Adaptive MapReduce Similarity Joins

Extended Abstract

Samuel McCauley
BARC and IT U. Copenhagen, Denmark.
samc@itu.dk

Francesco Silvestri
University of Padova, Italy.
silvestri@dei.unipd.it

Abstract

Similarity joins are a fundamental database operation. Given data sets S and R, the goal of a similarity join is to find all points $x \in S$ and $y \in R$ with distance at most r. Recent research has investigated how locality-sensitive hashing (LSH) can be used for similarity join, and in particular two recent lines of work have made exciting progress on LSH-based join performance. Hu, Tao, and Yi (PODS 17) investigated joins in a massively parallel setting, showing strong results that adapt to the size of the output. Meanwhile, Ahle, Aumüller, and Pagh (SODA 17) showed a sequential algorithm that adapts to the structure of the data, matching classic bounds in the worst case but improving them significantly on more structured data.

We show that this adaptive strategy can be adapted to the parallel setting, combining the advantages of these approaches. In particular, we show that a simple modification to Hu et al.'s algorithm achieves bounds that depend on the density of points in the dataset as well as the total outsize of the output. Our algorithm uses no extra parameters over other LSH approaches (in particular, its execution does not depend on the structure of the dataset), and is likely to be efficient in practice.

1 Introduction

Similarity search is a fundamental problem in computer science where we seek to find items that are similar to one another. In this paper, we focus on the problem of similarity joins for high dimensional data, which can be viewed as a large number of batched similarity searches. In particular, given two sets R and S, we wish to find all pairs (x, y) (with $x \in R$ and $y \in S$) where x and y have similarity above some threshold r. Our results are largely agnostic to the particular similarity function used; for example one can immediately apply our techniques to Hamming distance, ℓ_1 or ℓ_2 distances, cosine similarity, Jaccard or Braun-Blanquet similarity [9] or even more exotic measures like Frechet distance [10]. Similarity joins have wide-ranging applications, such as web deduplication [5], document clustering [7], and data cleaning [3].

Unfortunately, similarity joins are extremely computationally intensive.[1] For this reason, when performing similarity joins on large datasets it is often useful to use massively parallel machines using frameworks like MapReduce and Spark. A recent work [12] has proposed an output sensitive MapReduce algorithm that leverages Locality Sensitive Hashing (LSH). When executed on p machines and on two relations containing n tuples, their solution requires $O(1)$ rounds and load (i.e., maximum number of messages received/sent by a processor)

$$O\left(\sqrt{\frac{\mathrm{OUT}_r}{p}p^{\rho/(1+\rho)}} + \sqrt{\frac{\mathrm{OUT}_{cr}}{p}} + \frac{n}{p}p^{\rho/(1+\rho)}\right), \tag{1}$$

where OUT_r is the number of pairs with distances smaller than or equal to r, OUT_{cr} is the number of pairs with distances in the range $(r, cr]$, and $\rho \in [0, 1]$ is a value characterizing the LSH (see Section 2.4). This bound highlights some limitations of the standard LSH approach: there is an OUT_{cr} term due to "false

[1] In fact, this large computation may be unavoidable for some metrics, see e.g. [17].

positives" of the LSH, and there is a multiplicative term $p^{\rho/(1+\rho)}$ in the OUT_r contribution due to near pairs being reported multiple times.

In this paper we show how the load of the previous algorithm can be improved by exploiting the novel LSH approach presented in [2], which leverages a multi-level LSH data structure for solving the range reporting problem. Specifically, we set a small term κ based on p and the LSH parameters. For each $1 \leq i < \kappa$, we say that a point in $x \in R$ is *i-dense* if its number of near points in S is in the range $[np_2^{i-1}, np_2^i)$; a point is *κ-dense* if its number of near points in S is smaller than np_2^κ (similar definitions hold for points in S). We let OUT_r, i denote the number of near pairs containing at least one i-dense point. In this paper, we describe an MPC algorithm requiring $O(1)$ rounds and load

$$\tilde{O}\left(\sqrt{\left(\sum_{i=0}^{\kappa} \frac{\mathrm{OUT}_{r,i}}{pp_1^i}\right)} + \sqrt{\frac{\mathrm{OUT}_{cr}}{p}} + \frac{n}{p}p^{\rho/(1+\rho)}\right).$$

In the most extreme cases, there are no dense clusters (i.e. all close pairs contain only κ-dense points), in which case we get the same performance as [12]. However, the data structure has any dense clusters we get improved bounds. For example, if there is a cluster of $\Omega(n)$ close points, the first term of Hu et al. is $O(n/p^{1/(2(1+\rho))})$ while for us it is $O(n/p^{1/2})$. Our bounds give a smooth decrease in performance as the size of the cluster decreases.

