y)$ form the candidate set C, which can be efficiently constructed using a hash map. Using exact distance computations on all pairs in C, report only those pairs of points at distance at most r_1.

This approach has several downsides that need to be addressed: For a fixed collision probability q_2, there might be as many as $q_2|R|^2$ pairs of far points colliding under the hash function h in expectation. The standard solution to make this number smaller is to create a new LSH family $\mathcal{H}^k$ obtained by concatenating $k \geq 1$ hash functions from $\mathcal{H}$. The new LSH family is then (r_1, r_2, q_1^k, q_2^k)-sensitive [16]. While this allows us to control for the number of far pairs, a near pair might only be found with probability as low as q_1^k. This means that we have *to repeat* the construction with independent random choices from $\mathcal{H}^k$ at least $L = 1/q_1^k$ times to obtain constant probability bounds for reporting a near pair. On the other hand, in expectation, for a monotonic

[1] If either of the two conditions does not hold, the computation can be split into multiple rounds. However, we found that tuning the algorithm's parameters so to meet conditions (i) and (ii) gives better performance.

LSH each r_2-near point is inspected at most $q_1^k \cdot 1/q_1^k = 1$ time over all repetitions, and not more than $O\left((q_2/q_1)^k\right)$ far neighbors collide over all repetitions.

2.4 Similarity Join by Hu et al.

In [15], Hu et al. described a straight-forward LSH-based similarity join algorithm for high-dimensional data. We will refer to their algorithm as OneLevelLSH. It works as follows for a self-join on R and a distance threshold $r_1 > 0$ in a setting using p servers. To set up parameters for the LSH function, select an approximation factor $c > 1$ such that pairs of points at distance at least $r_2 = cr_1$ are considered far. (This is up to the user.) From that, the ρ value of the LSH is obtained as the ratio of the logarithms of the collision probabilities q_1 and q_2 of the LSH function at distance r_1 and r_2, respectively. Next, choose k as the smallest integer such that $q_1^k \leq 1/p^{\frac{\rho}{1+\rho}}$, which ends the parameter estimation phase. To compute the similarity join, choose $1/q_1^k$ hash functions at random from $\mathcal{H}^k$ and distribute them to the servers. For each input point, create $1/q_1^k$ copies applying each hash function once. As is standard in the LSH literature, we call a copy a *repetition*. Finally, perform an equi-join on the pairs that collide under the same hash value and report the points that are at distance at most r_1 from each other by doing a naive all-to-all comparison among the points having the same hash value. We summarize the performance of this approach in the following lemma. The load discussion was provided in [15, Theorem 6.1]. We augment their result considering hash function evaluations and distance computations.

Lemma 2.2. *In expectation, OneLevelLSH evaluates*

$$O\left(k \cdot p^{1/\rho} \cdot \mathrm{IN}\right)$$

hash functions, has load

$$O\left(\sqrt{\frac{\mathrm{OUT}}{p^{1/(1+\rho)}}} + \sqrt{\frac{\mathrm{OUT}_{r_2}}{p}} + \frac{\mathrm{IN}}{p^{1/(1+\rho)}}\right),$$

and carries out $O\left(\mathrm{OUT} \cdot p^{\frac{\rho}{1+\rho}} + \mathrm{OUT}_{r_2} + \mathrm{IN}^2 \cdot p^{\frac{\rho-1}{\rho+1}}\right)$ *distance computations over the whole computation. Each pair in* OUT *has constant probability of being reported.*

For convenience, we provide a sketch of the proof.

Proof. By linearity of expectation, we may consider pairs of near points, r_2-near points, and far points independently. First, each pair of near points could collide under each of the $1/q_1^k$ copies of the point, so we expect $O(\mathrm{OUT}/q_1^k)$ such collisions. By the properties of a monotonic LSH, each pair of r_2-near points is expected to collide $O(1)$ times over all $1/q_1^k$ repetitions. Finally, each pair of far points has a chance of q_2^k of colliding. This means that we expect to see $O((q_2/q_1)^k \mathrm{IN}^2)$ such pairs. This is the output size of the natural join on the hash, and the equi-join algorithm described in [15, Section 3.1] has expected load $O(\sqrt{\mathrm{OUT}/p} + \mathrm{IN}/p)$. As shown in [15], the choice $q_1^k \leq 1/p^{\rho/(1+\rho)}$ minimizes this load. Plugging in this value provides both the number of required distance computations and the expected load stated in the lemma. □

We note that there is a huge cost involved in the parameter setting that minimizes the load: The number of generated tuples of pairs that are far away might be only a constant fraction away from IN^2, i.e., the cost of the naïve brute-force solution. Moreover, it is not clear how to handle duplicates, which in a naive solution might need another round of distributed computation to be removed.

2.5 Fewer Hash Function Evaluations via Tensoring

The approach discussed above means that each dataset vector incurs $O(k \cdot L) = O\left(k \cdot (1/q_1)^k\right)$ hash function evaluations, which can be potentially expensive. To reduce this cost, Christiani [11] proposed a technique, named *tensoring*, to reduce the number of evaluated hash functions, based on the work of Andoni and Indyk [3]. We will describe a simplified construction that is a special case of [11].

Assume we are given an (r_1, r_2, q_1, q_2)-sensitive hash family $\mathcal{H}$ and an integer $k \geq 1$. Split up k into $k_\ell = \lceil k/2 \rceil$ and $k_r = \lfloor k/2 \rfloor$ such that $k_\ell + k_r = k$. For an integer $m \geq 1$, let $\mathcal{H}_\ell$ be a collection of m hash functions sampled from $\mathcal{H}^{k_\ell}$, and let $\mathcal{H}_r$ be a collection of m hash functions sampled from $\mathcal{H}^{k_r}$. The collection $(h_a, h_b) \in \mathcal{H}_\ell \times \mathcal{H}_r$, $1 \leq a, b \leq m$ provides m^2 (interdependent) repetitions, with only km hash function evaluations.

To guarantee that two near points collide with probability at least $1 - \delta$, we observe that a collision happens if there is an $h \in \mathcal{H}_\ell$ and an $h' \in \mathcal{H}_r$ such that the points collide under both h and h'. Since these hash functions are drawn independently, this happens with probability at least

$$(1 - (1 - q_1^{k_\ell})^m)(1 - (1 - q_1^{k_r})^m), \tag{1}$$

and we may pick the smallest m such that (1) is at least $1 - \delta$. A straight-forward manipulation of (1) shows that m is bounded from below by $\Omega\left(\sqrt{\ln(1/\delta)/q_1^k}\right)$ to achieve recall at least $1 - \delta$ for constant q_1.

In Section 3.4 we show that the tensoring data structure for computing hash values does not only provide an efficient way of computing hash values, but its structure can also be used to efficiently check for duplicates.

2.6 Sketching

Christiani [11] describes the following general way to create 1-bit sketches for points $x \in \mathcal{X}$. First, choose a random hash function h_{LSH} from an (r_1, r_2, q_1, q_2)-sensitive family $\mathcal{H}$ with range R. Next, choose a universal hash function $h \colon R \longrightarrow \{0, 1\}$ [8]. The sketch for $x \in \mathcal{X}$ is $s(x) = h(h_{\mathrm{LSH}}(x))$. To create b-bit sketches, we repeat the process above b times with independent functions. Note that for $R = \{0, 1\}$, we do not need to apply h.

As Christiani observes in [11], Hoeffding's inequality gives rise to a threshold λb such that pairs of points at distance at most r_1 collide at least λb times with good probability, while far points are unlikely to collide.

