Dynamic Enumeration of Similarity Joins

Pankaj K. Agarwal
Duke University, Durham, NC, USA

Xiao Hu
Duke University, Durham, NC, USA

Stavros Sintos
University of Chicago, IL, USA

Jun Yang
Duke University, Durham, NC, USA

Abstract

This paper considers enumerating answers to *similarity-join* queries under dynamic updates: Given two sets of n points A, B in $\mathbb{R}^d$, a metric $\phi(\cdot)$, and a distance threshold $r > 0$, report all pairs of points $(a, b) \in A \times B$ with $\phi(a, b) \leq r$. Our goal is to store A, B into a dynamic data structure that, whenever asked, can enumerate all result pairs with worst-case *delay* guarantee, i.e., the time between enumerating two consecutive pairs is bounded. Furthermore, the data structure can be efficiently updated when a point is inserted into or deleted from A or B.

We propose several efficient data structures for answering similarity-join queries in low dimension. For exact enumeration of similarity join, we present near-linear-size data structures for ℓ_1, ℓ_∞ metrics with $\log^{O(1)} n$ update time and delay. We show that such a data structure is not feasible for the ℓ_2 metric for $d \geq 4$. For *approximate* enumeration of similarity join, where the distance threshold is a soft constraint, we obtain a unified linear-size data structure for ℓ_p metric, with $\log^{O(1)} n$ delay and update time. In high dimensions, we present an efficient data structure with worst-case delay-guarantee using *locality sensitive hashing* (LSH).

2012 ACM Subject Classification Theory of computation → Data structures and algorithms for data management

Keywords and phrases dynamic enumeration, similarity joins, worst-case delay guarantee

Digital Object Identifier 10.4230/LIPIcs.ICALP.2021.11

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2105.01818`

Funding This work has partially supported by NSF grants IIS-1814493 and CCF-2007556.

1 Introduction

There has been extensive work in many areas including theoretical computer science, computational geometry, and database systems on designing efficient dynamic data structures to store a set $\mathcal{O}$ of objects so that certain queries on $\mathcal{O}$ can be answered quickly and objects can be inserted into or deleted from $\mathcal{O}$ dynamically. A query $\mathcal{Q}$ is specified by a set of constraints and the goal is to report the subset $\mathcal{Q}(\mathcal{O}) \subseteq \mathcal{O}$ of objects that satisfy the constraints, the so-called *reporting* or *enumeration* queries. More generally, $\mathcal{Q}$ may be specified on k-tuples of objects in $\mathcal{O}$, and we return the subset of $\mathcal{O}^k$ that satisfy $\mathcal{Q}$. One may also ask to return certain statistics on $\mathcal{Q}(\mathcal{O})$ instead of $\mathcal{Q}(\mathcal{O})$ itself, but here we focus on enumeration queries. As an example, $\mathcal{O}$ is set of points in $\mathbb{R}^d$ and a query $\mathcal{Q}$ specifies a simple geometric region Δ (e.g., box, ball, simplex) and asks to return $\mathcal{O} \cap \Delta$, the so-called *range-reporting* problem. As another example, $\mathcal{O}$ is again a set of points in $\mathbb{R}^d$, and $\mathcal{Q}$ now specifies a value $r \geq 0$ and asks to return all pairs $(p, q) \in \mathcal{O} \times \mathcal{O}$ with $\|p - q\| \leq r$. Traditionally, the performance of a data structure has been measured by its size, the time needed to update the data structure when an object is inserted or deleted, and the *total time* spent in reporting $\mathscr{Q}(\mathscr{O})$. In some applications, especially in exploratory or interactive data analysis of large datasets, it is desirable to report $\mathscr{Q}(\mathscr{O})$ incrementally one result at a time so that users can start exploiting the first answers while waiting for the remaining ones. To offer guarantees on the regularity during the enumeration process, we consider an important complexity measure of such data structures as the maximum *delay* between the enumeration of two consecutive objects [11]. Formally speaking, *δ-delay enumeration* requires that the time between the start of the enumeration process to the first result, the time between any consecutive pair of results, and the time between the last result and the termination of the enumeration process should all be at most δ.

In this paper, we are interested in dynamic data structures for *(binary) similarity join* queries, which have numerous applications in data cleaning, data integration, collaborative filtering, etc. Given two sets of points A and B in $\mathbb{R}^d$, a metric $\phi(\cdot)$, and a distance threshold $r > 0$, the similarity join asks to report all pairs of $(a, b) \in A \times B$ with $\phi(a, b) \leq r$. Similarity joins have been studied extensively in the database and data mining literature [19, 33, 40, 43, 45], but it is still unclear how to enumerate similarity join results efficiently when the underlying data is updated. Our goal is to design a dynamic data structure that can be efficiently updated when an input point is inserted or deleted; and whenever an enumeration query is issued, all join results can be enumerated from it with *worst-case delay* guarantee.

1.1 Previous results

We briefly review the previous work on similarity join and related problems. See surveys [8, 10, 44] for more results.

Enumeration for Conjunctive Queries. Conjunctive queries are built upon natural join ($\bowtie$), which is a special case of similarity join with $r = 0$, i.e., two tuples can be joined if and only if they have the same value on the join attributes. Enumeration for conjunctive queries has been extensively studied in the static settings [11, 42, 16] for a long time. In 2017, two papers [14, 31] started to study dynamic enumeration for conjunctive query. Both obtained a dichotomy. First, a linear-size data structure that can be updated in $O(1)$ time while supporting $O(1)$-delay enumeration exists for a conjunctive query if and only if it is *q-hierarchical* (e.g., the degenerated natural join over two tables is q-hierarchical). However, for non-q-hierarchical queries with input size n, they showed a lower bound $\Omega(n^{\frac{1}{2}-\varepsilon})$ on the update time for any small constant $\varepsilon > 0$, if aiming at $O(1)$ delay. This result is very negative since q-hierarchical queries are a very restricted class; for example, the matrix multiplication query $\pi_{X,Z} R_1(X, Y) \bowtie R_2(Y, Z)$, where $\pi_{X,Y}$ denotes the projection on attributes X, Y, and the triangle join $R_1(X, Y) \bowtie R_2(Y, Z) \bowtie R_3(Z, X)$ are already non-q-hierarchical. Later, Kara et al. [34] designed optimal data structures supporting $O(\sqrt{n})$-time maintenance for some selected non-q-hierarchical queries such as the triangle queries. However, it is still unclear if a data structure of $O(\sqrt{n})$-time maintenance exists for a large class of queries. Some additional trade-off results have been obtained in [35, 46].

Range search. A widely studied problem related to similarity join is *range searching* [2, 3, 13, 47]: Preprocess a set A of points in $\mathbb{R}^d$ with a data structure so that for a query range γ (e.g., rectangle, ball, simplex), all points of $A \cap \gamma$ can be reported quickly. A particular instance of range searching, the so-called *fixed-radius-neighbor* searching, in which the range is a ball of fixed radius centered at query point is particularly relevant for similarity joins.

Table 1 Summary of Results: n is the input size; r is the distance threshold; d is the dimension of input points; $\rho \leq \frac{1}{(1+\varepsilon)^2} + o(1)$ is the quality of LSH family for the ℓ_2 metric. For ℓ_1, Hamming $\rho \leq \frac{1}{1+\varepsilon}$. $\widetilde{O}$ notation hides a $\log^{O(1)} n$-factor; for the results where d is constant the $O(1)$ exponent is at most linear on d, while for the high dimensional case the exponent is at most 3.

Enumeration	Metric	Properties	Data Structures		
			Space	Update	Delay
Exact	ℓ_1/ℓ_∞	r is fixed	$\widetilde{O}(n)$	$\widetilde{O}(1)$	$\widetilde{O}(1)$
	ℓ_2	r is fixed	$\widetilde{O}(n)$	$\widetilde{O}(n^{1-\frac{1}{d+1}})$	$\widetilde{O}(n^{1-\frac{1}{d+1}})$
ϵ-Approximate	ℓ_p	r is fixed	$O(n)$	$\widetilde{O}(\epsilon^{-d})$	$\widetilde{O}(\epsilon^{-d})$
		r is variable spread is poly(n)	$O(\varepsilon^{-d} n)$	$\widetilde{O}(\varepsilon^{-d})$	$O(1)$
	ℓ_1, ℓ_2, hamming	r is fixed high dimension	$\widetilde{O}(dn + n^{1+\rho})$	$\widetilde{O}(dn^{2\rho})$	$\widetilde{O}(dn^{2\rho})$

For a given metric ϕ, let $\mathscr{B}_\phi(x, r)$ be the ball of radius r centered at x. A similarity join between two sets A, B can be answered by querying A with ranges $\mathscr{B}_\phi(b, r)$ for all $b \in B$. Notwithstanding this close relationship between range searching and similarity join, the data structures for the former cannot be used for the latter: It is too expensive to query A with $\mathscr{B}_\phi(b, r)$ for every $b \in B$ whenever an enumeration query is issued, especially since many such range queries may return empty set, and it is not clear how to maintain the query results as the input set A changes dynamically.

Reporting neighbors. The problem of reporting neighbors is identical to our problem in the offline setting. In particular, given a set P of n points in $\mathbb{R}^d$ and a parameter r, the goal is to report all pairs of P within distance r. The algorithm proposed in [36] can be modified to solve the problem of reporting neighbors under the ℓ_∞ metric in $O(n + k)$ time, where k is the output size. Aiger et al. [7] proposed randomized algorithms for reporting neighbors using the ℓ_2 metric in $O((n + k) \log n)$ time, for constant d.

Scalable continuous query processing. There has been some work on scalable *continuous query* processing, especially in the context of data streams [21, 18, 49] and publish/subscribe [25], where the queries are standing queries and whenever a new data item arrives, the goal is to report all queries that are affected by the new item [6, 5]. In the context of similarity join, one can view A as the data stream and $\mathscr{B}_\phi(b, r)$ as standing queries, and we update the results of queries as new points in A arrive. There are, however, significant differences with similarity joins – arbitrary deletions are not handled; continuous queries do not need to return previously produced results; basing enumeration queries on a solution for continuous queries would require accessing previous results, which can be prohibitive if stored explicitly.