1.1 Related Work

Exact similarity search has been widely studied in the literature; we refer to [4] for a survey. Approximate algorithms for similarity join often rely on LSH, with the underlying idea to adapt the indexing approach in [11]. This approach has been adapted for use in the I/O model [14] and in the MPC model [12].

A novel sequential LSH approach was recently introduced [2] which dynamically adapts to the difficulty of each query; our result implements this idea in a massively parallel setting. In short, this approach uses a simple recursive stopping rule to adapt to the structure of the dataset; we adapt this rule to the similarity join setting. Recently, the paper [9] adapted this approach for similarity join under Braun-Blanquet similarity; our results share the same basic principles but apply to more general distance metrics.

2 Preliminaries

In this section, we describe the adopted computational model and some relevant results on equi-joins, similarity search and LSH.

2.1 Computational model

In the literature, there are several computational models for massively parallel systems aiming at describing MapReduce-like systems (e.g., [15, 6, 13]). The majority of these models are very close to the bulk-synchronous parallel (BSP) model by L. Valiant [16], and in general they differ from the BSP on the cost functions (e.g., round number vs running time), parameters (e.g., local and global memory vs bandwidth and latency), and on some modeling aspects (e.g., a dynamic number of processing units vs a fixed number of processors in order to capture elastic settings in cloud).

In this paper, in continuity with the previous work on similarity join in MapReduce [12], we use the *Massively Parallel Computational* (MPC) model in [6]. It consists of p processors $P_1, P_2, \ldots, P_p$ that are connected by a complete network. In each round, each server does some local computation and then sends messages to other servers, which will be received at the beginning of the subsequent round. The complexity of the algorithm is the *number of rounds* and the *load* L, which is the maximum size of sent/received messages by each processor in each round. For simplicity, we assume in the paper that $p < n^\epsilon$ for some constant $0 < \epsilon < 1$: this implies that sorting and prefix sum computations on input size n can be performed in $O(1)$ rounds and load $O(n/p)$. In general, the goal is to design MapReduce algorithms with a constant number of

rounds. However, some works have shown that there are some inherent tradeoffs between round number and total communciation cost [8, 1].

2.2 Equi-Join

Let R, S be two relations of total size n. The *equi-join* of R and S, denoted with $R \bowtie S$, is the set containing all pairs (r, s) such that $r \in R$, $s \in S$ and $r = s$. Hu et al. [12] provided an optimal output sensitive and $O(1)$-round MPC algorithm for equi-join, with the following bounds:

Theorem 1 ([12]). *There is an optimal deterministic algorithm that computes the equi-join between two relations of total size n in $O(1)$ rounds and with load $\Theta\left(\sqrt{O/p} + n/p\right)$, where $O = |R \bowtie S|$ denotes the equi-join size. It does not assume any prior statistical information about the data.*

The term optimal holds for tuple-based algorithms, that is algorithms where tuples are atomic elements that must be processed or communicated in their entirety (i.e., indivisibility assumption).

2.3 Similarity search problems

Consider a space $\mathbb{U}$ and a distance function $d : \mathbb{U} \times \mathbb{U} \to \mathbf{R}$. Let $r > 0$ be an input parameter. The *r-near neigbor* problem is defined as follows: given a set R of n points from $\mathbb{U}$ and a query point $q \in \mathbb{U}$, return a point x in R at distance $d(x, q) \leq r$ if it exists. The approximate version of the problem, named the *(r, c)-near neighbor* problem where $c > 1$ is the approximation factor, returns a point x in R with distance $d(x, q) \leq cr$ when there exists a point $x' \in R$ with distance $d(x', q) \leq r$. The *r-range reporting problem* requires to find all points at distance at most r from a given query q.

For convenience, we say that a pair (x, y) is *far* if $d(x, y) > cr$ (those that should not be reported), *near* if $d(x, y) \leq r$ (those that should be reported), and c-near if $r < d(x, y) \leq cr$ (those that should not be reported but the LSH provides no collision guarantees). We also assume that each point in $\mathbb{U}$ can be stored in $O(1)$ memory words and that $d(x, y)$ can be computed in constant time (it is easy to extend bounds to the general case).