Lemma 2.3 ([11]). *Let $\mathcal{H}$ be a (r_1, r_2, q_1, q_2)-sensitive family and let $\lambda = (1 + q_2)/2 + (q_1 - q_2)/4$. Then for sketches of length $b \geq 1$ and for every pair of points $x, y \in \mathcal{X}$:*

- *If $dist(x, y) \leq r_1$ then*

$$\Pr[\|s(x) - s(y)\|_1 \leq \lambda b] \leq e^{-b(q_1-q_2)^2/8}.$$

- *If $dist(x, y) \geq r_2$ then*

$$\Pr[\|s(x) - s(y)\|_1 > \lambda b] \leq e^{-b(q_1-q_2)^2/8}.$$

As we can see, the larger our sketch length b, the better it is at differentiating between close and far points.

Since the number of colliding bits of two sketches of points x and y at distance r is binomially distributed with parameters b and $p = f(r)$, we can also derive these thresholds exactly from the CDF of the Binomial distribution.

It remains to discuss which sketch functions to use. For inner product similarity on the unit sphere, using random hyperplane LSH [9] provides collision probability $q = 1 - \frac{\arccos(t)}{\pi}$ for two points x and y with inner product t. For Jaccard similarity and two points with similarity t, using one-bit MinHash [17] the probability of two bits of the sketches being the same is simply $(1+t)/2$. We can think of one-bit MinHash as Christiani's general sketching function with h_{LSH} being Broder's MinHash [7].

We remark that ad-hoc sketching techniques have been used in other approaches as well, e.g., in [6, 25, 26].

3 ALGORITHMS

Starting from the similarity join algorithm by Hu et al. [15], we categorize distributed similarity join algorithms into four natural categories, depending where and how they use LSH, as detailed in Table 2. First, we can choose whether to distribute points in a data-dependent way (using LSH) or in a data-independent way. In the former case, subproblems are defined by points colliding on the same hash value in some repetition (as in OneLevelLSH). In the latter case, we define subproblems by partitioning the cartesian product of the input. Once subproblems have been defined, we can choose how to solve them locally in each worker: either we evaluate all possible pairs of points, or we use LSH. We therefore have the following four cases:

- *Data independent partitioning, all to all local computation.* This is equivalent to computing the cartesian product of the input, and filtering the resulting pairs depending on their similarity. This is the simplest approach. It has load $\Theta(\text{IN}/\sqrt{p})$ and computes $\Theta(\text{IN}^2)$ distances. We refer to it as Cartesian.
- *Data dependent partitioning (with LSH), all to all local computation.* This is the approach OneLevelLSH by Hu et al. [15] with the properties stated in Lemma 2.2.
- *Data independent partitioning, LSH-based local computation.* This is LocalLSH described in Section 3.1.
- *Data dependent partitioning (with LSH), LSH-based local computation.* This is TwoLevelLSH described in Section 3.2.

We remark that while this categorization is natural, it was not considered by Hu et al. because local computation is considered free in the MPC model.

In the following, we will discuss the two aforementioned algorithms. For simplicity, we assume that the wished recall guarantee $1 - \delta$ is a constant. This allows us to hide the additional factor of $\log(1/\delta)$ in the repetition count in the big-Oh notation. Furthermore, we will discuss the practical role of sketching separately in Section 3.3.

3.1 LocalLSH algorithm

For a given set R, a distance threshold r_1, and a recall guarantee of $1 - \delta$, the algorithm works as follows. First, the user picks a distance threshold $r_2 > r_1$, in the same spirit as the choice of c in Section 2.4. Consider an (r_1, r_2, q_1, q_2)-sensitive family $\mathcal{H}$. Let $\rho = \frac{\log 1/q_1}{\log 1/q_2}$, and let k be a parameter to be set later.

(1) Sample km hash functions as described in Section 2.5 and sample a sketch function σ of length b (Section 2.6). Distribute the hash functions and the sketch function to all workers.
(2) Compute $\sigma(x)$ for each point in R. Distribute R (associated with their sketches) to workers in the same way as for computing the cartesian product. Each worker receives a subset R' and R'' of size $(|R'| + |R''|)/\sqrt{p}$.
(3) Each worker computes $R' \bowtie_{\leq r} R''$ using the basic LSH approach with tensoring, see Section 2. For each pair of points (x, y) colliding on the same hash value, if the sketches $\sigma(x)$, $\sigma(y)$ differ in more than λb bits (see Section 2.6), the pair is discarded. Otherwise, the algorithm checks whether the pair has already been seen using the duplication check strategy described in Section 3.4. If this is not the case, the actual distance between the two points is computed, and reported if it is at most r_1.

Theorem 3.1. *In expectation, LocalLSH computes the similarity self-join of a set R with load $O\left(\text{IN}/\sqrt{p}\right)$, evaluates $O\left(\text{IN}^{1+\rho/2}\right)$ hash functions, and carries out*

$$O\left(\text{IN}^{\rho} \cdot \text{OUT} + \text{OUT}_{r_2} + \text{IN}^{1+\rho}\right)$$

distance computations.

Proof. Let $(x, y) \in R \times R$ be a pair of close points. There exists exactly one worker that receives both x and y. This worker carries out the local LSH approach using an m value described in Section 2.5 such that x and y are guaranteed to collide with probability at least $1 - \delta'$, for an $\delta' < \delta$. Choose the sketch length according to Lemma 2.3 such that x and y pass the threshold with probability p' such that $p' \cdot (1 - \delta') \geq (1 - \delta)$. Any such choice of p' and δ' that satisfies the inequality guarantees that x and y will be reported with probability at least $1 - \delta$, which completes the correctness argument.

Distributing the sets incurs a load of $O(\text{IN}/\sqrt{p})$. The distribution of the hash functions to all worker has load km, which is $O(\text{IN}^{\rho/2})$. The number of hash function evaluations follows by the properties of the tensoring approach that is carried out for each point in R. We now focus on the local computation.

As in the proof of Lemma 2.2, for every choice of k we expect to see a pair of close points $O(1/q_1^k)$ times, we expect to see an r_2-near pair $O(1)$ times, and we expect to see not more than $O(\text{IN}^2 \cdot q_2^k/q_1^k)$ pairs of far away points. Thus, the number of distance computations can be bounded from above by $O\left(\text{OUT}/q_1^k + \text{OUT}_{r_2} + \text{IN}^2(q_2/q_1)^k\right)$. By setting k as the smallest value that satisfies $q_2^k \leq 1/\text{IN}$ (and using that $q_1 = q_2^{\rho}$), we obtain the bound claimed in the theorem. □

Clearly, the load of this algorithm is not output optimal. However, we notice that for the regime $\text{OUT} = O(\text{IN}^{2-\rho})$, the algorithm does fewer distance computations than OneLevelLSH, cf. Lemma 2.2.

3.2 TwoLevelLSH algorithm

In this section we use LSH to both define subproblems being assigned to workers and to solve these subproblems. In a nutshell, we run OneLevelLSH to distribute data points to workers, and then apply LocalLSH to compute the similarity join for the data points received by a worker. It will turn out that this approach enjoys both the load bound from OneLevelLSH as well as the improvements in terms of local computation from LocalLSH.

Table 2: Taxonomy of similarity join algorithm depending on their usage of Locality Sensitive Hashing.

		Solution of subproblems	
		All to all	LSH-based
Data partitioning	Data independent	Cartesian	LocalLSH [this paper]
	Data dependent (LSH)	OneLevelLSH [15]	TwoLevelLSH [this paper]

Consider an (r_1, r_2, q_1, q_2)-sensitive family $\mathcal{H}$ and let k_{inner} and k_{outer} be integers to be set below. Let $\rho = \frac{\log 1/q_1}{\log 1/q_2}$, and let $1 - \delta$ be the recall guarantee.