1.2 Our results

We present several dynamic data structures for enumerating similarity joins under different metrics. Table 1 summarizes our main results. It turns out that dynamic similarity join is hard for some metrics, e.g., ℓ_2. Therefore we also consider *approximate similarity join* where the distance threshold r is a soft constraint. Formally, given parameter $r, \varepsilon > 0$, the *ε-approximate similarity join* relaxes the distance threshold: (1) all pairs of $(a, b) \in A \times B$ with $\phi(a, b) \leq r$ should be returned; (2) no pair of $(a, b) \in A \times B$ with $\phi(a, b) > (1 + \varepsilon)r$ is returned; (3) some pairs of $(a, b) \in A \times B$ with $r < \phi(a, b) \leq (1 + \varepsilon)r$ may be returned. We classify our results in four broad categories:

Exact similarity join. Here we assume that d is constant and the distance threshold is fixed. Our first result (Section 2.1) is an $\widetilde{O}(1)$-size data structure for similarity join under the ℓ_1/ℓ_∞ metrics that can be updated in $\widetilde{O}(1)$ time whenever A or B is updated, and ensures $\widetilde{O}(1)$ delay during enumeration. Based on range trees [12, 23], the data structure stores the similarity join pairs *implicitly* so that they can be enumerated without probing every input point. We extend these ideas to construct a data structure for similarity join under the ℓ_2 metric (in Section 2.3) with $\widetilde{O}(n^{1-1/d})$ amortized update time while supporting $\widetilde{O}(n^{1-1/d})$-delay enumeration. Lower bounds on ball range searching [1, 20] rule out the possibility of a linear-size data structure with $\widetilde{O}(1)$ delay.

Approximate similarity join in low dimensions. Due to the negative result for ℓ_2 metric, we shift our attention to ε-approximate similarity join. We now allow the distance threshold to be part of the query, but the value of ε, the error parameter, is fixed. We present a simple linear-size data structure based on quad trees and the notion of well-separated pair decomposition, with $O(\epsilon^{-d})$ update time and $O(1)$ delay. If we fix the distance threshold, then the data structure can be further simplified and somewhat improved by replacing the quad tree with a simple uniform grid.

Approximate similarity join in high dimensions. So far we assumed d to be constant and the big O notation in some of the previous bounds hides a constant that is exponential in d. Our final result is an LSH-based [27] data structure for similarity joins in high dimensions. Two technical issues arise when enumerating join results from LSH: one is to ensure bounded delay because we do not want to enumerate false positive results identified by the hash functions, and the other is to remove duplicated results as one join result could be identified by multiple hash functions. For the ℓ_2 metric (the results can also be extended to ℓ_1 and Hamming metrics) we propose a data structure of $\widetilde{O}(nd + n^{1+\rho})$ size and $\widetilde{O}(dn^{2\rho})$ amortized update time that supports $(1+2\varepsilon)$-approximate enumeration with $\widetilde{O}(dn^{2\rho})$ delay with high probability, where $\rho \leq \frac{1}{(1+\varepsilon)^2} + o(1)$ is the quality of the LSH family. Alternatively, we present a data structure with $\widetilde{O}(dn^{\rho})$ amortized update time and $\widetilde{O}(dn^{3\rho})$ delay. Our data structure can be extended to the case when the distance threshold r is variable. If we allow worse approximation error we can improve the results for the Hamming distance. Finally, we show a lower bound by relating similarity join to the *approximate nearest neighbor* query.

We also consider similarity join beyond binary joins.

Triangle similarity join in low dimensions. Given three sets of points A, B, S in $\mathbb{R}^d$, a metric $\phi(\cdot)$, and a distance threshold $r > 0$, the *triangle similarity join* asks to report the set of all triples of $(a, b, s) \in A \times B \times S$ with $\phi(a, b) \leq r, \phi(a, s) \leq r, \phi(b, s) \leq r$. The *$\varepsilon$-approximate triangle similarity join* can be defined similarly by taking the distance threshold r as a soft constraint. In the full version [4], we extend our data structures to approximate *triangle similarity join* by paying an extra factor of $\log^{O(1)} n$ in the performance.

High-level framework. All our data structures rely on the following common framework. We model the (binary) similarity join as a bipartite graph $G' = (A \cup B, E)$, where an edge $(a, b) \in E$ if and only if $\phi(a, b) \leq r$. A naive solution by maintaining all edges of G' explicitly leads to a data structure of $\Theta(n^2)$ size that can be updated in $\Theta(n)$ time while supporting $O(1)$-delay enumeration. To obtain a data structure with poly-logarithmic update time and delay enumeration, we find a compact representation of G' with a set $\mathscr{F} = \{(A_1, B_1), (A_2, B_2), \ldots, (A_u, B_u)\}$ of edge-disjoint bi-cliques such that (i) $A_i \subseteq A$,

$B_i \subseteq B$ for any i, (ii) $E = \bigcup_{i=1}^{u} A_i \times B_i$, and (iii) $(A_i \times B_i) \cap (A_j \times B_j) = \emptyset$ for any $i \neq j$. We represent $\mathscr{F}$ using a tripartite graph $\mathscr{G} = (A \cup B \cup C, E_1 \cup E_2)$ where $C = \{c_1, \ldots, c_u\}$ has a node for each bi-clique in $\mathscr{F}$ and for every $i \leq u$, we have the edges $(a_j, c_i) \in E_1$ for all $a_j \in A_i$ and $(b_k, c_i) \in E_1$ for all $b_k \in B_i$. We *cannot* afford to maintain E_1 and E_2 explicitly. Instead, we store some auxiliary information for each c_i and use geometric data structures to recover the edges incident to a vertex $c_i \in C$. We also use data structures to maintain the set C and the auxiliary information dynamically as A and B are being updated. We will not refer to this framework explicitly but it provides the intuition behind all our data structures. Section 2 describes the data structures to support this framework for exact similarity join, and Section 3 presents simpler, faster data structures for approximate similarity join. Both Sections 2 and 3 assume d to be constant. Section 4 describes the data structure for approximate similarity join when d is not constant.

2 Exact Similarity Join

In this section, we describe the data structure for exact similarity joins under the $\ell_\infty, \ell_1, \ell_2$ metrics, assuming d is constant. We first describe the data structure for the ℓ_∞ metric. We show that similarity join under the ℓ_1 metric in $\mathbb{R}^d$ can be reduced to that under the ℓ_∞ metric in $\mathbb{R}^{d+1}$. Finally, we describe the data structure for the ℓ_2 metric. Throughout this section, the threshold r is fixed, which is assumed to be 1 without loss of generality.

2.1 Similarity join under ℓ_∞ metric

Let A and B be two point sets in $\mathbb{R}^d$ with $|A| + |B| = n$. For a point $p \in \mathbb{R}^d$, let $\mathscr{B}(p) = \{x \in \mathbb{R}^d \mid \|p - x\|_\infty \leq 1\}$ be the hypercube of side length 2. We wish to enumerate pairs $(a, b) \in A \times B$ such that $a \in \mathscr{B}(b)$.

Data structure. We build a d-dimensional dynamic range tree $\mathscr{T}$ on the points in A. For $d = 1$, the range tree on A is a balanced binary search tree $\mathscr{T}$ of $O(\log n)$ height. The points of A are stored at the leaves of $\mathscr{T}$ in increasing order, while each internal node v stores the smallest and the largest values, α_v^- and α_v^+, respectively, contained in its subtree. The node v is associated with an interval $I_v = [\alpha_v^-, \alpha_v^+]$ and the subset $A_v = I_v \cap A$. For $d > 1$, $\mathscr{T}$ is constructed recursively: We build a 1D range tree $\mathscr{T}_d$ on the x_d-coordinates of points in A. Next, for each node $v \in \mathscr{T}_d$, we recursively construct a $(d-1)$-dimensional range tree $\mathscr{T}_v$ on A_v^*, which is defined as the projection of A_v onto the hyperplane $x_d = 0$, and attach $\mathscr{T}_v$ to v as its secondary tree. The size of $\mathscr{T}$ in $\mathbb{R}^d$ is $O(n \log^{d-1} n)$ and it can be constructed in $O(n \log^d n)$ time. See [23] for details.

For a node v at a level-i tree, let $p(v)$ denote its parents in that tree. If v is the root of that tree, $p(v)$ is undefined. For each node u of the d-th level of $\mathscr{T}$, we associate a d-tuple $\pi(u) = \langle u_1, u_2, \ldots, u_d = u \rangle$, where u_i is the node at the i-th level tree of $\mathscr{T}$ to which the level-$(i+1)$ tree containing u_{i+1} is connected. We associate the rectangle $\square_u = \prod_{j=1}^{d} I_{u_j}$ with the node u. For a rectangle $\rho = \prod_{i=1}^{d} \delta_i$, a d-level node u is called a *canonical node* if for every $i \in [1, d]$, $I_{u_i} \subseteq \delta_i$ and $I_{p(u_i)} \not\subseteq \delta_i$. For any rectangle ρ, there are $O(\log^d n)$ canonical nodes in $\mathscr{T}$, denoted by $\mathscr{N}(\rho)$, and they can be computed in $O(\log^d n)$ time [23]. $\mathscr{T}$ can be maintained dynamically, as points are inserted into A or deleted from A using the standard partial-reconstruction method, which periodically reconstructs various bottom subtrees. The amortized time is $O(\log^d n)$; see [39] for details.

We query $\mathscr{T}$ with $\mathscr{B}(b)$ for all $b \in B$ and compute $\mathscr{N}(b) := \mathscr{N}(\mathscr{B}(b))$ the sets of its canonical nodes. For each level-d tree node u of $\mathscr{T}$, let $B_u = \{b \in B \mid u \in \mathscr{N}(b)\}$. We have $\sum_u |B_u| = O(n \log^d n)$. By construction, for all pairs $(a, b) \in A_u \times B_u$, $\|a - b\|_\infty \leq 1$, so

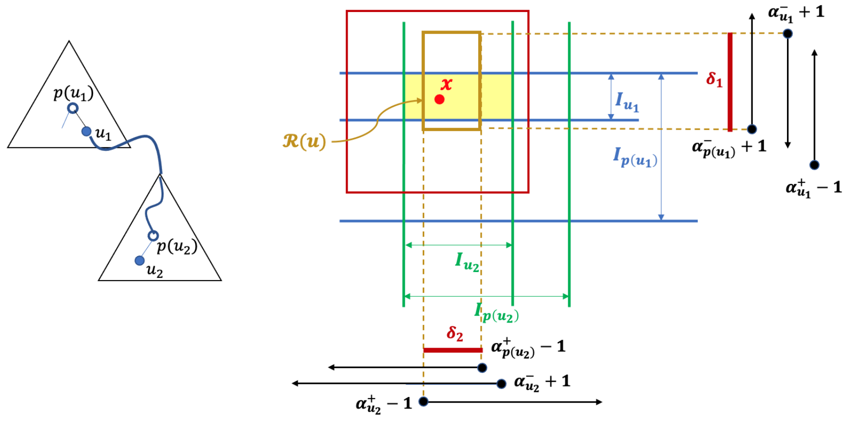


Figure 1 Left: Two levels of the range tree. Right: Definition of $\mathscr{R}(u)$.