The similarity join problem is a batch version of the near neigbor problem. Specifically, the *similarity join* with radius $r > 0$ on the sets $R, S \subseteq \mathbb{U}$ is defined as the set $R \bowtie_{\leq r} S = \{(x, y) \in R \times S \mid d(x, y) \leq r\}$. We let $n = |R| + |S|$. In this paper, we will give a randomized solution that finds all pairs in $R \bowtie_{\leq r} S$ with high probability. The proposed solution will generate also pairs at distance larger than r (i.e., false positives): while false positives can be easily removed by checking the true distance before emitting a pair, they still affect the performance of the algorithm (see the term OUT_r in the load upper bound).

2.4 Locality-Sensitive Hashing

Much of recent work on similarity search and join has focused on locality-sensitive hashing (LSH). In LSH, we hash each item in the dataset to a single hash bucket given by a randomized hash function. Once the entire dataset is hashed, we perform a brute-force comparison between all pairs of points in the bucket, returning any similar points found. The key idea behind LSH is the following property. At a high level, an LSH family must map similar points to the same bucket with a much higher probability than far points. Formally, we have:

Definition 2. *Fix a distance function $d : \mathbb{U} \times \mathbb{U} \to \mathbf{R}$. A* locality-sensitive hash *(LSH) family $\mathcal{H}$ is a family of functions $h : \mathbb{U} \to \mathbf{R}$ such that for each pair $x, y \in X$ and a random $h \in \mathcal{H}$, for arbitrary $q \in \mathbb{U}$, whenever $d(q, x) \leq d(q, y)$ we have $\Pr[h(q) = h(x)] \geq \Pr[h(q) = h(y)]$.*

When an LSH is applied to solve the (c, r)-near neighbor problem, it is common to describe the scheme with the probabilities p_1 and p_2 defined as follows: for each x, y with $d(x, y) \leq r$ then $\Pr[h(x) = h(y)] \geq p_1$; for each x, y with $d(x, y) > cr$ then $\Pr[h(x) = h(y)] \leq p_2$ (there are no requirements for pairs with distance

in $(r, cr]$). We define $\rho(r_1, r_2) = \log p(r_1)/\log p(r_2)$, where $p(r)$ is the collision probability at distance r, and define $\rho = \rho(r, cr)$.

Oftentimes, probabilities p_1 and p_2 are constants. Thus, hashing a single time would lead to poor recall (we would be likely to miss close points) and poor precision (most points—np_2 of them to be precise—would hash to a given bucket in expectation, so searching within a bucket would be extremely costly). For many use cases, this can be handled using a two-pronged approach: we concatenate many hash functions to improve precision, and use many independent repetitions to improve recall. This approach can be formalized as follows, using a k-concatenated hash $\mathcal{H}_k$. Let $\mathcal{H}$ be a hash family with $p_1, p_2 = \Omega(1)$. Let $H_k = (h_1, h_2, \ldots, h_k)$ be a hash function consisting of the concatenation of k independent hash functions from $\mathcal{H}$, and let $k = \log_{1/p_2} n$. Then

- for x, y with $s(x, y) \geq r$, $\Pr(H_k(x) = H_k(y)) \geq 1/n^\rho$. Thus, after n^ρ independent repetitions, x and y will hash to the same bucket with constant probability.
- for x, y with $s(x, y) \leq cr$, $\Pr(H_k(x) = H_k(y)) \leq 1/n$. Thus, each bucket will contain one far point in expectation.

In a single-processor setting, this framework allows us to perform single similarity searches in $\tilde{O}(n^\rho)$ time. Let R and S be sets of size n. Let $R \bowtie_r S$ denote the set of pairs $x \in R$ and $y \in S$ with $s(x, y) \geq r$; likewise let $R \bowtie_{cr} S$ denote the set of pairs with $s(x, y) \geq cr$. Then the join between S and R can be computed using a k-concatenated LSH with parameters p_1 and p_2 in time $O\left(n^{1+\rho} + n^\rho |R \bowtie_r S| + |R \bowtie_{cr} S|\right)$. These LSH-based approaches form the basic building blocks of this paper.

2.5 Adaptive Near Neighbor

Our work leverages the recent work by Ahle et al. [2] that presents a data structure for the range reporting problem (in which we wish to find all near points to a query). Let $N_r(q)$ be the number of near points to a query q (similarly, $N_{cr}(q)$ is the number of c-near points). Let the *expansion* c_q^* be the largest value such that there are at most twice as many c_q^*-near points of q than there are r-near points of q. (i.e. $N_{cr}(q) \leq 2N_r(q)$).