(1) Choose δ' such that $1-\delta' = \sqrt{1-\delta}$. Next, sample $k_{\text{inner}} m_{\text{inner}}$ and $k_{\text{outer}} m_{\text{outer}}$ hash functions from $\mathcal{H}$ according to Section 2.5 for failure probability δ'. Sample a b-bit sketch function σ. Distribute these functions to the workers.
(2) Each worker having data point x produces m^2_{outer} copies $(i, h_i(x), x, \sigma(x))$ with $1 \le i \le m^2_{\text{outer}}$. It uses the equi-join algorithm on $(i, h_i(x))$ to distribute these tuples.
(3) Each worker receiving $(i, h_i(x), x, \sigma(x))$ forms the sets R'_i and R''_i for $1 \le i \le m^2_{\text{outer}}$. For each $i \in \{1, \dots, m^2_{\text{outer}}\}$, carry out step 3 of LocalLSH to compute $R'_i \bowtie_{\le r_1} R''_i$ with parameter k_{inner}.

Theorem 3.2. *In expectation, TwoLevelLSH computes the similarity self-join of a set R with load*

$$O\left(\sqrt{\frac{\text{OUT}}{p^{1/(1+\rho)}}} + \sqrt{\frac{\text{OUT}_{r_2}}{p}} + \frac{\text{IN}}{p^{1/(1+\rho)}}\right),$$

evaluates $O\left(\text{IN}^{1+\rho/2}\right)$ *hash functions, and computes not more than* $O\left(\text{IN}^{\rho} \cdot \text{OUT} + \text{OUT}_{r_2} + \text{IN}^{1+\rho}\right)$ *distances.*

Proof. We first discuss the correctness. Fix a pair of points $(x, y) \in R \times R$ at distance at most r_1. A necessary condition to report (x, y) is that the pair collides under at least one of the outer functions, i.e., it is part of one subproblem. Next, it has to collide in step 3 as well. Since the hash functions are independently chosen, the algorithm succeeds with probability $(1-\delta')^2 = \sqrt{1-\delta}^2 = 1-\delta$.

By setting k_{outer} such that $q_1^{k_{\text{outer}}} = (1/p)^{\rho/(\rho+1)}$, we get the same expected load as OneLevelLSH, cf. the proof of Lemma 2.2. The number of hash function evaluations follows from using the tensoring approach as discussed in Section 2.5. It remains to bound the local computation.

From Lemma 2.2 we expect

$$O\left(\text{OUT} \cdot p^{\frac{\rho}{1+\rho}} + \text{OUT}_{r_2} + \text{IN}^2 \cdot p^{\frac{\rho-1}{\rho+1}}\right)$$

tuples $(x, y) \in R \times R$ over all workers. Each near pair might collide $O(1/q_1^{k_{\text{inner}}})$ times. Thus, we expect not more than

$$O(\text{OUT}/q_1^{k_{\text{inner}}} p^{\frac{\rho}{1+\rho}})$$

such pairs. For r_2-near pairs, we expect at most $O(1)$ collisions. Finally, we expect $O\left((q_2/q_1)^{k_{\text{inner}}} \text{IN}^2 p^{\frac{\rho-1}{\rho+1}}\right)$ collisions of far pairs.

We set parameters such that $(1/q_1)^{k_{\text{inner}}} p^{\frac{\rho}{1+\rho}} \le \text{IN}^{\rho}$, which results in the choice $k_{\text{inner}} = \frac{\log(\text{IN}^{\rho}/p^{\frac{\rho}{1+\rho}})}{\log(1/q_1)}$. Using the identity $q_1 = q_2^{\rho}$ and plugging the choice of k_{inner} into the equation above gives a bound of $O(\text{IN}^{1+\rho})$ on the number of far away pairs. □

As stated before, TwoLevelLSH combines the best of both worlds: it has load asymptotically similar to OneLevelLSH and performs local work asymptotically as well as LocalLSH. However, these asymptotics hide one factor that might become important for practical performance: Instead of a factor of $\ln(1/\delta)$ more repetitions for success probability $1-\delta$, the success probability on each level has to be as large as $\sqrt{1-\delta}$. This means that we carry out a factor of $\ln^2(1/(1-\sqrt{1-\delta}))$ more repetitions. If we wish for recall at least 80%, for example, this translates into roughly 3 times more repetitions compared to LocalLSH or OneLevelLSH.

3.3 The role of sketching

From Theorem 3.1 and Theorem 3.2 we know that the algorithms, in expectation, carry out $O(\text{IN}^{\rho}\text{OUT} + \text{OUT}_{r_2} + \text{IN}^{1+\rho})$ distance computations, which can easily dominate the running time. Using the sketch approach discussed in Section 2.6, we can replace these distance computations by a first filtering step (using sketches), followed by the actual distance computation.

Lemma 3.3. *Using sketches of length* $\Theta(\ln \text{IN}/(q_1-q_2)^2)$, *OneLevelLSH and TwoLevelLSH (Theorem 3.1 and Theorem 3.2) carry out* $O(\text{IN}^{\rho}\text{OUT} + \text{OUT}_{r_2} + \text{IN}^{1+\rho})$ *filter computations using sketches and* $O(\text{IN}^{\rho}\text{OUT} + \text{OUT}_{r_2} + \text{IN}^{\rho})$ *distance computations, in expectation.*

Proof. From Lemma 2.3, we may set a sketch bitlength of $b = \Theta(\ln \text{IN}/(q_1 - q_2)^2)$ to obtain sketches (and a threshold λb) such that far pairs will be filtered out with probability $1 - 1/\text{IN}$, and close pairs survive with probability at least $1 - 1/\text{IN}$. This means that, in expectation, the algorithm carries out $O(\text{IN}^{\rho}\text{OUT} + \text{OUT}_{r_2} + \text{IN}^{1+\rho})$ filter computations using sketches, but only $O(\text{IN}^{\rho}\text{OUT} + \text{OUT}_{r_2} + \text{IN}^{\rho})$ actual distance computations. □

In practice, one can think about choosing a different threshold r' with $r' < r_2$ such that the sketching approach takes care of removing r_2-near pairs that are not r'-near. This increases the sketch length by a factor $(\frac{q_1-q_2}{q_1-q'})^2$ and gives an interesting tradeoff between the internal parameters of the LSH and the sketch parameters. We will explore these options in the experimental evaluation in Section 4.

3.4 Duplicate removal

Replicating a point multiple times introduces the additional problem of duplicate detection. The main challenge lies in the problem that a given pair of near points may be allocated to different workers in different repetitions. The most straightforward way to remove duplicates in this setting would be to perform an additional round of communication. However, the load of this round could be as high as $\tilde{O}(OUT)$, which is clearly too high.[2]

[2]There can be $O(p)$ copies of the same output pair, one in each worker (assuming that local duplicates are removed), and output pairs can be hash-partitioned in $\tilde{O}(OUT/p)$ groups, each assigned to a different worker.

The following lemma says that as long as the workers have access to the (small) collection of tensor hash functions from Section 2.5, they can check for duplicates in isolation.

Lemma 3.4. *Let $\mathcal{H}_\ell$ and $\mathcal{H}_r$ be the collection of $2m$ tensor hash functions. Order all m^2 repetitions in an arbitrary way. If a pair (x, y) collides in repetition j, then by evaluating $O(m)$ hash functions we can decide if there exists a repetition $j' < j$ in which (x, y) collides as well.*

Proof. Let j with $0 \le j < m^2$ be a repetition under which a pair (x, y) collides. Split up $j = a \cdot m + b$ such that the hash function used in the j-th repetition is $(h_a, h_b) \in \mathcal{H}_\ell \times \mathcal{H}_r$. To check if there exists a repetition $j' < j$ in which (x, y) collide as well, we compute all $h_{a'}(x), h_{a'}(y)$ for $h_{a'} \in \mathcal{H}_\ell$ with $0 \le a' < a$ and $h_{b'}(x), h_{b'}(y)$ for each $h_{b'} \in \mathcal{H}_r$. If there exist values $a' < a$ and b' such that x and y collide under $(h_{a'}, h_{b'})$, or if there exists $b' < b$ such that $(h_a, h_{b'})$ collides, the pair (x, y) is a duplicate. □

Since we are exchanging the hash functions to all workers, this check only requires local information. Hence duplicate removal can be performed without an additional communication round, that is without increasing the load of the algorithm. In practice, we check for a pair being a duplicate right after the pair passed the sketch. We remark that the strategy translates naturally to TwoLevelLSH.