(A_u, B_u) is a bi-clique of join results. We call u *active* if both $A_u, B_u \neq \emptyset$. A naive approach for reporting join results is to maintain A_u, B_u for every d-level node u of $\mathscr{T}$ as well as the set $\mathscr{C}$ of all active nodes. Whenever an enumerate query is issued, we traverse $\mathscr{C}$ and return $A_u \times B_u$ for all $u \in \mathscr{C}$ (referring to the tripartite-graph framework mentioned in Introduction, C is the set of all level-d nodes of $\mathscr{T}$). The difficulty with this approach is that when A changes and $\mathscr{T}$ is updated, some d-level nodes change and we have to construct B_u for each new level-d node $u \in \mathscr{T}$. It is too expensive to scan the entire B at each update. Furthermore, although the average size of B_u is small, it can be very large for a particular u and this node may appear and disappear several times. So we need a different approach. The following lemma is the key observation.

▶ **Lemma 1.** *Let u be a level-d node, and let $\pi(u) = \langle u_1, \ldots, u_d = u\rangle$. Then there is a d-dimensional rectangle $\mathscr{R}(u) = \prod_{i=1}^{d} \delta_i$, where the endpoints of δ_i, for $i \in [1, d]$, are defined by the endpoints of I_{u_i} and $I_{p(u_i)}$, such that for any $x \in \mathbb{R}^d$, $u \in \mathscr{N}(x)$ if and only if $x \in \mathscr{R}(u)$. Given u_i's and $p(u_i)$'s, $\mathscr{R}(u)$ can be constructed in $O(1)$ time.*

Proof. Notice that $\mathscr{B}(x)$ is the hypercube of side length 2 and center x. Let $I_{u_i} = [\alpha^-_{u_i}, \alpha^+_{u_i}]$ for any u_i and $i \in [1, d]$. Recall that $u \in \mathscr{N}(x)$ if and only if for each $i \in [1, d]$,

$$I_{u_i} \subseteq [x_i - 1, x_i + 1] \text{ and } I_{p(u_i)} \not\subseteq [x_i - 1, x_i + 1], \quad (*)$$

Fix a value of i. From the construction of a range tree either $\alpha^-_{u_i} = \alpha^-_{p(u_i)}$ or $\alpha^+_{u_i} = \alpha^+_{p(u_i)}$. Without loss of generality, assume $\alpha^-_{u_i} = \alpha^-_{p(u_i)}$; the other case is symmetric. Then $(*)$ can be written as: $x_i \le \alpha^-_{u_i} + 1$ and $\alpha^+_{u_i} - 1 \le x_i < \alpha^+_{p(u_i)} - 1$. Therefore x_i has to satisfy three 1D linear constraints. The feasible region of these constraints is an interval δ_i and $x_i \in \delta_i$ (see also Figure 1). Hence, u is a canonical node of $\mathscr{B}(x)$ if and only if for all $i \in [1, d]$, $x_i \in \delta_i$. In other words, $x = (x_1, \ldots, x_d) \in \prod_{i=1}^{d} \delta_i := \mathscr{R}(u)$. The endpoints of δ_i are the endpoints of I_{u_i} or $I_{p(u_i)}$. In order to construct $\mathscr{R}(u)$, we only need the intervals I_{u_i} and $I_{p(u_i)}$ for each $i \in [1, d]$, so it can be constructed in $O(d) = O(1)$ time. ◀

In view of Lemma 1, we proceed as follows. We build a dynamic range tree $\mathscr{Z}$ on B. Furthermore, we augment the range tree $\mathscr{T}$ on A as follows. For each level-d node $u \in \mathscr{T}$, we compute and store $\mathscr{R}(u)$ and $\beta_u = |B_u|$. By construction, $|A_u| \ge 1$ for all u. We also store a pointer at u to the leftmost leaf of the subtree of $\mathscr{T}$ rooted at u, and we thread all the leaves of a d-level tree so that for a node u, A_u can be reported in $O(|A_u|)$ time. Updating these pointers as $\mathscr{T}$ is updated is straightforward. Whenever a new node u of $\mathscr{T}$ is constructed, we query $\mathscr{Z}$ with $\mathscr{R}(u)$ to compute β_u. Finally, we store $\mathscr{C}$, the set of all active nodes of $\mathscr{T}$, in a red-black tree so that a node can be inserted or deleted in $O(\log n)$ time. The total size of the data structure is $O(n \log^{d-1} n)$, and it can be constructed in $O(n \log^d n)$ time.

Update and Enumerate. Updating A is straightforward. We update $\mathscr{T}$, query $\mathscr{Z}$ with $\mathscr{R}(u)$, for all newly created d-level nodes u in $\mathscr{T}$ to compute β_u, and update $\mathscr{C}$ to delete all active nodes that are no longer in $\mathscr{T}$ and to insert new active nodes. Since the amortized time to update $\mathscr{T}$ as a point is inserted or deleted is $O(\log^d n)$, the amortized update time of a point in A is $O(\log^{2d} n)$ – we spend $O(\log^d n)$ time to compute β_u for each of the $O(\log^d n)$ newly created nodes. If a point b is inserted (resp. deleted) in B, we update $\mathscr{Z}$ and query $\mathscr{T}$ with $\mathscr{B}(b)$. For all canonical nodes u in $\mathscr{N}(b)$, we increment (resp. decrement) b_u. If u becomes active (resp. inactive), we insert (resp. delete) u in $\mathscr{C}$ in $O(\log n)$ time. The amortized update time for b is $O(\log^{d+1} n)$.

Finally, to enumerate the pairs in join results, we traverse the active nodes $\mathscr{C}$ and for each $u \in \mathscr{C}$, we first query $\mathscr{Z}$ with $\mathscr{R}(u)$ to recover B_u. Recall that B_u is reported as a set of $O(\log^d n)$ canonical nodes of $\mathscr{Z}$ whose leaves contain the points of B_u. We simultaneously traverse the leaves of the subtree of $\mathscr{T}$ rooted at u to compute A_u and report $A_u \times B_u$. The traversals can be performed in $O(\log^d n)$ maximum delay. Putting everything together, we obtain:

▶ **Theorem 2.** *Let A, B be two sets of points in $\mathbb{R}^d$, where $d \geq 1$ is a constant, with $|A| + |B| = n$. A data structure of $\widetilde{O}(n)$ size can be built in $\widetilde{O}(n)$ time and updated in $\widetilde{O}(1)$ amortized time, while supporting $\widetilde{O}(1)$-delay enumeration of similarity join under ℓ_∞ metric.*

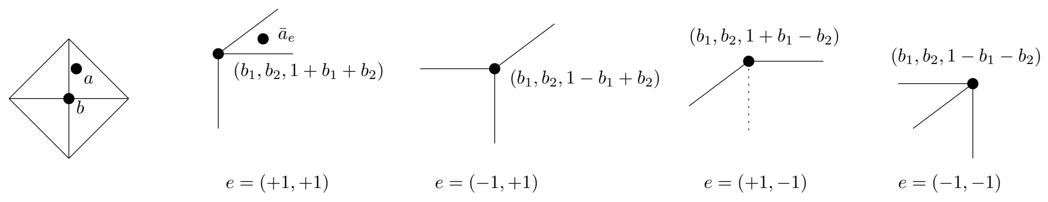
2.2 Similarity join under ℓ_1 metric

For $d \leq 2$ it is straightforward to reduce similarity join under ℓ_1 metric to ℓ_∞ metric. For $d = 1$, ℓ_1 metric is obviously equivalent to the ℓ_∞ metric. For $d = 2$, notice that the ℓ_1 ball is a diamond, while the ℓ_∞ ball is a square. Hence, given an instance of the similarity join under the ℓ_1 metric we can rotate $A \cup B$ by 45 degrees to create an equivalent instance of the similarity join problem under the ℓ_∞ metric.

Next, we focus on $d \geq 3$. The data structure we proposed in Section 2.1 for the ℓ_∞ norm can be straightforwardly extended to the *rectangle-containment* problem in which for each $b \in B$, $\mathscr{B}(b)$ is an arbitrary axis-aligned hyper-rectangle with center b, and the goal is to report all $(a, b) \in A \times B$ such that $a \in \mathscr{B}(b)$. Lemma 1 can be extended so that $\mathscr{R}(u)$ is a $2d$-dimensional rectangle. Overall, Theorem 2 remains the same assuming $\mathscr{B}(b)$ are hyper-rectangles (and not hypercubes).

Given an instance of similarity join under ℓ_1 metric in $\mathbb{R}^d$, we next show how to reduce it to 2^d $(d+1)$-dimensional rectangle-containment problems. As above, assume $r = 1$, so our goal is to report all pairs $a = (a_1, \ldots, a_d) \in A$, $b = (b_1, \ldots, b_d) \in B$ such that $\sum_{i=1}^d |a_i - b_i| \leq 1$.

Let $E = \{-1, +1\}^d$ be the set of all 2^d vectors in $\mathbb{R}^d$ with coordinates either 1 or -1. For each vector $e \in E$, we construct an instance of the rectangle-containment problem. For each $e = (e_1, \ldots, e_d) \in E$, we map each point $a = (a_1, \ldots, a_d) \in A$ to a point $\bar{a}_e = (a_1, \ldots, a_d, \sum_{i=1}^d e_i a_i) \in \mathbb{R}^{d+1}$. Let $\bar{A}_e = \{\bar{a}_e \mid a \in A\}$. For each point $b = (b_1, \ldots, b_d) \in B$, we construct the axis-align rectangle $\bar{b}_e = \prod_{i=1}^{d+1} b_e^{(i)}$ in $\mathbb{R}^{d+1}$, where $b_e^{(i)}$ is the interval $[b_i, \infty)$ if $e_i = 1$ and $(-\infty, b_i]$ if $e_i = -1$ for each $i = 1, \ldots, d$, and $b_e^{(d+1)} = (-\infty, 1 + \sum_{i=1}^d e_i b_i]$. Let $\bar{B}_e = \{\bar{b}_e \mid b \in B\}$. See Figure 2.



a
b
$\bar{a}_e$
$(b_1, b_2, 1 + b_1 + b_2)$
$(b_1, b_2, 1 - b_1 + b_2)$
$(b_1, b_2, 1 + b_1 - b_2)$
$(b_1, b_2, 1 - b_1 - b_2)$
$e = (+1, +1)$
$e = (-1, +1)$
$e = (+1, -1)$
$e = (-1, -1)$

Figure 2 An illustration of mapping each b to rectangles.