Theorem 3 ([2])**.** *Consider a query point q, and parameters $r > 0$ and $c > 1$. Then, the near neighbors of q can be reported with constant probability in time*

- $O\left(N_r(q)(n/N_r(q))^{\rho(r, c_q^*)}\right)$ *if* $c_q^* \geq c$, *or*
- $O\left(N_r(q)(n/N_r(q))^{\rho(r,c)} + N_{cr}(q)\right)$ *if* $c_q^* < c$.

Let $\mathcal{H}_k$ denote the LSH obtained concatenating k randomly and uniformly selected hashes from $\mathcal{H}$ and let $K = \lceil \rho \log_{1/p_2} n \rceil$. The data structure leverages a multi-level LSH: in each level $0 \leq k < K$, the input set is partitioned according to $O\left(p_1^{-k} \log k\right)$ hash functions in $\mathcal{H}_k$. For a given query q, we let $W_{q,k}$ be the cost for finding the near neighbor of q using the LSH at level k. Since $\Pr[h_k(q) = h_k(x)] = \Pr[h(q) = h(x)]^k$, the expected value of $W_{q,k}$ is:

$$\mathbf{E}[W_{q,k}] = p_1^{-k} \left(1 + \sum_{x \in R} \Pr[h(q) = h(x)]^k\right).$$

The value $k_x = \arg\min_{k \in [K]} W_{q,k}$ gives the best level to use for finding all near neighbors of q. The data structure computes an estimate $\tilde{W}_{q,k}$ of $W_{q,k}$ by summing the sizes of buckets where q collides at level k and then computing $\tilde{k}_x = \arg\min_{k \in [K]} \tilde{W}_{q,k}$. It is relevant to recall that the cost of removing x at level k'_x is upper bounded by the cost at level $\lceil \log(n/N(q, r))/\log(1/p_2) \rceil$ (without knowing the actual output size).

3 A constant-round algorithm for similarity join

Let $\mathcal{H}$ be an (r, cr, p_1, p_2) LSH family, let $\rho = \log p_1/\log p_2$, and let $\mathcal{H}_k$ denote the LSH family obtained by concatenating $k \geq 1$ copies of independent and identically distributed LSHs in $\mathcal{H}$.

At the high level, the algorithm constructs a multi-level LSH data structure as in [2], where the i-level uses i-concatenated LSHs and the bottom levels (i.e., below a given threshold κ) are removed. Then, the algorithm removes each input point $x \in R \cup S$ by searching in the buckets of LSHs at level k_x, which is a suitable value that reduces the communication cost for removing point x. We initially assume that all the k_x values are known, and we will later show that they can be computed (with a slight increase in the communication complexity) in one round.

Let $\kappa = \lceil (\rho/(1+\rho)) \log_{p_1^{-1}} p \rceil$. For each point $x \in R$, we define $k_x = \arg\min_{i \in [\kappa]} \mathbf{E}[W_{x,i}]$, that is:

$$k_x = \arg\min_{i \in [\kappa]} \left(p_1^{-i} \left(1 + \sum_{y \in S} Pr\left[h_i(x) = h_i(y) \right] \right) \right).$$

We recall that the value k_x defined in [2] is defined using $\kappa = \lceil \rho \log_{1/p_1} n \rceil$. The term k_y for each $y \in S$ is defined equivalently.

The algorithm consists of κ phases, each one requiring $O(1)$ rounds. These phases can be executed sequentially, resulting in an $O(\kappa)$-round algorithm; however, since there is no dependency among phases, they can be executed concurrently to give an $O(1)$-round algorithm. We assume the input to be evenly distributed among the p processors. An input point $x \in R \cup S$ is said *active* during the k_x-th phase, *passive* during the i-th phase for each $1 \leq i < k_x$, and *dead* during the i-th phase for each $i > k_x$. The i-th phase, with $i \in [\kappa]$, is organized as follows:

1. Choose $t_i = \Theta(p_1^{-i})$ hash functions $h_1^i, \ldots, h_{t_i}^i$ in $\mathcal{H}_k$ randomly and independently, and broadcast them to all p processors.
2. Verify if x, for each $x \in R \cup S$, is passive, active, or dead by comparing the index i with the value k_x.
3. For each point $x \in R \cup S$ and hash functions $h_1^i, \ldots, h_{t_i}^i$ in $\mathcal{H}_k$, generate a tuple with key $(i, j, h_j^i(x))$ and value x (including its status as active or passive).
4. Let $Q_{(i,j,\ell)}$ be the set of tuples with key (i, j, ℓ). Remove all tuples associated with a key (i, j, ℓ) that does not have an active point in $Q_{(i,j,\ell)}$. This step can be executed with a sort and a prefix-sum computation.
5. Using the equi-join algorithm from [12] (see Theorem 1) on the remaining tuples, generate all pairs (x, y) such that $x \in R$, $y \in S$ and at least one of x and y is active, and then output only near pairs (i.e., $d(x, y) \leq r$).

When the κ phases are run concurrently, each entry x is at the same time passive, active and dead. In other words, for each entry x we generate the hash values $(i, j, h_j^i(x)$ for each $1 \leq i \leq k_x$ and $1 \leq j < t_i$. Note that the κ equi-joins can be run as a single equi-join, with a slight improve of the load. We have the following theorem.

Theorem 4. *The above algorithm runs in $O(1)$ rounds and load:*

$$\tilde{O}\left(\sqrt{\left(\sum_{i=0}^{\kappa} \frac{\mathrm{OUT}_{r,i}}{p p_1^i} \right)} + \sqrt{\frac{\mathrm{OUT}_{cr}}{p}} + \frac{n}{p^{1/(1+\rho)}} \right).$$

Proof. The correctness of the algorithm follows from [2]. We now upper bound the cost of Step 4. Consider a point x in $R \cup S$, and let x be ℓ_x-dense. Denote with $N_r(x)$ and $N_{cr}(x)$ the number of near and cr-near points to x respectively. We observe that

$$\mathrm{OUT}_{r,i} = \Theta\left(\sum_{i\text{-dense x}} N_r(x) \right), \text{ and } \mathrm{OUT}_{cr} = \Theta\left(\sum_{x} N_{cr}(x) \right).$$

By construction, the total number of output pairs involving x is upper bounded by $W_{k_x,x} \leq W_{\ell_x,x}$. We now bound $W_{\ell_x,x}$; summing over these gives the total number of output pairs.

If $\ell_x = \kappa$, then $W_{\ell_x,x} \leq N_r(x)/p_1^{\kappa} + N_{cr}(x) + n(p_2/p_1)^{\kappa}$. We have $n(p_2/p_1)^{\kappa} = n/p_1^{\kappa(1-1/\rho)} = \Theta(n/pp_1^{2\kappa})$.

If $\ell_x < \kappa$, then $W_{\ell_x,x} \leq N_r(x)/p_1^{\ell_x} + N_{cr}(x) + n(p_2/p_1)^{\ell_x}$. Since x is ℓ_x-dense, $N_r(x)/p_1^{\ell_x} = \Theta(n(p_2/p_1)^{\ell_x})$. Thus, $W_{\ell_x,x} = O(N_r(x)/p_1^{\ell_x} + N_{cr}(x))$.

Summing over all x and invoking Theorem 1, Step 4 has load

$$\tilde{O}\left(\sqrt{\left(\sum_{i=0}^{\kappa} \frac{\mathrm{OUT}_{r,i}}{p_1^i} + \mathrm{OUT}_{cr} + \frac{n^2}{pp_1^{2\kappa}}\right)\frac{1}{p}}\right)$$

which reduces to the claimed bounds using Jensen's inequality.

We now consider the first three steps. We observe that the number of processors is at most n^{ϵ} for some constant $\epsilon > 0$, and hence prefix-sum, broadcast, sorting require constant rounds and load $O(I/p)$, where I is the input size. Therefore the first three steps require $O(1)$ rounds. Step 1 requires load $\tilde{O}\left(p_1^{-\kappa}/p\right)$. For Step 2, we observe that each input point is only copied once for each hash function; since there are $\tilde{O}\left(p_1^{-\kappa}\right)$ hash functions and each processor contains $O(n/p)$ points, the load is $\tilde{O}\left(np_1^{-\kappa}/p\right) = \frac{n}{p^{1/(1+\rho)}}$ (indeed the sorting step guarantee load balancing). Step 3 consists of a sorting and prefix sum on $\tilde{O}\left(np_1^{-\kappa}\right)$ points and hence its load is $\tilde{O}\left(np_1^{-\kappa}/p\right) = \frac{n}{p^{1/(1+\rho)}}$. Therefore, the first three steps meet the claimed bound and the theorem follows. □

The previous algorithm assumes that k_x is known for each input point. However, it is easy to see that the values can be computed in $O(1)$ rounds and load $\tilde{O}\left(\frac{n}{p^{1/(1+\rho)}}\right)$, which is negligible compared to the overall cost of the algorithm. Indeed, it suffices to generate all keys up to level κ, sort them and compute some prefix sums to estimate the expected costs $\mathbf{E}[W_{i,x}]$ as in [2].