Pagh et al. discussed in [22] a simpler approach if the application that uses the result of the join is robust with respect to each pair being reported $O(1)$ times: Let L' be the expected number of times a near pair (x, y) collides under the tensoring approach. When we consider (x, y), we emit the pair (x, y) only with probability $(1 - \delta)/L'$. By linearity of expectation, we expect to report each near pair (x, y) a constant number of times. If the collision probability of a tensor hash function can be evaluated in time $O(1)$, the duplicate check will run in time $O(1)$ per pair.

4 EXPERIMENTS

The goal of this experimental section is to assess the relative merits of the four approaches to data distribution and local computation that we outlined in the previous sections, as well as investigating the influence of parameters on the performance. In particular, we focus on the *self join* case.

Implementation details. We implement[3] all our algorithms on top of *Timely Dataflow* [21], using Rust 1.51 *nightly* (`c5a96fb79` 2021-01-19). Our experimental setup allows to easily tune parameters, so that the interested reader can further explore the behavior of the implementations [5].

Figure 1 provides an overview over the dataflow of the four different implementations considered in this paper.

Load balancing. When data is partitioned across the p processors using LSH, we run all the repetitions in a single distributed round, balancing the load similarly to [15]. Let n be the total number of hashed values. If a hash sees more than n/p collisions, we split it into smaller subproblems of size n/p: all the candidate output pairs are considered by means of a cartesian product of the subproblems. Then, subproblems are sorted by decreasing size, and assigned to processors greedily: a processor accepts subproblems until its load does not exceed n/p. Once the assignment is computed, the hashed values are distributed accordingly and the implementation proceeds to find similar pairs. Note that in [15] the authors propose a slightly different *practical* join algorithm based on sampling rather than collecting the exact histograms of hash values. Nonetheless our heuristic is able to successfully balance the load while requiring negligible time, compared to the overall running time.

[3]https://github.com/Cecca/danny

Table 3: Datasets considered in this experimental evaluation. For the two similarity threshold we consider, we report the average number of neighbors per point. The *dim* column reports the number of dimensions of the vectors for `Glove` and `SIFT`, and the size of the universe for `Orkut` and `Livejournal`.

				avg. neighb.	
Data	n	*dim*	size	0.5	0.7
SIFT	1 000 000	128	522 Mb	2 832.7	115.9
Livej.	3 201 203	7 489 073	486 Mb	51.7	15.6
Orkut	2 783 196	8 730 857	1.3 Gb	1.7	1.1
Glove	1 193 514	200	924 Mb	76.8	2.2

Experimental setup. To evaluate similarity predicates on the Jaccard distance, we use the verification procedure of [18, Algorithm 1]. As for the cosine distance, we normalize all the vectors at read time, and thus only compute the inner product (with SIMD instructions).

To measure the actual time required to perform the join, we simply count the number of matching pairs, without reporting them explicitly. Doing so would in fact perturb the measurement with the time required to output large volumes of data, which is an issue orthogonal to the goals of our experimental evaluation. Nevertheless, counting the matching pairs is still sufficient to compute the recall, since all the algorithms we consider remove duplicates and do not return false positives.

We run our implementation on a 5 node cluster, with a Intel©Xeon©CPU E5-1650 v3 @ 3.50GHz processor each with 8 cores and 32 Gb of memory. Therefore we have $p = 40$. The nodes are connected by a 1 Gb/s network. We report on experiments on a larger cluster in Section 4.5.

Datasets. We use four datasets as a benchmark, whose characteristics are reported in Table 3. `Glove` is a word vector dataset, while `SIFT` is a standard dataset for general Euclidean space. To avoid implementing another LSH function, we use the kernel embedding by Rahimi and Recht [23] to embed vectors onto the unit sphere. Vectors are dense, and the metric we use is the cosine similarity. `Livejournal` and `Orkut` are datasets of users of social networks. Each element of these datasets is a set, and the metric used is the Jaccard similarity. We employ two similarity thresholds: 0.5 and 0.7. These are the thresholds used in [14] as well. As can be seen in Table 3, the former gives substantially larger outputs. In particular, `Orkut` features a very small output size, whereas `SIFT` has a very large output, for both thresholds[4]. In the remainder of the paper, recalls are computed with respect to the output sizes computed using Cartesian with no sketches.

[4] Another threshold choice would be to use different thresholds per dataset, for example by fixing OUT to some constant and choosing the threshold that results in OUT close pairs on average. This choice would entail using different number of repetitions for each datasets, which would reflect on the running time and memory usage. Conversely, having different output sizes with the same number of repetitions allows to inspect the behavior of the algorithms because of widely different output sizes.

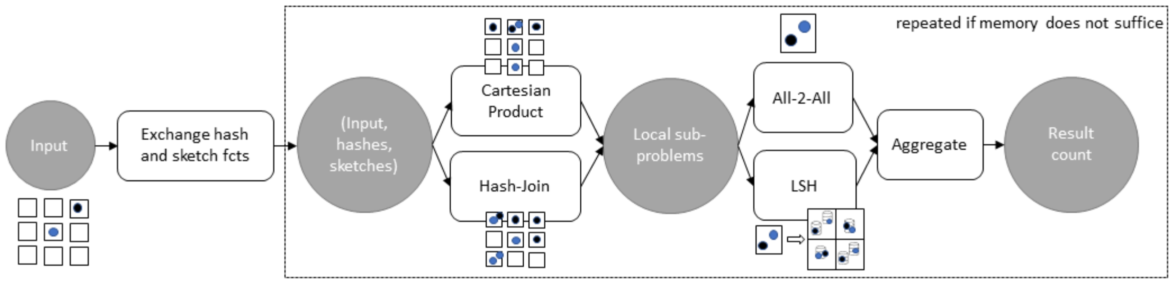
Input
Exchange hash and sketch fcts
(Input, hashes, sketches)
Cartesian Product
Hash-Join
Local sub-problems
All-2-All
LSH
Aggregate
Result count
repeated if memory does not suffice

Figure 1: Overview over the dataflow of the different implementations considered. The example shows 9 workers arranged in a 3-by-3 grid, and focuses on the flow of two vectors depicted as a blue and a black circle, initially residing in two of the workers. Data is distributed to individual workers using the cartesian product (Cartesian, LocalLSH) or a hash-join on LSH values (OneLevelLSH, TwoLevelLSH, 4 copies in the example). Next, the local subproblems on the worker are solved using an all-to-all comparison (Cartesian, OneLevelLSH) in quadratic time, or they are solved using LSH locally (LocalLSH, TwoLevelLSH, using 4 hash functions) in expected subquadratic running time.

Table 4: Best configuration for each algorithm on each dataset. Running times reported in *minutes*.