For each $e \in E$, we construct the dynamic data structure for $\bar{A}_e, \bar{B}_e$. Whenever A or B is updated, we update all 2^d rectangle-containment data structures. A similarity join enumeration query on A, B is answered by enumerating containment pairs $(\bar{A}_e, \bar{B}_e)$ for $e \in E$. If a pair $(\bar{a}_e, \bar{b}_e)$ is reported, we report (a, b). The update time and delay are $\tilde{O}(1)$. The correctness of the algorithm follows from the following lemma. Let $\mathrm{sgn}(x) = +1$ if $x \geq 0$ and -1 otherwise.

▶ **Lemma 3.** *Let $a = (a_1, \dots, a_d) \in A$, $b = (b_1, \dots, b_d) \in B$ be an arbitrary pair of points. Let $e^* = (e_1^*, \dots, e_d^*)$ where $e_i^* = \mathrm{sgn}(a_i - b_i)$ for $1 \leq i \leq d$. Then $\bar{a}_e \notin \bar{b}_e$ for all $e \in E \setminus \{e^*\}$. Furthermore, $\bar{a}_{e^*} \in \bar{b}_{e^*}$ if and only if $\|a - b\|_1 \leq 1$.*

Proof. First, we note that for any $e \in E \setminus \{e^*\}$, there must exist some i such that $e_i \neq e_i^*$. Without loss of generality, assume $e_j = 1$ when $a_j < b_j$. By the definition of $\bar{a}_e, \bar{b}_e$, $a_j \notin [b_j, \infty)$, thus $\bar{a}_e \notin \bar{b}_e$. Next, we show that $\bar{a}_{e^*} \in \bar{b}_{e^*}$ if and only if $\|a - b\|_1 \leq 1$. On one hand, we assume $\bar{a}_{e^*} \in \bar{b}_{e^*}$. By definition, $\sum_{i=1}^d e_i^* a_i$ lies in the interval associated with $b_{e^*}^{d+1}$, i.e., $\sum_{i=1}^d e_i^* a_i \leq 1 + \sum_{i=1}^d e_i^* b_i$, or $\sum_{i=1}^d e_i^*(a_i - b_i) \leq 1$. Implied by the fact that $\|a - b\|_1 = \sum_{i=1}^d e_i^*(a_i - b_i)$, we have $\|a - b\|_1 \leq 1$. On the other hand, assume $\|a - b\|_1 \leq 1$. Similarly, we have $\|a - b\|_1 = \sum_{i=1}^d e_i^*(a_i - b_i) \leq 1 \Leftrightarrow \sum_{i=1}^d e_i^* a_i \leq 1 + \sum_{i=1}^d e_i^* b_i$, or $\sum_{i=1}^d e_i^* a_i \in (-\infty, 1 + \sum_{i=1}^d e_i^* b_i]$. Moreover, for any $i \in \{1, \dots, d\}$, we have: (1) if $e_i^* = 1$, $a_i \geq b_i$, i.e., $a_i \in [b_i, \infty)$; (2) if $e_i^* = -1$, $a_i \leq b_i$, i.e., $a_i \in (-\infty, b_i]$. Hence, $\bar{a}_{e^*} \in \bar{b}_{e^*}$. ◀

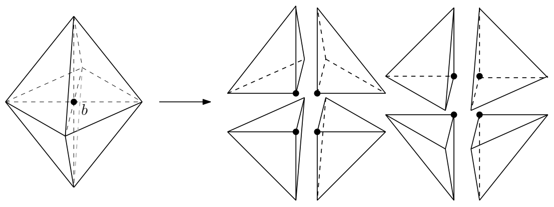


Figure 3 An illustration of ℓ_1 ball in $\mathbb{R}^3$. It is decomposed to $2^d = 8$ types of simplices.

▶ Remark. Roughly speaking, we partition the ℓ_1-ball centered at 0 into 2^d simplices $\Delta_1, \dots, \Delta_{2^d}$ (see Figure 3) and build a separate data structure for each simplex Δ_i. Namely, let $\mathscr{B}_i = \{b + \Delta_i \mid b \in B\}$ and we report all pairs $(a, b) \in A \times B$ such that $a \in b + \Delta_i$. If $\|a - b\|_1 \leq 1$ then a lies in exactly one simplex $b \in \Delta_i$. We map each simplex to a rectangle in $\mathbb{R}^{d+1}$ and use the previous data structure.

Using Theorem 2, we obtain:

▶ **Theorem 4.** *Let A, B be two sets of points in $\mathbb{R}^d$, where $d \geq 1$ is a constant, with $|A| + |B| = n$. A data structure of $\tilde{O}(n)$ size can be built in $\tilde{O}(n)$ time and updated in $\tilde{O}(1)$ amortized time, while supporting $\tilde{O}(1)$-delay enumeration of similarity join under ℓ_1 metric.*

2.3 Similarity join under ℓ_2 metric

In this section, we consider the similarity join between two point sets A and B in $\mathbb{R}^d$ under the ℓ_2 metric.

Reduction to halfspace containment. We use the *lifting transformation* [23] to convert an instance of the similarity join problem under ℓ_2 metric to the halfspace-containment problem in $\mathbb{R}^{d+1}$. For any two points $a = (a_1, \dots, a_d) \in A$ and $b = (b_1, \dots, b_d) \in B$, $\|a - b\|_2 \leq 1$ if and only if $(a_1 - b_1)^2 + \ldots + (a_d - b_d)^2 \leq 1$, or a lies in the unit sphere centered at b. The above condition can be rewritten as

$$a_1^2 + b_1^2 + \cdots + a_d^2 + b_d^2 - 2a_1b_1 - \cdots - 2a_db_d - 1 \geq 0.$$

We map the point a to a point $a' = (a_1, \dots, a_d, a_1^2 + \cdots + a_d^2)$ in $\mathbb{R}^{d+1}$ and the point b to a halfspace b' in $\mathbb{R}^{d+1}$ defined as

$$b' : -2b_1z_1 - \cdots - 2b_dz_d + z_{d+1} + b_1^2 + \cdots + b_d^2 - 1 \geq 0.$$

Note that $\|a - b\|_2 \leq 1$ if and only if $a' \in b'$. Set $A' = \{a' \mid a \in A\}$ and $B' = \{b' \mid b \in B\}$. Thus, in the following, we study the halfspace-containment problem, where given a set of points A' and a set of halfspaces B' we construct a dynamic data structure that reports all pairs $(a \in A', b \in B')$, such that a belongs in the halfspace b, with delay guarantee.

Partition tree. A partition tree on a set P of points in $\mathbb{R}^d$ [17, 37, 48] is a tree data structure formed by recursively partitioning a set into subsets. Each point is stored in exactly one leaf and each leaf usually contains a constant number of points. Each node u of the tree is associated with a simplex Δ_u and the subset $P_u = P \cap \Delta_u$; the subtree rooted at u is a partition tree of P_u. We assume that the simplices associated with the children of a node u are pairwise disjoint and lie inside Δ_u, as in [17]. In general, the degree of a node is allowed to be non-constant. Given a query simplex Δ, a partition tree finds a set of $O(n^{1-1/d})$ *canonical* nodes whose cells contain the points of $P \cap \Delta$. Roughly speaking, a node u is a canonical node for Δ if $\Delta_u \subset \Delta$ and $\Delta_{p(u)} \not\subseteq \Delta$. A simplex counting (resp. reporting) query can be answered in $O(n^{1-1/d})$ (resp. $O(n^{1-1/d} + k)$) time using a partition tree. Chan [17] proposed a randomized algorithm for constructing a linear size partition tree with constant degree, that runs in $O(n \log n)$ time and it has $O(n^{1-1/d})$ query time with high probability.

Data structure. For simplicity, with slight abuse of notation, let A be a set of points in $\mathbb{R}^d$ and B a set of halfspaces in $\mathbb{R}^d$ each lying below the hyperplane bounding it, and our goal is to build a dynamic data structure for halfspace-containment join on A, B. The overall structure of the data structure is the same as for rectangle containment described in Section 2.1, so we simply highlight the difference.

Instead of constructing a range tree, we construct a dynamic partition tree $\mathscr{T}_A$ for A so that the points of A lying in a halfspace can be represented as the union of $O(n^{1-1/d})$ canonical subsets. For a halfplane bounding a halfspace $b \in B$, let $\bar{b}$ denote its dual point in $\mathbb{R}^d$ (see [23] for the definition of duality transform). Note that a point a lies in b if and only if the dual point $\bar{b}$ lies in the halfspace lying below the hyperplane dual to a. Set $\bar{B} = \{\bar{b} \mid b \in B\}$. We construct a multi-level dynamic partition tree on $\bar{B}$, so that for a pair of simplices Δ_1 and Δ_2, it returns the number of halfspaces of B that satisfy the following two conditions: (i) $\Delta_1 \subseteq b$ and (ii) $\Delta_2 \cap \partial b \neq \emptyset$, where ∂b is the hyperplane boundary defined by the halfspace b. This data structure uses $O(n)$ space, can be constructed in $\widetilde{O}(n)$ time, and answers a query in $\widetilde{O}(n^{1-1/d})$ time.

For each node $u \in \mathcal{T}_A$, we issue a counting query to $\mathcal{T}_B$ and get the number of halfspaces in B that have u as a canonical node. Hence, $\mathcal{T}_A$ can be built in $\widetilde{O}(n^{2-1/d})$ time. For a node u, $\mu_A(u)$ can be computed in $O(1)$ time by storing A_u at each node $u \in \mathcal{T}_A$. Recall that $\mu_B(u)$ is the number of halfspaces b of B for which u is a canonical node, i.e., $\Delta_u \subseteq b$ and $\Delta_{p(u)} \cap \partial b \neq \emptyset$, where $p(u)$ is the parent of u. Using $\mathcal{T}_B$, $\mu_B(u)$ can be computed in $\widetilde{O}(n^{1-1/d})$ time.

