Dynamic Enumeration of Similarity Joins

Pankaj K. Agarwal, Xiao Hu, Stavros Sintos, Jun Yang
Duke University
{pankaj,xh102,ssintos,junyang}@cs.duke.edu

ABSTRACT

This paper considers enumerating answers to similarity join queries under dynamic updates: Given two sets of points A, B in $\mathbb{R}^d$, a metric $\phi(\cdot)$, and a similarity threshold $r > 0$, it asks to report all pairs of points $(a, b) \in A \times B$ with $\phi(a, b) \leq r$. Our goal is to design an index that can be efficiently updated when an input point is inserted or deleted. Furthermore, whenever asked, it enumerates all join results with worst-case delay guarantee, i.e., the time between enumerating two consecutive answers is bounded. A minimal index is one that can be built in near-linear time with near-linear space, and supports polylog-time maintenance under update and polylog-delay enumeration.

We propose several efficient indexes for answering similarity joins under ℓ_1, ℓ_2, and ℓ_∞ metrics. More specifically, we obtain a minimal index for exact similarity join enumeration under ℓ_1/ℓ_∞ metrics; and a minimal index for *approximated* similarity join enumeration under ℓ_2 metric, where the distance threshold is a soft constraint. In high dimensions, we present an efficient index toward a worst-case delay-guarantee framework using *locality sensitive hashing* (LSH). Beyond a fixed similarity threshold, we also investigate the setting in which the similarity threshold is part of enumeration queries. If the input has bounded spread, we propose a unified minimal index for approximated similarity join enumeration under any ℓ_p norm, which is oblivious to the threshold r.

ACM Reference Format:
Pankaj K. Agarwal, Xiao Hu, Stavros Sintos, Jun Yang. 2020. Dynamic Enumeration of Similarity Joins. In *Proceedings of* . ACM, New York, NY, USA, 17 pages.

1 INTRODUCTION

Dynamic queries are central to many real-time data analytical systems, where input data can be updated. Given a database $\mathcal{D}$ and a query Q, let $Q(\mathcal{D})$ be the result of query over $\mathcal{D}$. An update Δ on $\mathcal{D}$ could be an insertion or deletion of a tuple. The goal is to design data structures such that $Q(\mathcal{D} + \Delta)$ can be computed efficiently. Traditional databases have approached this problem with *incremental view maintenance* [31] or *high-order incremental view maintenance* [4, 40, 41, 46], by materializing a set of views based on the original query or delta queries (taking changes to the view as a new query) recursively. However, this approach suffers from the high space complexity of views, and unbounded time of maintenance after each update.

A different approach for handling this problem is to build an index for $\mathcal{D}$ such that each result of $Q(\mathcal{D})$ can be *enumerated* from it efficiently. In large-scale analytical systems, the query result may be huge and an application may process the first few answers while waiting for the remaining ones. From this perspective, it is useful to obtain an index that offers guarantees for the *regularity* of the enumerating process; the *delay* is an important measurement. Formally, 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 be at most δ. We are interested in constant-delay enumeration [10]. Furthermore, in the dynamic setting, any index should also be updated efficiently once a tuple is inserted or deleted.

In this paper, we focus on *similarity join*, which has been extensively studied in the database and data mining literature [21, 37, 49, 54, 55]. However, it is still unraveled how to enumerate answers to similarity join queries using indexes that can be updated efficiently, with provable guarantees on their performance. This work makes progress on this question.

1.1 Problem definition

The similarity-join problem is defined as follows: Given two sets of points A and B in $\mathbb{R}^d$, a metric $\phi(\cdot)$, and a similarity threshold $r > 0$, report all pairs of points $(a, b) \in A \times B$ with $\phi(a, b) \leq r$. Let $n = |A| + |B|$ be the *input size*. We are also interested in the case where the distance threshold is a soft constraint. For some parameter $\varepsilon > 0$, the *ε-approximate similarity join* relaxes the distance threshold: all pairs of points $(a, b) \in A \times B$ with $\phi(a, b) \leq r$ should be returned, as well as some pairs of points $(a, b) \in A \times B$ with $r < \phi(a, b) \leq (1+\varepsilon)r$ may be returned. No pair $(a, b) \in A \times B$ with $\phi(a, b) > (1 + \varepsilon)r$ is returned. In this paper, we are mostly interested in the ℓ_1, ℓ_2 and ℓ_∞ metrics, although some of our results can be extended to other metrics as well.

When considering a similarity-join query in dynamic setting, our goal is to design an index that can be efficiently updated when an input point is inserted or deleted. Furthermore, whenever a query is issued, it enumerates all (exact or approximate) join results with worst-case delay guarantee. The complexity of such an index is measured by the following three parameters, space for storing the index, update time for the index after inserting/deleting a point, and the worst-case delay once an enumeration query is issued. An index for dynamic similarity join is called *optimal* if it uses linear space, and supports constant-time maintenance under update and constant-delay enumeration [14, 35]. Obviously, an optimal index is the best we could hope for, but may not always be achievable. In this paper, we relax this notion slightly by tolerating poly-logarithmic factors in each quality measure, i.e., an index is *minimal* if it has $O(n\text{polylog}(n))$ size and supports polylog-time maintenance and polylog-delay enumeration.