		0.5				0.7			
data.	alg.	time	recall	k	b	time	recall	k	b
Glove	Cart.	21.9	1.00	0	256	16.2	1.00	0	256
Glove	Local	4.1	0.84	8	512	1.4	0.87	8	256
Glove	1 Level	20.0	0.83	6	512	5.5	0.87	8	512
Glove	2 Level	12.9	0.83	3	512	2.4	0.79	3	512
SIFT	Cart.	18.0	1.00	0	256	11.7	1.00	0	256
SIFT	Local	7.0	0.83	8	512	1.4	0.84	8	512
SIFT	1 Level	21.5	0.82	6	512	3.5	0.84	8	512
SIFT	2 Level	67.7	0.85	3	512	4.2	0.87	3	512
LiveJ	Cart.	122.6	1.00	0	256	115.2	1.00	0	0
LiveJ	Local	4.0	0.88	17	256	1.0	0.99	19	0
LiveJ	1 Level	8.3	0.88	8	256	2.0	0.99	12	256
LiveJ	2 Level	6.9	0.88	4	256	1.2	0.99	4	256
Orkut	Cart.	93.2	1.00	0	256	93.1	1.00	0	256
Orkut	Local	3.4	0.84	8	256	1.4	1.00	20	0
Orkut	1 Level	6.6	0.84	8	256	2.0	0.99	8	256
Orkut	2 Level	4.2	0.84	4	256	1.0	1.00	4	256

Research questions. Our experiments aim at answering the following questions:

(1) How do the different data distribution and local computation strategies compare with each other? (§ 4.1)
(2) What is the influence of k on the performance? (§ 4.2)
(3) What is the influence of sketching on the running time? (§ 4.3)
(4) How expensive is removing duplicates? (§ 4.4)

Parameter setting. In the following experimental evaluation, we test values of $k \in [3, 20]$ for all algorithms, and $k_{inner} \in [4, 12]$ for TwoLevelLSH. When using sketches, we test sketch sizes of 256 and 512 bits. The number of LSH repetitions is set according to Equation 1. If not stated otherwise, we perform enough repetitions to get a guaranteed recall of 0.8. Note that not all combinations of parameters are feasible, in that they may entail prohibitive use of memory or impractically long running times, in which case the corresponding data points are missing from the figures.

4.1 Algorithm comparison

As a first step, we take a bird's eye view of the performance of the algorithms, concentrating on the configurations giving the best performance, reported in Table 4. In the next sections we will investigate the influence of each parameter on the performance. For reference, we include the running time for Cartesian, which evaluates all possible pairs using sketches to speed up the computation.

First, we notice how LocalLSH is the fastest algorithm on all datasets, with the exception of SIFT at threshold 0.7, running 2 to 30 times faster than Cartesian. Remember that LocalLSH and Cartesian leverage the same data distribution strategy, while choosing different strategies for solving the subproblems locally. Therefore, the gap between the two algorithms is a measure of the impact of using LSH locally to speed up the computation.

As for the algorithms partitioning data using LSH, for threshold 0.5 they perform 2 to 9 times worse than LocalLSH, even though they are faster than Cartesian (with the notable exception of `SIFT` at similarity 0.5). Even though both algorithms are able to theoretically take advantage of a lower load compared to LocalLSH, in practice distributing several copies of the data (one for each repetition) to each processor counterbalances the potential gains. At a higher similarity threshold such as 0.7, however, where fewer repetitions are needed, they both become competitive with LocalLSH for Jaccard-based datasets.

As for the quality of the solution, the recall obtained by all algorithms is always larger than 0.8, as expected since we are setting the number of repetitions accordingly. For Jaccard-based datasets at threshold 0.7, the recall is *much* better than 0.8.

In general, all the algorithms benefit from the use of sketching, as we will detail in Section 4.3. The only exception we observed is for LocalLSH on `Livejournal` and `Orkut` at threshold 0.7, where LSH is already selective enough to make sketching redundant.

4.2 Dependency on k

We now focus on the influence of the parameter k on the performance of the algorithms. For the sake of clarity in the interpretation of the results, in this set of experiments we count the number of candidate pairs that need to be evaluated under each parameter configuration, without carrying out the actual similarity verifications. This also allows to assess the performance of non-optimal configurations, whose actual running time would

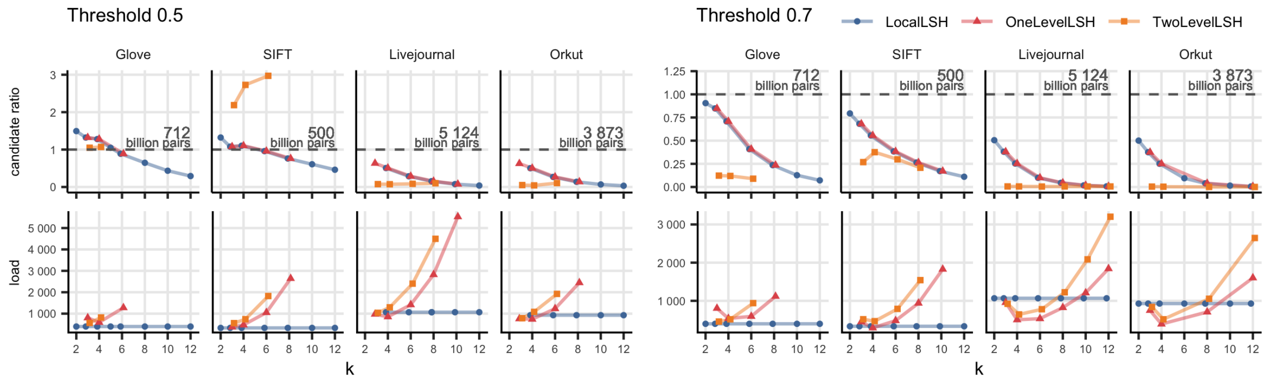

Figure 2: Dependency of the number of distance computations and load on the value of the parameter k. The dashed line reports the number of distance computations carried out by the cartesian product.

be impractically high. In Section 4.3, we will run experiments reporting the actual running times of the best configuration.

Figure 2 reports, for both similarity thresholds, the number of candidates as a fraction of the total number of pairs (top) and loads (bottom) against different values of k. The former measure allows to assess how effectively each approach can prune the number of candidates. Values larger than 1 indicate that a particular parameter configuration evaluates the similarity of more pairs than the naive Cartesian algorithm. The dashed horizontal line reports the number of pairs evaluated by the Cartesian algorithm, for reference. To follow more closely the MPC model, the load is measured in terms of maximum number of *messages* received by a processor, not by their size. For TwoLevelLSH, we set the k for solving local subproblems with LSH to 12, which we found to be the best among the several values we tested. Data points for OneLevelLSH and TwoLevelLSH for high values of k are missing from the plots: The reason is that, as can be observed from the bottom row of plots in Figure 2, the load increases with k, to the point that the machines of our clusters do not have enough memory to process all the messages at once. Our implementation supports to divide the computation into several parallel rounds to meet the memory limits, but this further decreased the performance in a parameter range that already turned out to be non-competitive, see Table 4.

We observe that for all algorithms the number of candidates is heavily influenced by k. The best value of k for OneLevelLSH and TwoLevelLSH is smaller than the best for LocalLSH: using a smaller value implies less duplication of the points due to the need of doing fewer repetitions, even if this requires dealing with larger subproblems compared to higher values of k.

Interestingly, the value of k minimizing the number of distance computations for OneLevelLSH and TwoLevelLSH is not the one minimizing the load, a fact which is eventually reflected in the running time. Consider for instance `Livejournal` at threshold 0.5. OneLevelLSH with $k = 4$ has load $\approx$ 851 000 and needs to evaluate $\approx$ 50% of the pairs, whereas with $k = 8$ has load over 2 000 000, but evaluates only $\approx$ 15% of the pairs evaluated by Cartesian. We stress that many of these pairs will be duplicates. As we shall see in Section 4.4, discarding duplicates is considerably faster than verifying the similarity of a pair. Therefore, tuning k to minimize the load might not the best strategy: rather, it should be tuned in such a way as to strike the balance between the load of the processors and the cost of solving the subproblems. As expected, the load of LocalLSH is constant across all values of k, since it depends only on the size of the input rather than its distribution.