Update and enumeration. The update procedure is the same that in Section 2.1, however the query time now on $\mathcal{T}_A$ or $\mathcal{T}_B$ is $\widetilde{O}(n^{1-\frac{1}{d}})$ so the amortized update time is $\widetilde{O}(n^{1-\frac{1}{d}})$. The enumeration query is also the same as in Section 2.1 but a reporting query in $\mathcal{T}_B$ takes $\widetilde{O}(n^{1-\frac{1}{d}} + k)$ time (and it has delay at most $\widetilde{O}(n^{1-\frac{1}{d}})$), so the overall delay is $\widetilde{O}(n^{1-\frac{1}{d}})$.

▶ **Theorem 5.** *Let A be a set of points and B be a set of half-spaces in $\mathbb{R}^d$ with $|A|+|B| = n$. A data structure of $\widetilde{O}(n)$ size can be built in $\widetilde{O}(n^{2-\frac{1}{d}})$ time and updated in $\widetilde{O}(n^{1-\frac{1}{d}})$ amortized time while supporting $\widetilde{O}(n^{1-\frac{1}{d}})$-delay enumeration of halfspace-containment query.*

Using Theorem 5 and the lifting transformation described at the beginning of this section we conclude with Corollary 6.

▶ **Corollary 6.** *Let A, B be two sets of points in $\mathbb{R}^d$, where $d \geq 1$ is a constant, with $|A| + |B| = n$. A data structure of $\widetilde{O}(n)$ size can be constructed in $\widetilde{O}(n^{2-\frac{1}{d+1}})$ time and updated in $\widetilde{O}(n^{1-\frac{1}{d+1}})$ amortized time, while supporting $\widetilde{O}(n^{1-\frac{1}{d+1}})$-delay enumeration of similarity join under the ℓ_2 metric.*

Lower bound. We show a lower bound for the similarity join in the pointer-machine model under the ℓ_2 metric based on the hardness of unit sphere reporting problem. Let P be a set of n points in $\mathbb{R}^d$ for $d > 3$. The unit-sphere reporting problem asks for a data structure on the points in P, such that given any unit-sphere b report all points of $P \cap b$. If the space is $\widetilde{O}(n)$, it is not possible to get a data structure for answering unit-sphere reporting queries in $\widetilde{O}(k + 1)$ time in the pointer-machine model, where k is the output size for $d \geq 4$ [1].

For any instance of sphere reporting problem, we construct an instance of similarity join over two sets, with $A = \emptyset$, $B = P$, and $r = 1$. Given a query unit-sphere of center q, we insert point q in A, issue an enumeration query, and then remove q from A. All results enumerated (if any) are the results of the sphere reporting problem. If there exists a data structure for enumerating similarity join under ℓ_2 metric using $\widetilde{O}(n)$ space, with $\widetilde{O}(1)$ update time and $\widetilde{O}(1)$ delay, we would break the barrier.

▶ **Theorem 7.** *Let A, B be two sets of points in $\mathbb{R}^d$ for $d > 3$, with $|A| + |B| = n$. If using $\widetilde{O}(n)$ space, there is no data structure under the pointer-machine model that can be updated in $\widetilde{O}(1)$ time, while supporting $\widetilde{O}(1)$-delay enumeration of similarity join under the ℓ_2 metric.*

3 Approximate Enumeration

In this section we propose a dynamic data structure for answering approximate similarity-join queries under any ℓ_p metric. For simplicity, we use the ℓ_2 norm to illustrate the main idea and assume $\phi(a, b) = ||a - b||_2$. Recall that all pairs of $(a, b) \in A \times B$ with $\phi(a, b) \leq r$ must be reported, along with (potentially) some pairs of (a', b') with $\phi(a', b') \leq (1 + \varepsilon)r$, but no pair (a, b) with $\phi(a, b) > (1 + \varepsilon)r$ is reported.

We will start with the setting where the distance threshold r is not fixed and specified as part of a query, and then move to a simpler scenario where r is fixed.

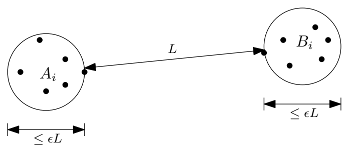


Figure 4 An example pair of ε-WSPD.

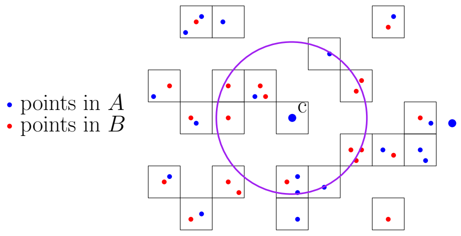


Figure 5 An example of active cell c in the grid.

3.1 Variable Similarity Threshold

We describe the data structure when r is part of the query. In this subsection we assume that the spread of $A \cup B$ is polynomially bounded, i.e., $sp(A \cup B) = \frac{\max_{p,q \in A\cup B} \phi(p,q)}{\min_{p\neq q \in A\cup B} \phi(p,q)} = n^{O(1)}$. We use a quad tree and well-separated pair decomposition (WSPD) for our data structure. We describe them briefly here and refer the reader to [28, 41] for details.

Quad tree and WSPD. A d-dimensional quad tree over a point set P is a tree data structure $\mathscr{T}$ in which each node u is associated with a hypercube $\square_u$ in $\mathbb{R}^d$, called a *cell*, and each internal node has 2^d children. The root is associated with a hypercube containing P. For a node u, let $P_u = P \cap \square_u$. A node u is a leaf if $|P_u| \leq 1$. The tree recursively subdivides the space into 2^d congruent hypercubes until a box contains at most one point from P. If $sp(P) = n^{O(1)}$, the height of $\mathscr{T}$ is $O(\log n)$.

Given two point sets $A, B \subset \mathbb{R}^d$, with $|A| + |B| = n$, and a parameter $0 < \varepsilon < \frac{1}{2}$, a family of pairs $\mathscr{W} = \{(A_1, B_1), (A_2, B_2), \cdots, (A_s, B_s)\}$ is an ε-WSPD if the following conditions hold: (1) for any $i \leq s$, $A_i \subseteq A$, $B_i \subseteq B$ (2) for each pair of points $(a, b) \in A \times B$, there exists a unique pair $(A_j, B_j) \in \mathscr{W}$ such that $a \in A_j$ and $b \in B_j$ (3) for any $i \leq s$, $\max\{\mathrm{diam}(A_i), \mathrm{diam}(B_i)\} \leq \varepsilon \cdot \phi(A_i, B_i)$, where $\mathrm{diam}(X) = \max_{x,y\in X} \phi(x, y)$ and $\phi(X, Y) = \min_{x\in X, y\in Y} \phi(x, y)$ (see Figure 4). As shown in [28, 30] if $sp(A \cup B) = n^{O(1)}$, a quad tree T on $A \cup B$ can be used to construct, in time $O(n \log n + \varepsilon^{-d} n)$, a WSPD $\mathscr{W}$ of size $O(\varepsilon^{-d} n)$ such that each pair $(A_i, B_i) \in \mathscr{W}$ is associated with pair of cells $(\square_i, \boxplus_i)$ in $\mathscr{T}$ where $A_i = A \cap \square_i$ and $B_i = B \cap \boxplus_i$. It is also known that for each pair $(A_i, B_i) \in \mathscr{W}$ (i) $\square_i \cap \boxplus_i = \emptyset$, (ii) $\max\{\mathrm{diam}(\square_i), \mathrm{diam}(\boxplus_i)\} \leq \varepsilon\phi(\square_i, \boxplus_i)$, and each cell appears in $O(\varepsilon^{-d} \log n)$ cells (see Figure 4). We will use $\mathscr{W} = \{(\square_1, \boxplus_i), \ldots, (\square_s, \boxplus_s)\}$ to denote the WSPD, with A_i, B_i being implicitly defined from the cells. Using the techniques in [15, 26], the quad tree $\mathscr{T}$ and the WSPD $\mathscr{W}$ can be maintained under insertions and deletions of points in $\widetilde{O}(\varepsilon^{-d})$ time.

Data structure. We construct a quad tree $\mathscr{T}$ on $A \cup B$. For each node $u \in \mathscr{T}$, we store a pointer A_u (and B_u) to the leftmost leaf of subtree $\mathscr{T}_u$ that contains a point from A (and B). Furthermore, we store sorted lists L_A and L_B of the leaves that contain points from A and B, respectively. We use these pointers and lists to report points in $\square_u$ with $O(1)$ delay. Using $\mathscr{T}$, we can construct a WSPD $\mathscr{W} = \{(\square_1, \boxplus_1), \ldots, (\square_s, \boxplus_s)\}$, $s = O(\varepsilon^{-d})$. For each i, let $\Delta_i = \min_{p\in\square_i, q\in\boxplus_i} \phi(p, q)$. We store all pairs $(\square_i, \boxplus_i)$ in a red-black tree $\mathscr{Z}$ using Δ_i as the key. The data structure has $O(\varepsilon^{-d} n)$ size and $O(\varepsilon^{-d} n \log n)$ construction time.

Update. After inserting or deleting an input point, the quad tree $\mathscr{T}$ and W can be updated in $\widetilde{O}(\varepsilon^{-d})$ time, following the standard techniques in [15, 26]. As there are at most $\widetilde{O}(\varepsilon^{-d})$ pairs changed, we can update the tree $\mathscr{Z}$ in $\widetilde{O}(\varepsilon^{-d})$ time. Furthermore, we note that there are only $O(1)$ changes in the structure of quad tree $\mathscr{T}$ and the height of $\mathscr{T}$ is $O(\log n)$, so we can update all necessary pointers A_u, B_u and sorted lists L_A, L_B in $O(\log n)$ time.

Enumeration. Let r be the threshold parameter specified as part of a query. We traverse the tree $\mathscr{X}$ in order and report pairs of cells until we reach a pair $(\square_j, \boxplus_j)$ with $\Delta_j > r$. For each pair $(\square_i, \boxplus_i)$ reported, we traverse we enumerate $(a, b) \in (A \cap \square_i) \times (B \cap \boxplus_i)$ using the stored pointers and the sorted lists L_A, L_B. The delay guarantee is $O(1)$.

Let $(a, b) \in A \times B$ be a pair with $\phi(a, b) \le r$. Implied by the definition, there exists a unique pair $(A_i, B_i) \in \mathscr{W}$ such that $a \in A_i$ and $b \in B_i$. Notice that $\phi(\square_i, \boxplus_i) \le \phi(a, b) \le r$. Thus, all results of $A_i \times B_i$ will be reported, including (a, b). Next, let $(\square_i, \boxplus_i)$ be a pair that is reported by the enumeration procedure in $\mathscr{X}$, with $\phi(\square_i, \boxplus_i) \le r$. For any pair of points $x \in \square_i, y \in \boxplus_i$, we have $\phi(x, y) \le \phi(\square_i, \boxplus_i) + \mathrm{diam}(\square_i) + \mathrm{diam}(\boxplus_i) \le (1 + 2 \cdot \frac{\varepsilon}{2}) \cdot \phi(\square_i, \boxplus_i) \le (1 + \varepsilon)r$, thus $\phi(a, b) \le (1 + \varepsilon)r$ for any pair $(a, b) \in A_i \times B_i$.