4 Conclusion

We have seen how to improve the output sensitivity of the algorithm in [12] with a simple $O(1)$-round algorithm. A limit of our approach is the $n/p^{1/(1+\rho)}$ term in the load, which is due to the computation of k_x values (and to a non tight bound in the proof of Theorem 4). We conjecture that it is possible to reduce this term for some output densities by using approximations to the k_x values. One promising approach is the technique for the Braun-Blanquet similarity in [9], where k_x values are replaced with greedily-computed collision probabilities. However, such an approach seems to increase the number of rounds to $O(\log n)$.

Another interesting direction is an experimental evaluation of our approach. Analysis for specific inputs in [2] and experiments in [9] indicate that our recursive approach may give speedups beyond the worst-case theoretical analysis. It would be interesting to see the load incurred using our approach on practical datasets.

5 Acknowledgements

We would like to thank R. Pagh, J. Sivertsen, S. Singh, J. Augustine, and M. Daga for helpful discussions. We also thank the participants of the AlgoPARC Workshop on Parallel Algorithms and Data Structures, in part supported by the NSF grant no. 1745331. The authors were supported by the ERC grant agreement no. 614331, and by project SID2017 of the University of Padova. BARC, Basic Algorithms Research Copenhagen, is supported by VILLUM Foundation grant 16582.

References

[1] F. N. Afrati, M. R. Joglekar, C. M. Re, S. Salihoglu, and J. D. Ullman. GYM: A Multiround Distributed Join Algorithm. In *Proc 20th Int. Conf. on Database Theory (ICDT)*, volume 68 of *LIPIcs*, pages 4:1–4:18, 2017.

[2] T. D. Ahle, M. Aumüller, and R. Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *Proc. 28th Symp. on Discrete Algorithms (SODA)*, pages 239–256, 2017.

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proc. 32nd Conf. on Very Large Data Bases (VLDB)*, pages 918–929, 2006.

[4] N. Augsten and M. H. Bhlen. *Similarity Joins in Relational Database Systems.* Morgan & Claypool Publishers, 2013.

[5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proc. 27th Conf. on World Wide Web, (WWW)*, pages 131–140, 2007.

[6] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proc. 32nd Symp. on Principles of Database Systems (PODS)*, pages 273–284, 2013.

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

[8] M. Ceccarello and F. Silvestri. Experimental evaluation of multi-round matrix multiplication on mapreduce. In *Proc. 17th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 119–132, 2015.

[9] T. Christiani, R. Pagh, and J. Sivertsen. Scalable and robust set similarity join. In *Proc. 34th Int. Conf. on Data Engineering (ICDE)*, 2018.

[10] A. Driemel and F. Silvestri. Locality-Sensitive Hashing of Curves. In *Proc. 33rd Symp. on Computational Geometry (SoCG)*, volume 77 of *LIPIcs*, pages 37:1–37:16, 2017.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. 25th Conf. on Very Large Data Bases (VLDB)*, pages 518–529, 1999.

[12] X. Hu, Y. Tao, and K. Yi. Output-optimal parallel algorithms for similarity joins. In *Proc. 36th Symp. on Principles of Distributed Computing (PODS)*, 2017.

[13] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *Proc. 21st Symp. on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[14] R. Pagh, N. Pham, F. Silvestri, and M. Stöckel. I/O-efficient similarity join. In *Proc. 23rd European Symp. on Algorithms (ESA)*, pages 941–952, 2015.

[15] A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. Space-round tradeoffs for mapreduce computations. In *Proc. 26th Int. Conf. on Supercomputing (ICS)*, pages 235–244, 2012.

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

[17] R. Williams. On the difference between closest, furthest, and orthogonal pairs: Nearly-linear vs barely-subquadratic complexity. In *Proc. 29th Symp. on Discrete Algorithms (SODA)*, pages 1207–1215, 2018.