On all datasets, LocalLSH and OneLevelLSH perform a comparable number of distance evaluations, for the same value of k. This is expected, since both prune pairs to be compared using LSH functions configured in the same way. The difference between the two algorithms lies in the load, which is generally higher for OneLevelLSH, with the exception `Livejournal` and `Orkut` at similarity threshold 0.7. This difference in load makes OneLevelLSH in general slower than LocalLSH, even though they evaluate a comparable number of candidate pairs.

On the other hand, TwoLevelLSH allows to prune a lot of candidate pairs, at the expense of a higher number of repetitions. This is reflected in a load which is much higher than that of OneLevelLSH. In the next section, we shall see how the aggressive pruning of TwoLevelLSH can in some cases balance the increased load, making it competitive with LocalLSH.

Overall, we note that LocalLSH has a very predictable behavior, which allows to exercise a wide parameter range on all datasets[5]. OneLevelLSH and TwoLevelLSH, on the other hand, are much more sensitive to setting parameters correctly: as is clear from Figure 2, setting k to too large values—trying to prune many candidate pairs—leads to very high load; this can possibly exceed the memory limits of the cluster, as is happening whenever there is a missing point in the plot.

As we will see in the next section, using sketches to speed-up local work changes the influence of k on the performance, to the point that the configuration minimizing the number of distance computations might not be the fastest.

4.3 Effect of sketching

To further reduce the cost of local work, we can use sketching as described in Section 3.3 in order to discard far away pairs without the need of verifying their similarity. Specifically, we set sketches so that close pairs are rejected with probability at most 1%.

First, to assess the influence of sketching on the recall, we run the Cartesian algorithm with 256, and 512-bits sketches, comparing the size of the output with the baseline without sketching. We found that at most 0.5% of the output pairs are lost due to sketching (twice as good as the theoretical guarantee), therefore we conclude that the effect of sketching on the recall is negligible.

[5]In the supplemental material (https://doi.org/10.6084/m9.figshare.c.5598171) we provide an extended version of Figure 2, testing LocalLSH with k up to 20.

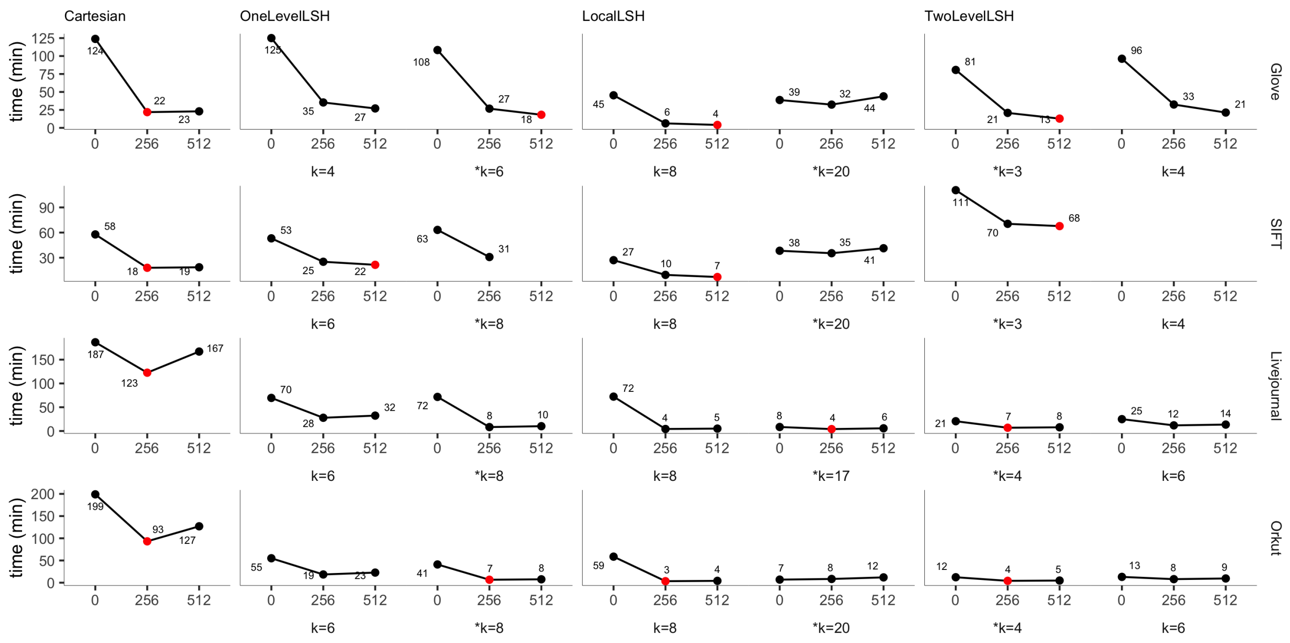

Figure 3: Influence of sketches of different sizes on the runtime performance, at similarity threshold 0.5. The red dot marks the best configuration for a given algorithm. Each point is labeled with the running time in minutes. When a value is missing, the corresponding experiment crashed due to exceeding the memory capacity of our cluster.

Figure 3 reports on the results for sketches of different sizes in combination with different values of the parameter k. A higher resolution image is provided in the supplemental material. For LocalLSH, we try the value of k minimizing the number of distance computations (17 for `Livejournal`, 20 for the others) and $k = 8$, in order to be able to assess the effect of sketching on fairly different configurations. As for OneLevelLSH and TwoLevelLSH, due to the smaller parameter range that can be exercised before exhausting the memory available on the system, we test the value of k minimizing the number of distance computations along with a neighboring k value. The value of k minimizing distance computations is marked with * in the plots.

Before considering the impact of sketching on the performance, we consider the running times without sketches (first dot in each line of Figure 3). The results are in line with our findings from the previous section, where we used the number of distance computations as a proxy for running time performance. LocalLSH is the fastest algorithm, on `Glove` and `SIFT`, whereas on `Livejournal` and `Orkut` TwoLevelLSH is the best performing, with LocalLSH and OneLevelLSH running in comparable time.

We note that the introduction of sketching greatly improves the general performance. In particular, for `Orkut` and `Livejournal` the speedup is between 3 and 18, and for `Glove` and `SIFT` it is between 3 and 6.

The Cartesian algorithm has the best performance with 256-bits sketches: using larger sketches deteriorates the performance since it requires more communication without a compensating gain in local work. Note that LocalLSH incurs the same communication as Cartesian, but is not affected by a similar performance degradation with larger sketches. This is because it performs less local work compared to Cartesian due to the usage of LSH, thus balancing the increased network communication. TwoLevelLSH, on the other hand, seems to benefit less from the use of sketching: the maximum improvement is ≈ 6 (for `Glove`), compared with the ≈ 18 speedup attained by LocalLSH on `Livejournal`. In fact, its aggressive use of LSH implies that most of the local work consists in evaluating repetitions of close pairs, which cannot be discarded by sketching. For OneLevelLSH, the sketching can make its performance comparable with the other two algorithms.

The runs of LocalLSH make for a particularly interesting case. The higher value of k makes the algorithm perform less distance computations compared to the smaller one. Nonetheless, on `Glove` and `SIFT` the running times are comparable with both values of k when sketches are not used, with the smaller k value giving the best performance when sketches are in use. On `Livejournal` and `Orkut`, without sketches a higher k value is indeed beneficial in terms of performance. When sketches are used, however, the running times become comparable. The reason is that discarding far-away pairs is so efficient with sketching that we can afford to evaluate more such pairs using a smaller k.