► **Theorem 8.** *Let A, B be two sets of points in $\mathbb{R}^d$ for constant d, with $O(n^{O(1)})$ spread and $|A| + |B| = n$. A data structure of $O(\varepsilon^{-d} n)$ space can be built in $\widetilde{O}(\varepsilon^{-d} n)$ time and updated in $\widetilde{O}(\varepsilon^{-d})$ time, while supporting ε-approximate enumeration for similarity join under any ℓ_p metric with $O(1)$ delay, for any query similarity threshold r.*

3.2 Fixed distance threshold

Without loss of generality we assume that $r = 1$. We use a grid-based data structure for enumerating similarity join with fixed distance threshold r.

Data structure. Let $\mathscr{G}$ be an infinite uniform grid[1] in $\mathbb{R}^d$, where the size of each grid cell is $\frac{\varepsilon}{2\sqrt{d}}$ and the diameter is $\frac{\varepsilon}{2}$. For a pair of cells $c, c' \in$, define $\phi(c, c') = \min_{p \in c, q \in c'} \phi(p, q)$. Each grid cell $c \in \mathscr{G}$ is associated with (1) $A_c = A \cap c$; (2) $B_c = B \cap c$; (3) $m_c = \sum_{c' : \phi(c, c') \le 1} |B_{c'}|$ as the number of points in B that lie in a cell c' within distance 1 from cell c. Let $\mathscr{C}_{NE} \subseteq \mathscr{G}$ be the set of all non-empty cells, $\mathscr{C}_{NE} = \{c \in \mathscr{G} \mid A_c \cup B_c \neq \emptyset\}$. A grid cell $c \in \mathscr{C}_{NE}$ is *active* if and only if $A_c \neq \emptyset$ and $m_c > 0$ (see Figure 5 for an example). Let $\mathscr{C} \subseteq \mathscr{C}_{NE}$ be the set of active grid cells (Figure 5). Notice that a grid cell is stored when there is at least one point from A or B lying inside it, so $|\mathscr{C}_{NE}| \le n$. Finally, we build a balanced search tree on $\mathscr{C}$ so that whether a cell c is stored in $\mathscr{C}$ can be answered in $O(\log n)$ time. Similarly, we build another balanced search tree to store the set of non-empty cells $\mathscr{C}_{NE}$.

Update. Assume point $a \in A$ is inserted into cell $c \in \mathscr{G}$. If c is already in $\mathscr{C}_{NE}$, simply add a to A_c. Otherwise, we add c to $\mathscr{C}_{NE}$ with $A_c = \{a\}$ and update m_c as follows. We visit each cell $c' \in \mathscr{C}_{NE}$ with $\phi(c, c') \le 1$, and add $|B_{c'}|$ to m_c. A point of A is deleted in a similar manner. Assume point $b \in B$ is inserted into cell $c \in \mathscr{G}$. If $c \notin \mathscr{C}_{NE}$, we add it to $\mathscr{C}_{NE}$. In any case, we first insert b into B_c and for every cell $c' \in \mathscr{C}_{NE}$ with $\phi(c, c') \le 1$, we increase $m_{c'}$ by 1 and add c' to $\mathscr{C}$ if c' turns from inactive to active. A point from B is deleted in a similar manner. As there are $O(\varepsilon^{-d})$ cells within distance 1 from c, this procedure takes $\widetilde{O}(\varepsilon^{-d})$ time.

Enumeration. For each active cell $c \in \mathscr{C}$, we visit each cell $c' \in \mathscr{C}_{NE}$ within distance 1. If $B_{c'} \neq \emptyset$, we report all pairs of points in $A_c \times B_{c'}$. It is obvious that each pair of points is enumerated at most once. For an active cell c, there must exists a pair $(a \in A_c, b \in B_{c'})$ for some cell $c' \in \mathscr{C}_{NE}$ such that $\phi(a, b) \le \phi(c, c') + \mathrm{diam}(c) + \mathrm{diam}(c') \le 1 + \varepsilon$. So it takes at most $O(\varepsilon^{-d} \log n)$ time before finding at least one result for c; thus, the delay is $O(\varepsilon^{-d} \log n)$.

[1] When extending it to any ℓ_p norm, the size of each grid cell is $\varepsilon/(2d^{1/p})$ and the diameter is $\frac{\epsilon}{2}$.

Furthermore, consider every pair of points a, b with $\phi(a,b) \leq 1$. Assume $a \in c$ and $b \in c'$. By definition, c must be an active grid cell. Thus, (a,b) will definitely be enumerated in this procedure, thus guaranteeing the correctness of ε-enumeration.

▶ **Theorem 9.** *Let A, B be two sets of points in $\mathbb{R}^d$ for some constant d, with $|A| + |B| = n$. A data structure of $O(n)$ size can be constructed in $O(n\varepsilon^{-d}\log n)$ time and updated in $O(\varepsilon^{-d}\log n)$ time, while supporting ε-approximate enumeration of similarity join under any ℓ_p metric with $O(\varepsilon^{-d}\log n)$ delay.*

Note that if for each active cell $c \in \mathscr{C}$, we store the cells within distance 1 that contain at least a point from B, i.e., $\{c' \in C \mid \phi(c,c') \leq 1, B_c \neq \emptyset\}$, then the delay can be further reduced to $O(1)$ but the space becomes $O(\varepsilon^{-d}n)$.

4 Similarity Join in High Dimensions

So far, we have treated the dimension d as a constant. In this section we describe a data structure for approximate similarity join using the *locality sensitive hashing* (LSH) technique, so that the dependency on d is a small polynomial. For simplicity, we assume that r is fixed, however our results can be extended to the case in which r is part of the enumeration query.

For $\varepsilon > 0$, $0 < p_2 < p_1 \leq 1$, a family $\mathscr{H}$ of hash functions is $(r, (1+\varepsilon)r, p_1, p_2)$-sensitive, if for any uniformly chosen hash function $h \in H$ and any two points x, y:

- $\Pr[h(x) = h(y)] \geq p_1$ if $\phi(x,y) \leq r$;
- $\Pr[h(x) = h(y)] \leq p_2$ if $\phi(x,y) \geq (1+\varepsilon)r$.

The quality of $\mathscr{H}$ is measured by $\rho = \frac{\ln p_1}{\ln p_2} < 1$, which is upper bounded by a number that depends only on ε; and $\rho = \frac{1}{1+\varepsilon}$ for many common distance functions [27, 22, 29]. For ℓ_2 the best result is $\rho \leq \frac{1}{(1+\varepsilon)^2} + o(1)$ [9].

The essence of LSH is to hash "similar" points into the same buckets with high probability. A simple approach based on LSH is to (i) hash points into buckets; (ii) probe each bucket and check for each pair of points $(a,b) \in A \times B$ inside the same bucket whether $\phi(a,b) \leq r$; and (iii) report (a,b) if the inequalities holds. However, two challenges arise for enumeration. First, without any knowledge of false positive results inside each bucket, checking every pair of points could lead to a huge delay. Our key insight is that after checking a small number (to be determined later) of pairs of points in one bucket, we can safely skip the bucket since any pair of result missed in this bucket will be found in another one with high probability. Second, one pair of points may collide under multiple hash functions, so an additional step is necessary in the enumeration to remove duplicates. If we wish to keep the size of data structure to be near-linear and if we are not allowed to store the reported pairs (so that the size remains near linear), detecting duplicates requires care.

Since we do not define new hash functions, our results hold for any metric for which LSH works, in particular for Hamming, ℓ_2, ℓ_1 metrics.

Data structure. We fix an LSH family $\mathscr{H}$. Let ρ be its quality parameter. To ensure high-probability guarantee, we maintain $O(\log n)$ copies of the whole data structure below.

We randomly choose $\tau = O(n^\rho)$ hash functions. Each possible value in the range of hash functions defines a bucket. We maintain some extra statistics for all buckets. We choose a parameter $M = O(n^\rho)$. For a bucket $\square$, let $A_\square = A \cap \square$ and $B_\square = B \cap \square$. We choose two arbitrary subsets $\bar{A}_\square, \bar{B}_\square$ of $A_\square, B_\square$, respectively, of M points each. For each point $a \in \bar{A}_\square$, we maintain a counter $\beta_a = |\{b \in \bar{B}_\square \mid \phi(a,b) \leq 2(1+\varepsilon)r\}|$, i.e., the number of points in $\bar{B}_\square$ with distance at most $2(1+\varepsilon)r$ from a. We store $\bar{A}_\square$ in an increasing order of their

β values. If there exists some positive counter $\beta_a > 0$, we denote bucket $\Box$ as *active* and store an arbitrary pair $(a,b) \in \bar{A}_\Box \times \bar{B}_\Box$ with $\phi(a,b) \le 2(1+\varepsilon)r$ as its *representative* pair, denoted as $(a_\Box, b_\Box)$. Let $\mathscr{C}$ denote the set of active buckets.

Before diving into the details of update and enumeration, we give some intuition about active buckets. Given a set P of points and a distance threshold r, let $\overline{\mathscr{B}}(q,P,r) = \{p \in P \mid \phi(p,q) > r\}$. For any pair of points $(a,b) \in A \times B$ and a hashing bucket $\Box$, we refer to $\Box$ as the *proxy bucket* for (a,b) if (i) $a \in A_\Box, b \in B_\Box$; (ii) $|\bar{\mathscr{B}}(a, A_\Box \cup B_\Box, (1+\varepsilon)r)| \le M$. A crucial property of proxy bucket is captured by Lemma 10. Moreover, it can be shown that with high probability each close pair of points has a proxy bucket in Lemma 11 (more details are given in the full version [4]). In this way, it is safe to skip a bucket after we have seen up to M^2 faraway pairs of points inside, since the close pairs of points in this bucket will be captured by other buckets. In this way, we only need to report pairs from active buckets.