Unfortunately, there does not seem to be a single sketch size that works best in all situations: large sketches work best on datasets based on the cosine similarity, whereas on `Livejournal`

Table 5: Benchmark of individual verification operations. Times are in nanoseconds.

		sketch		deduplication		similarity	
data type	pair	median	max	median	max	median	max
Cosine	close	7.0	9.0	9.0	28.0	30.0	37.0
Cosine	far	7.0	8.0	9.0	31.0	30.0	38.0
Jaccard	close	7.0	8.0	7.0	17.0	23.0	418.0
Jaccard	far	7.0	9.0	8.0	17.0	11.0	267.0

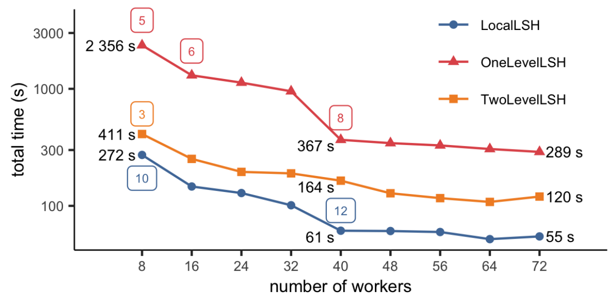

Figure 4: Scalability with respect to the number of workers.

and Orkut 256-bit sketches work better, even though the difference in running time between different sketch sizes is not so marked.

4.4 Micro-benchmarks

We concentrate now on the cost of the individual steps of the evaluation pipeline, namely sketch evaluation, similarity verification and duplicate elimination using tensoring. We run the following benchmark: at similarity 0.5 we sample 10 000 close pairs and 10 000 far pairs from the Glove dataset (for cosine similarity) and Livejournal (for Jaccard similarity). Then, for each pair, we measure the time employed to run 10 000 times each of the aforementioned operations on each pair individually, using the arithmetic mean as an estimate for the mean of the individual invocation. The results are reported in Table 5.

Evaluating sketches is very fast and stable in terms of running time, and does not depend on the similarity measure nor on the actual similarity of the pair. This is in line with our expectations, since we implement sketch comparison with simple bitwise operations. On the other hand, the verification of the similarity predicate is more expensive and subject to greater variability in the case of Jaccard similarity. In fact, since the verification procedure described in [18] stops as soon as the similarity is bound to be above the threshold, far pairs are cheaper to verify than close ones.

These measurements allow us to put in context the cost of duplicate removal using tensoring that we described in Section 3.4. We have that our procedure for duplicate removal has a runtime cost comparable to sketching and much less than similarity verification. Therefore it can be used to remove the duplicates introduced by LSH repetitions with a low overhead, certainly less than the cost of the cluster-wide shuffle that is usually leveraged to remove duplicates.

4.5 Scalability

We now focus on the scalability of the three implementations, running experiments on a cluster with 9 machines, each equipped with 32GiB of RAM, an Intel i7-4790 CPU @ 3.60GHz with 4 cores, connected by a 1Gbit network. On each machine, we spawn 8 workers, which allows us to run scalability experiments with 8-72 workers on 1-9 physical nodes.

We consider the Glove dataset, with similarity threshold 0.7. For a given number of workers, we test several parameters for each algorithm, reporting the performance of the best configuration. Note that this is consistent with the theoretical analysis, where changing the number of processors might change the best value of k.

Figure 4 reports the results of this experiment. Points are annotated with a text box reporting the value of k used, whenever it changes going from left to right.

Consider the LocalLSH algorithm. From 8 to 40 workers it exhibits almost ideal scalability: using 5 times more resources yields a ≈4.5 speedup. From 40 workers onwards, we hit a plateau due to the increased communication cost balancing the reduced cost of local computation. As for OneLevelLSH, using a higher k creates more subproblems (hence more opportunities for parallelism) at the expense of a higher memory requirement. The first consequence is that the best value of k increases with the number of workers, as expected. Second, we see sharp improvements of the running time as soon as the system has enough memory to allow the usage of a higher k value: going from 8 to 40 workers gives 6 fold speedup. For a fixed value of k the improvement in running times is much less pronounced. Finally, for TwoLevelLSH the best configuration is $k = 3$ for data distribution and $k = 12$ for solving subproblems locally, for all the number of workers we tested. The speedup from 8 to 40 workers is ≈2.5. Above 40 workers, however, the number of generated subproblems is so small that most workers will not see their workload decrease. This would be improved by using a larger k value, but the additional communication and replication did not yield faster running times for the dataset.

4.6 System profiling

Finally, we inspect the usage of system resources throughout the execution of each algorithm. Figure 5 reports the CPU, memory, and network usage of one machine in the experiment with 40 workers presented in the previous subsection, with the x-axis reporting the elapsed time. All three implementations fully utilize the CPU, even when network communication is underway. The reason is that while communication is ongoing, the implementation is using the CPU to arrange data for more efficient access at later stages of execution. There are three execution stages: at first hash values and sketches are computed locally (used memory increases, but the network is not used)[6], then hash values, sketches, and input vectors are exchanged on the network, and finally the subproblems are solved locally (used memory decreases as parts of the subproblems are solved, and the network is not used). LocalLSH, other than being the fastest in this setting, is also the implementation using the network for the shortest span of time. TwoLevelLSH uses the network for a shorter time compared to OneLevelLSH (because using a smaller k for the data distribution implies a smaller number of repetitions), but requires a longer setup phase, since the hash values for solving the subproblems need to be computed as well.

We now give some intuition about the memory usage of different algorithms. In the implementation, we have to differentiate between the memory usage of the actual data structures and the overhead induces by buffers of the distributed system.

As discussed in Section 2.5, for a given k and recall guarantee of $1 - \delta$, we carry out $\Theta(\ln(1/\delta)/q_1^k)$ many repetitions in the LSH scheme, where the constant in the notation will be at most 2. For both LocalLSH and OneLevelLSH, this is carried out once; for TwoLevelLSH each subproblem is again considered as the input for an LSH. For a vector being part of a repetition, we store both the vector and its sketch on the machine. For example, for Glove

[6]This part is clearly visible only for TwoLevelLSH

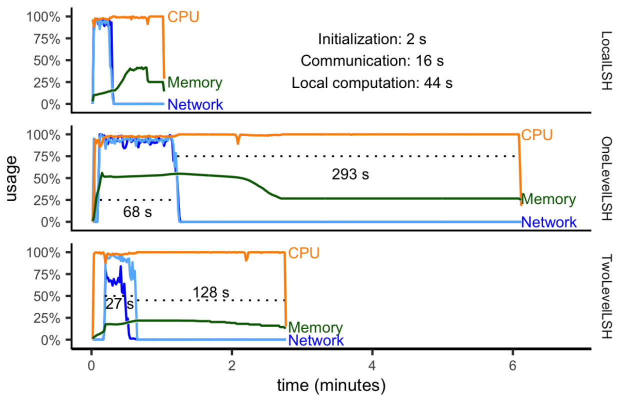

Figure 5: System profiling for a single machine.

and 512 bit sketches, a single vector takes up $800 + 512/8 = 864$ bytes. For OneLevelLSH with $k = 6$ on `Glove` with threshold 0.7, we carry out 16 repetitions which results in a total space usage roughly 18GB. Initially, there is also a memory overhead on the worker for reading the input from disk.

On the configuration considered in this section each worker, to store input vectors, sketches, and pools of hash values received after the exchange on the network[7], uses ≈240 Mbytes for LocalLSH, ≈1045 Mbytes for OneLevelLSH (≈28 Mbytes per LSH repetition), and ≈ 248 Mbytes for TwoLevelLSH (≈50 Mbytes per LSH repetition). LocalLSH allocates the smallest memory since data is replicated fewer times, compared to the other two approaches. TwoLevelLSH allocates almost the same memory overall, and much fewer than OneLevelLSH since it performs fewer LSH repetitions in the data distribution phase, due to the smaller value of k being used. As it can be seen from Figure 5, memory measurements are around a factor of two larger, due to the communication buffers and auxiliary data structures used by Timely Dataflow.

4.7 Discussion

We summarize our main empirical takeaways as follows.