▶ **Lemma 10.** *For any bucket $\Box$, if there exist M points from $A_\Box$ and $B_\Box$ each, such that none of the M^2 pairs has its distance within $2(1+\varepsilon)r$, $\Box$ is not a proxy bucket for any pair $(a,b) \in A_\Box \times B_\Box$ with $\phi(a,b) \le r$.*

Proof of Lemma 10. Let A', B' be two sets of M points from $A_\Box, B_\Box$ respectively. We assume that all pairs of points in $A' \times B'$ have their distances larger than $2(1+\varepsilon)r$. Observe that $\Box$ is not a proxy bucket for any pair $(a \in A', b \in B')$. It remains to show that $\Box$ is not a proxy bucket for any pair $(a \in A_\Box \setminus A', b \in B_\Box)$. Assume $b \in B_\Box \setminus B'$ (the case is similar if $b \in B'$). If $A' \subseteq \bar{\mathscr{B}}(a, A, (1+\varepsilon)r)$ or $B' \subseteq \bar{\mathscr{B}}(a, B, (1+\varepsilon)r)$, $\Box$ is not a proxy bucket for (a,b). Otherwise, there must exist at least one point $a' \in A'$ as well as $b' \in B'$ such that $\phi(a,a') \le (1+\varepsilon)r$ and $\phi(a,b') \le (1+\varepsilon)r$, so $\phi(a',b') \le \phi(a,a') + \phi(a,b') \le 2(1+\varepsilon)r$. Thus, $(a',b') \in A' \times B'$ is a pair within distance $2(1+\varepsilon)r$, coming to a contradiction. ◀

▶ **Lemma 11** ([27, 28, 32])**.** *For $M = O(n^\rho)$, with probability $1 - 1/n$, every pair of points (a,b) with $\phi(a,b) \le r$ has a proxy bucket.*

Update. Assume a point $a \in A$ is being inserted. We visit every bucket $\Box$ into which a is hashed and insert a to $A_\Box$. If $|\bar{A}_\Box| \ge M$, we do nothing. Otherwise, we insert a to $\bar{A}_\Box$ and compute its counter β_a. If $\beta_a > 0$ and $\Box \notin \mathscr{C}$, we add $\Box$ to $\mathscr{C}$ and store an arbitrary pair (a,b) for some $b \in \bar{B}_\Box$ with $\phi(a,b) \le 2(1+\varepsilon)r$, as the representative pair of $\Box$. Notice that there always exists such a point b since $\beta_a > 0$.

Assume a point $a \in A$ is being deleted. We visit every bucket $\Box$ into which a is hashed and delete a from $A_\Box$. If $a \in \bar{A}_\Box$, we delete it from $\bar{A}_\Box$ and insert an arbitrary point (if any) from $A_\Box \setminus \bar{A}_\Box$ into $\bar{A}_\Box$. If $a = a_\Box$, i.e., a participates in the representative pair of $\Box$, we find a new representative pair by considering an arbitrary point $a' \in \bar{A}_\Box$ with $\beta_{a'} > 0$. If no such point exists, we remove $\Box$ from $\mathscr{C}$.

The case when point $b \in B$ is inserted or deleted is similar but with slight differences. Assume a point $b \in B$ is being inserted. We visit every bucket $\Box$ into which b is hashed and inserted b to $B_\Box$. If $|\bar{B}_\Box| \ge M$, we do nothing. Otherwise, we insert b to $\bar{B}_\Box$ and increment counter β_a for every point $a \in \bar{A}_\Box$ with $\phi(a,b) \le 2(1+\epsilon)r$. Moreover, if $\Box \notin \mathscr{C}$ and there exists some point $a \in \bar{A}_\Box$ with $\beta_a > 0$ after update, say a', we store (a',b) as the representative pair of $\Box$ and add $\Box$ to $\mathscr{C}$.

Assume a point $b \in B$ is being deleted. We visit every bucket $\Box$ into which b is hashed and delete b from $B_\Box$. If $b \in \bar{B}_\Box$, we delete it from $\bar{B}_\Box$ and insert an arbitrary point (if any) from $B_\Box \setminus \bar{B}_\Box$ into $\bar{B}_\Box$. Moreover, we need to update counter β_a for every point $a \in \bar{A}_\Box$. If

$b = b_\square$, i.e., b participates in the representative pair of $\square$, we find a new representative pair by considering an arbitrary point $a \in \bar{A}_\square$ with $\beta_a > 0$. If no such pair exists, we remove $\square$ from $\mathscr{C}$.

After performing $n/2$ updates, we reconstruct the entire data structure from scratch.

Enumeration. Let $\mathscr{R}$ be the set of representative pairs. As mentioned, the high-level idea is to enumerate representative pairs from active buckets. More specifically, we start with an arbitrary representative pair $(a, b) \in \mathscr{R}$, and enumerate all pairs involving point a from $\mathscr{C}(a)$, where $\mathscr{C}(a) \subseteq \mathscr{C}$ is the set of active buckets containing a. Then, we remove a from each bucket $\square \in \mathscr{C}(a)$. If $a \in \bar{A}_\square$, we remove a from $\bar{A}_\square$; moreover, we add a point $a' \in \bar{A} - \bar{A}_\square$ to $\bar{A}_\square$ to ensure $|\bar{A}_\square| = M$ if $|A_\square| \geq M$ and compute $\beta_{a'}$. If $a = a_\square$, we compute a new representative pair $(a_\square, b_\square)$ of $\square$. If no such new representative pair exists, we just remove $\square$ from the set of active buckets; otherwise, we add the new pair $(a_\square, b_\square)$ to $\mathscr{R}$. We repeat this process until $\mathscr{R}$ becomes empty. The whole procedure is described in Algorithm 1.

Algorithm 1 ENUMERATE.

```
1  $\mathscr{R} \leftarrow \{(a_\square, b_\square) : \square \in \mathscr{C}\}$;
2  while $\mathscr{R} \neq \emptyset$ do
3      $(a, b) \leftarrow \mathscr{R}$;
4      $\mathscr{C}(a) \leftarrow \{\square \in \mathscr{C} : a \in A_\square\}$;
5      Report$(a, \mathscr{C}(a))$;
6      foreach $\square \in \mathscr{C}(a)$ do
7          $A_\square \leftarrow A_\square - \{a\}$;
8          if $a \in \bar{A}_\square$ then
9              $\bar{A}_\square \leftarrow \bar{A}_\square - \{a\}$;
10             if $|A_\square| \geq M$ then
11                 Pick arbitrary point $a' \in A_\square - \bar{A}_\square$;
12                 $\bar{A}_\square \leftarrow \bar{A}_\square \cup \{a'\}$;
13         if $a = a_\square$ then
14             $\mathscr{R} \leftarrow \mathscr{R} - \{(a_\square, b_\square)\}$;
15             Recompute $(a_\square, b_\square)$ for $\square$;
16             if $(a_\square, b_\square) = \emptyset$ then
17                 $\mathscr{C} \leftarrow \mathscr{C} - \{\square\}$;
18             else
19                 $\mathscr{R} \leftarrow \mathscr{R} \cup \{(a_\square, b_\square)\}$;
```

In line 5 of Algorithm 1, we report all pairs including point a is described in Algorithm 2. For a bucket $\square \in \mathscr{C}(a)$, whenever we report a pair (a, b), we *mark* b with a. If b was already marked by some other point a' because (a', b) was reported, we overwrite a' with a. Let $X(\square, a) \subseteq B_\square$ be the set of points in $B_\square$ marked with a. We visit every bucket $\square \in \mathscr{C}(a)$ and check the distances between a and points in $B_\square \setminus X(\square, a)$. Each time a pair (a, b) with $\phi(a, b) \leq 2(1 + \varepsilon)r$ is found, we report it and mark b with a to ensure that we will not report (a, b) again. More specifically, we go over each active bucket $\square \in \mathscr{C}(a)$ into which b is also hashed, and put a marker on b with respect to a. Implied by line 3 of Algorithm 2, we only consider points not marked by $X(\square, a)$, thus avoiding repeated enumeration.[2] Whenever more than M points from $B_\square$ have been checked without finding a pair with distance less than $2(1+\varepsilon)r$ (or if all points in $B_\square$ have been considered), we just skip this bucket.

Algorithm 2 Report$(a, \mathscr{C}(a))$.

```
1  foreach $\square \in \mathscr{C}(a)$ do
2      $i \leftarrow 0$;
3      foreach $b \in B_\square - X(\square, a)$ do
4          if $\phi(a,b) \le 2(1+\varepsilon)r$ then
5              Report $(a,b)$;
6              foreach $\square' \in \mathscr{C}(a)$ with $b \in B_{\square'}$ do
7                  $X(\square', a) \leftarrow X(\square', a) \cup \{b\}$;
8          else
9              $i \leftarrow i+1$;
10             if $i > M$ then break;
```

Correctness analysis. The report procedure guarantees that each pair of points is enumerated at most once. It remains to show that $(1+2\varepsilon)$-approximate enumeration is supported.

We show that $(1+2\varepsilon)$-approximate enumeration is supported with probability $1-1/n$. It can be easily checked that any pair of points farther than $2(1+\varepsilon)r$ will not be enumerated. Hence, it suffices to show that all pairs within distance r are enumerated with high probability. From Lemma 11, with high probability every pair within distance 1 has a proxy bucket. Let $\square$ be a proxy bucket for pair (a,b). Implied by Lemma 10, there exist no M points from $A_\square$ (for example $\bar{A}_\square$) and M points from $B_\square$ (for example $\bar{B}_\square$) such that all M^2 pairs have their distance larger than $2(1+\varepsilon)r$, so $\square$ is active. Moreover, from the definition of M and the proof of Lemma 10 (check [4] the complete proof) there exist no M points from $B_\square$ such that all of them have distance more than $2(1+\varepsilon)r$ from a, so Algorithm 2 will report (a,b).

Complexity analysis. Recall that $\tau, M = O(n^\rho)$. The data structure uses $O(dn + n\tau\log n)$ space since we only use linear space with respect to the points in each bucket. The update time is $\widetilde{O}(dM\cdot\tau)$ as there are $\widetilde{O}(\tau)$ buckets to be investigated and it takes $\widetilde{O}(dM)$ time to update the representative pair. After $n/2$ updates we re-build the data structure so the update time is amortized. The delay is $\widetilde{O}(dM\cdot\tau)$. In order to replace a with an arbitrary point $a' \in A_\square - \bar{A}_\square$ in line 8 of Algorithm 1 we need $O(dM)$ time and there are $\widetilde{O}(\tau)$ buckets that we need to visit. In total, this step takes $\widetilde{O}(dM\tau)$ time. In Algorithm 2, we spend $\widetilde{O}(dM\tau)$ time to report a pair of results and $\widetilde{O}(\tau)$ time to mark point b over all buckets.

We conclude with the following result:

▶ **Theorem 12.** *Let A and B be two sets of points in $\mathbb{R}^d$, where $|A|+|B|=n$ and let ε, r be positive parameters. For $\rho = \frac{1}{(1+\varepsilon)^2}+o(1)$, a data structure of $\widetilde{O}(dn+n^{1+\rho})$ size can be constructed in $\widetilde{O}(dn^{1+2\rho})$ time, and updated in $\widetilde{O}(dn^{2\rho})$ amortized time, while supporting $(1+2\varepsilon)$-approximate enumeration for similarity join under the ℓ_2 metric with $\widetilde{O}(dn^{2\rho})$ delay.*

[2] To avoid conflicts with the markers made by different enumeration queries, we can generate them randomly and delete old values by lazy updates [24, 38, 39] after finding new pairs to report.

▶ Remark. Alternatively, we can insert or delete points from $A \cup B$ without maintaining the sets $\bar{A}_\square, \bar{B}_\square$ for every bucket $\square$. In the enumeration phase, given a bucket $\square$, we can visit M arbitrary points from $A_\square$ and M arbitrary points from $B_\square$ and compute their pairwise distances. If there is no pair $(a \in A_\square, b \in B_\square)$ with $\phi(a,b) \le 2(1+\varepsilon)r$, we just skip this bucket. Otherwise, we report the pair (a,b) and invoke Algorithm 2 for point a. In this case, the update time decreases to $\widetilde{O}(dn^\rho)$ but the delay will increases to $\widetilde{O}(dn^{3\rho})$.

The same result holds for Hamming and ℓ_1 metrics with $\rho = \frac{1}{1+\varepsilon}$. Using [32], for the Hamming metric and $\varepsilon > 1$ we can get $M = O(1)$. Skipping the details, we have:

▶ **Theorem 13.** *Let A and B be two sets of points in $\mathbb{H}^d$, where $|A| + B| = n$ and let ε, r be positive parameters. For $\rho = \frac{1}{1+\varepsilon}$, a data structure of $\widetilde{O}(dn + n^{1+\rho})$ size can be built in $\widetilde{O}(dn^{1+\rho})$ time, and updated in $\widetilde{O}(dn^\rho)$ amortized time, while supporting $(3+2\varepsilon)$-approximate enumeration for similarity join under the Hamming metric with $\widetilde{O}(dn^\rho)$ delay.*

In the full version [4], we show that our results can be extended to the case where r is part of the enumeration procedure, and we also prove a lower bound relating similarity join to the approximate nearest neighbor query.

5 Conclusion

In this paper, we presented dynamic data structures for enumerating similarity join queries with delay guarantees. We present several efficient data structures for dynamic enumeration of similarity joins in constant or higher dimensions over various metrics.

Note that our data structures provide worst-case delay guarantee for arbitrary input data and arbitrary updates. In practice, most real-world update sequences are “nice”, nowhere near these worst-case scenarios; and input points from two sets might be dependent, or follow certain parameterized distributions. A more fine-grained analysis on the intrinsic difficulty of update sequences in dynamic enumeration of similarity joins is quite interesting but still open. Similar instance-dependent analysis has been considered in [46].

Another interesting direction is to investigate other variants of similarity join queries under more general metrics, such as doubling metric space, and more complicated distance functions, such as the cosine distance.

References

1 Peyman Afshani. Improved pointer machine and I/O lower bounds for simplex range reporting and related problems. In *Proc. 28th Annual Symp. Comput. Geom.*, pages 339–346, 2012.

2 P. K. Agarwal. Simplex range searching and its variants: A review. In M. Loebl, J. Nešetřil, and R. Thomas, editors, *A Journey Through Discrete Mathematics*, pages 1–30. Springer, 2017.

3 P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Contemporary Mathematics*, volume 223, pages 1–56. American Math. Society, 1999.

4 Pankaj K. Agarwal, Xiao Hu, Stavros Sintos, and Jun Yang. Dynamic enumeration of similarity joins. `arXiv:2105.01818`.

5 Pankaj K Agarwal, Junyi Xie, Jun Yang, and Hai Yu. Monitoring continuous band-join queries over dynamic data. In *Int. Symp. Algorithms Comput.*, pages 349–359. Springer, 2005.

6 Pankaj K Agarwal, Junyi Xie, Jun Yang, and Hai Yu. Scalable continuous query processing by tracking hotspots. In *Proc. 32nd Int. Conf. Very Large Databases*, pages 31–42, 2006.

7 Dror Aiger, Haim Kaplan, and Micha Sharir. Reporting neighbors in high-dimensional euclidean space. *SIAM J. Comput.*, 43(4):1363–1395, 2014.

8 A. Al-Badarneh, A. Al-Abdi, M. Sana'a, and H. Najadat. Survey of similarity join algorithms based on mapreduce. *MATTER: International Journal of Science and Technology*, 2016.

9 Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proc. 47th Annual IEEE Symp. Foundations Comput. Sci.*, pages 459–468, 2006.

10 N. Augsten and M. Böhlen. Similarity joins in relational database systems. *Synthesis Lectures on Data Management*, 5(5):1–124, 2013.

11 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proc. 21th Int. Workshop Computer Science Logic*, pages 208–222, 2007.

12 J. Bentley. Decomposable searching problems. Technical report, CMU, 1978.

13 J. Bentley and J. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

14 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Proc. 36th ACM SIGMOD-SIGACT-SIGAI Symp. Principles Database Systems*, pages 303–318, 2017.

15 P. Callahan. *Dealing with Higher dimensions: The Well-Separated Pair Decomposition and Its Applications*. PhD thesis, Johns Hopkins University, 1995.

16 N. Carmeli and M. Kröll. Enumeration complexity of conjunctive queries with functional dependencies. *Theory of Computing Systems*, pages 1–33, 2019.

17 T. Chan. Optimal partition trees. *Discrete & Comput. Geom.*, 47(4):661–690, 2012.

18 Sirish Chandrasekaran and Michael J Franklin. Streaming queries over streaming data. In *Proc. 28th Int. Conf. Very Large Databases*, pages 203–214, 2002.

19 S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. 22nd Int. Conf. Data Eng.*, pages 5–5, 2006.

20 B. Chazelle and B. Rosenberg. Simplex range reporting on a pointer machine. *Comput. Geom.: Theory Apps.*, 5(5):237–247, 1996.

21 Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proc. 19th ACM SIGMOD Int. Conf. Management Data*, pages 379–390, 2000.

22 Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. 20th Annual Symp. Comput. Geom.*, pages 253–262, 2004.

23 M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

24 J. Erickson. Static-to-dynamic transformations. `http://jeffe.cs.illinois.edu/teaching/datastructures/notes/01-statictodynamic.pdf`.

25 Françoise Fabret, H Arno Jacobsen, François Llirbat, João Pereira, Kenneth A Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. 20th ACM SIGMOD Int. Conf. Management Data*, pages 115–126, 2001.

26 John Fischer and Sariel Har-Peled. Dynamic well-separated pair decomposition made easy. In *Proc. 17th Canadian Conf. Comput. Geom.*, pages 235–238, 2005.

27 Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Proc. 25nd Int. Conf. Very Large Databases*, volume 99 (6), pages 518–529, 1999.

28 S. Har-Peled. *Geometric Approximation Algorithms*. Number 173 in Mathematical Surveys and Monographs. American Math. Society, 2011.

29 Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of Computing*, 8(1):321–350, 2012.

30 Sariel Har-Peled and Manor Mendel. Fast construction of nets in low-dimensional metrics and their applications. *SIAM J. Comput.*, 35(5):1148–1184, 2006.

31 Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proc. 36th ACM SIGMOD Int. Conf. Management Data*, pages 1259–1274, 2017.

32 Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. 30th Annual ACM Symp. Theory of Comput.*, pages 604–613, 1998.

33 Edwin H Jacox and Hanan Samet. Metric space similarity joins. *ACM Trans. Database Systems*, 33(2):1–38, 2008.

34 Ahmet Kara, Hung Q Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. In *Proc. 22nd Int. Conf. Database Theory*, pages 4:1–4:18, 2019.

35 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proc. 39th ACM SIGMOD-SIGACT-SIGAI Symp. Principles Database Systems*, pages 375–392, 2020.

36 Hans-Peter Lenhof and Michiel Smid. Sequential and parallel algorithms for the k closest pairs problem. *Int. J. Comput. Geom. & Appl.*, 5(03):273–288, 1995.

37 J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8(3):315–334, 1992.

38 M. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inf. Process. Lett.*, 12(4):168–173, 1981.

39 Mark H Overmars. *The design of dynamic data structures*, volume 156. Springer Science & Business Media, 1987.

40 Rodrigo Paredes and Nora Reyes. Solving similarity joins and range queries in metric spaces with the list of twin clusters. *J. Discrete Algorithms*, 7(1):18–35, 2009.

41 H. Samet. Spatial data structures: Quadtree, octrees and other hierarchical methods, 1989.

42 Luc Segoufin. Enumerating with constant delay the answers to a query. In *Proc. 16th Int. Conf. Database Theory*, pages 10–20, 2013.

43 Y. Silva, W. Aref, and M. Ali. The similarity join database operator. In *Proc. 26th Int. Conf. Data Engi.*, pages 892–903, 2010.

44 Yasin N Silva, Jason Reed, Kyle Brown, Adelbert Wadsworth, and Chuitian Rong. An experimental survey of mapreduce-based similarity joins. In *Int. Conf. Similarity Search Appl.*, pages 181–195. Springer, 2016.

45 J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering? an adaptive framework for similarity join and search. In *Proc. 31th ACM SIGMOD Int. Conf. Management Data*, pages 85–96, 2012.

46 Q. Wang and K. Yi. Maintaining acyclic foreign-key joins under updates. In *Proc. 39th ACM SIGMOD Int. Conf. Management Data*, pages 1225–1239, 2020.

47 D. E. Willard. Applications of range query theory to relational data base join and selection operations. *J. Comput. Syst. Sci.*, 52(1):157–169, 1996.

48 Dan E Willard. Polygon retrieval. *SIAM J. Comput.*, 11(1):149–165, 1982.

49 Kun-Lung Wu, Shyh-Kwei Chen, and Philip S Yu. Interval query indexing for efficient stream processing. In *Proc. 30th ACM Int. Conf. Inform. Knowledge Management*, pages 88–97, 2004.