(1) It is difficult to improve on data-independent data distribution with a small cluster.
(2) A holistic approach on the whole similarity join pipeline is necessary to find good parameter choices.
(3) Sketches not only provide a way to speed up the join computation, but also make the parameter selection more robust.
(4) Tensoring-based duplicate detection provides an efficient way to discard duplicates using only local information.

We provide further discussion on the first item. A cluster with $p = 72$ provides a difficult setup for data-dependent data distribution: Given that data-independent partitioning replicates each point $\sqrt{p} < 9$ times, it is difficult to incur smaller load using LSH because of the necessary repetitions. Only for Jaccard-based datasets, we could identify parameter choices that provided smaller load. For OneLevelLSH, this did not translate into speedups; as predicted by theory, the associated choices of k incur too much local work. On the other hand, TwoLevelLSH can—in these cases—provide a speedup by providing smaller load with local work similar to LocalLSH. Again, parameter choices follow theory nicely: A small k_{outer} minimizes the load, a large k_{inner} minimizes local work.

[7]We do not take into account the memory needed to allocate auxiliary data structures and communication buffers.

Given that this paper uses OneLevelLSH by Hu et al. [15] as a starting point and provides further analyses and evaluation of their approach, our recommendations are as follows: Use LocalLSH—avoiding data-dependent distribution—if $\sqrt{p}$ is small; otherwise, TwoLevelLSH provides a way to incur smaller load, following parameter choices from [15] and using a larger k_{inner} as discussed in the proof of Theorem 3.2.

Our selection of datasets included the most challenging datasets from the large-scale distributed similarity join study [14]. On these datasets, the implementations in [14] timed out or exceeded compute capabilities on a cluster larger than ours. Furthermore, our results generalize to larger datasets with regard to recommendations on which approach to use. On larger clusters, data-dependent data distribution is more competitive, also for smaller thresholds.

5 CONCLUSION AND FUTURE WORK

In this paper, we provided further analyses and evaluation of the nearly load-optimal similarity-join algorithm by Hu et al. [15]. We extended the analysis and described the whole design space of LSH-based similarity join algorithms. Experimentally, we showed the importance of other techniques such as sketching and efficient duplicate elimination.

Using LSH to distribute work across processors is challenging: while it can be used to control the load of workers, the data replication introduced by repetitions can cancel out the benefits. In connection with this, however, we observed that the recall provided by LSH is often much higher than the target one: many repetitions are therefore not necessary in practice. Hence, an interesting open question in the distributed setting is to devise a way of stopping repetitions early, as soon as the target recall is reached.

Choosing k wisely is second only to using sketches to get good performance in practice. In this respect, an interesting avenue to explore is setting k adaptively for different subsets of the input, following [1, 19].

It is not clear if LSH is the best approach to compute a similarity join. A nice direction for future work is to compare it to LSF-based approaches [24]. However, the first step would be to devise a LSF framework for computing similarity joins that is as general as the LSH-based model.

ACKNOWLEDGEMENTS

We thank Rasmus Pagh for the many fruitful discussions. We are grateful for the comments and suggestions provided by the anonymous reviewers which improved both content and presentation of this paper.

The research leading to these results has received funding from the European Research Council under the European Union's 7th Framework Programme (FP7/2007-2013) / ERC grant agreement no. 6143311, and from the Free University of Bozen-Bolzano under project SSJoin.

REFERENCES

[1] Thomas D. Ahle, Martin Aumüller, and Rasmus Pagh. 2017. Parameter-free Locality Sensitive Hashing for Spherical Range Reporting. In *SODA*. SIAM, 239–256.
[2] Josh Alman and Ryan Williams. 2015. Probabilistic Polynomials and Hamming Nearest Neighbors. In *FOCS*. IEEE Computer Society, 136–150.
[3] Alexandr Andoni and Piotr Indyk. 2006. Efficient algorithms for substring near neighbor problem. In *SODA*. ACM Press, 1203–1212.

[4] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020).

[5] Martin Aumüller and Matteo Ceccarello. 2020. Running Experiments with Confidence and Sanity. In *SISAP*, Vol. 12440. Springer, 387–395. https://doi.org/10.1007/978-3-030-60936-8_31

[6] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. 2019. PUFFINN: Parameterless and Universally Fast FInding of Nearest Neighbors. In *ESA (LIPIcs)*, Vol. 144. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:16.

[7] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *SEQUENCES*. IEEE, 21–29.

[8] J Lawrence Carter and Mark N Wegman. 1979. Universal classes of hash functions. *J. Comp. Syst. Sci.* 18, 2 (1979), 143–154.

[9] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *STOC*. ACM, 380–388.

[10] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*. IEEE Computer Society, 5.

[11] Tobias Christiani. 2019. Fast Locality-Sensitive Hashing Frameworks for Approximate Near Neighbor Search. In *SISAP*. 3–17. https://doi.org/10.1007/978-3-030-32047-8_1

[12] Tobias Christiani and Rasmus Pagh. 2017. Set similarity search beyond MinHash. In *STOC*. ACM, 1094–1107.

[13] Tobias Christiani, Rasmus Pagh, and Johan Sivertsen. 2018. Scalable and Robust Set Similarity Join. In *ICDE*. IEEE Computer Society, 1240–1243.

[14] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set Similarity Joins on MapReduce: An Experimental Survey. *PVLDB* 11, 10 (2018), 1110–1122.

[15] Xiao Hu, Ke Yi, and Yufei Tao. 2019. Output-Optimal Massively Parallel Algorithms for Similarity Joins. *ACM Trans. Database Syst.* 44, 2 (April 2019), 1–36. https://doi.org/10.1145/3311967

[16] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. ACM, New York, NY, USA, 604–613. https://doi.org/10.1145/276698.276876

[17] Ping Li and Christian König. 2010. B-Bit Minwise Hashing. In *WWW (WWW '10)*. ACM, New York, NY, USA, 671–680. https://doi.org/10.1145/1772690.1772759

[18] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB* 9, 9 (May 2016), 636–647. https://doi.org/10.14778/2947618.2947620

[19] Samuel McCauley and Francesco Silvestri. 2018. Adaptive MapReduce Similarity Joins. In *BeyondMR@SIGMOD*. ACM, 4:1–4:4.

[20] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*. USENIX Association.

[21] Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. 2016. Incremental, Iterative Data Processing with Timely Dataflow. *Commun. ACM* 59, 10 (2016), 75–83. https://doi.org/10.1145/2983551

[22] Rasmus Pagh, Ninh Pham, Francesco Silvestri, and Morten Stöckel. 2017. I/O-Efficient Similarity Join. *Algorithmica* 78, 4 (2017), 1263–1283.

[23] Ali Rahimi and Benjamin Recht. 2008. Random Features for Large-Scale Kernel Machines. In *NIPS*, J. Platt, D. Koller, Y. Singer, and S. Roweis (Eds.), Vol. 20. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2007/file/013a006f03dbc5392effeb8f18fda755-Paper.pdf

[24] Cyrus Rashtchian, Aneesh Sharma, and David P. Woodruff. 2020. LSF-Join: Locality Sensitive Filtering for Distributed All-Pairs Set Similarity Under Skew. In *WWW*. ACM / IW3C2, 2998–3004.

[25] Venu Satuluri and Srinivasan Parthasarathy. 2012. Bayesian Locality Sensitive Hashing for Fast Similarity Search. *PVLDB* 5, 5 (2012), 430–441.

[26] Aneesh Sharma, C. Seshadhri, and Ashish Goel. 2017. When Hashes Met Wedges: A Distributed Algorithm for Finding High Similarity Vectors. In *WWW*. ACM, 431–440.

[27] Yasin N. Silva, Spencer S. Pearson, Jaime Chon, and Ryan Roberts. 2015. Similarity Joins: Their implementation and interactions with other database operators. *Inf. Syst.* 52 (2015), 149–162.