1.2 Previous results

Dynamic enumeration of conjunctive query. Equi-join is a special case of similarity join with $r = 0$, i.e., two tuples can be joined if

and only if their join attributes are equal. Enumeration of conjunctive queries against relational databases has been studied [10, 17, 53] in the static settings for a long time, but little is known under update. The *dynamic descriptive complexity* framework was first introduced in [50], for the expressive power of first-order logic on dynamic database (see [52] for a survey). However, this is different from our goal of designing an efficient index under update. In 2017, two simultaneous papers [14, 35] started to study the computational complexity of conjunctive query evaluation under updates. Both obtain a dichotomy that the minimal index exists for a conjunctive query if and only if it is *q-hierarchical* (e.g., the simplest equi-join over two tables is q-hierarchical). However, for non-q-hierarchical queries, the update time must be at least $\Omega(n^{\frac{1}{2}-\varepsilon})$ for any small constant $\varepsilon > 0$, if we want $O(1)$-delay enumeration. This result is very negative, since q-hierarchical queries are a very restricted class; for example, the matrix multiplication query $\pi_{X,Z} R_1(X,Y) \bowtie R_2(Y,Z)$ and the line-3 join $R_1(X,Y) \bowtie R_2(Y,Z) \bowtie R_3(Z,W)$ are already non-q-hierarchical. Indexes have been designed for certain specific non-q-hierarchical queries, including α-acyclic join [35], triangle join [38] and hierarchical join [39], but none of them achieved the minimal index, due to inherent difficulties with the worst case. Very recently, [56] has proposed an index for acyclic joins under foreign-key constraints and shown a more fine-grained analysis on the update sequence. Meanwhile, there is no previous work on dynamic enumeration for similarity joins with $r > 0$. A naive solution by maintaining all join results explicitly leads to an index of $O(dn^2)$ size that can be built in $O(dn^2)$ time, updated in $O(dn)$ time and support $O(1)$-delay enumeration.

Range search. A widely studied problem related to similarity join is *range searching*: Preprocess a set A of points in $\mathbb{R}^d$ with an index so that for a query range γ (e.g., rectangle, ball, simplex), all points of $A \cap \gamma$ can be reported quickly; alternatively, the points in A may have weights and we may wish to perform a simple aggregation operation (e.g., SUM, MIN, MAX) on the weight of $A \cap \gamma$. 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 join. For the given metric ϕ, let $\mathcal{B}_\phi(x,r)$ be the ball of radius r centered at x. The shape of ball depends on the metric ϕ. A similarity join query can be answered by querying A with ranges $\mathcal{B}_\phi(x,r)$ for all $b \in B$.

There is extensive literature on range searching both in databases and computational geometry. It is beyond the scope of this paper to give an overview of these results; we refer the readers to various surveys [2, 3, 12, 58]. We note that if the query ranges are axis-parallel rectangles, then an index of $\widetilde{O}(n)$ size can be constructed in $\widetilde{O}(n)$ time so that a query can be answered in $\widetilde{O}(1)$ time. Here $\widetilde{O}(f(n))$ means $O(f(n)\text{polylog}(n))$; the exponent of the hidden poly-log factor in $\widetilde{O}(\cdot)$ notation is a constant but may depend on d. If the query ranges are simplices, balls, or more generally semi-algebraic sets, then the best-known index of $\widetilde{O}(n)$ size answers a query in $\widetilde{O}(n^{1-1/d})$ [2, 18]. Techniques exist to improve the query time by increasing the size of the index [43]. Many of the range searching indexes can be modified to handle dynamic updates in the input sets, using the standard dynamization techniques [13].

Notwithstanding a close relationship between range searching and similarity join, the indexes for the former cannot be used for the latter for two main reasons. First, it is too expensive to query A with $\mathcal{B}_\phi(b,r)$ for every $b \in B$ whenever an enumeration query is issued, especially since many queries may return empty set, and it is not clear how to maintain the query results as the input set A changes dynamically. Second, range-searching indexes do not guarantee bounds on the delay between reporting successive points; they only focus on minimizing the overall query time.

In order to support dynamic insertions/deletions in range searching problems, the set of the input items is usually partitioned into non-overlapping groups (for example $O(\log n)$ groups) [13], and for each such group, an independent index is built. There are a lot of known methods on how to efficiently update the indices if specific conditions are met (see [27] for more details). A query is executed by all different indices and the results are merged to return the final answer. In our similarity join query, it is not possible to construct such a scheme over all input points $A \cup B$ and use the known techniques. The reason is that if we have have two sets $P_1 = A_1 \cup B_1$, and $P_2 = A_2 \cup B_2$ with $A_1 \cup A_2 = A$ and $B_1 \cup B_2 = B$ and get the solution to the similarity join query in P_1 and P_2 independently, we cannot efficiently merge the solutions to get an overall solution over $P_1 \cup P_2$. The main problem is that a point in A_1 might be within distance r from a point in B_2; there can be many dependencies between the two groups. Hence, new ideas are needed to overcome this challenge.

Scalable continuous query processing. The range-searching problem asks to answer a query on the current snapshot of the data. There has been some work on scalable continuous query processing, especially in the context of data streams [20, 23, 60] and publish/subscribe [28], 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 (e.g., in the context of range queries, reporting all query ranges that contain the new item) [5, 6]. One can view A as the data stream and $\mathcal{B}_\phi(b,r)$ 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. Thus, it is unclear how continuous-query indexes can be used for similarity join.

1.3 Our results

We present several dynamic indexes for answering similarity joins under ℓ_1, ℓ_2 and ℓ_∞ with $r > 0$. We begin with results assuming that r is known in advance and that d is constant; we then lift these assumptions later.

Exact minimal index for ℓ_1, and ℓ_∞; fixed r. Our first result (Section 4) is a minimal index for similarity join under the ℓ_1 and ℓ_∞ metrics, i.e., it has $\widetilde{O}(n)$ size, $\widetilde{O}(1)$ update time, and $\widetilde{O}(1)$ delay bound. It uses a range tree [11, 25], an index for range searching, but several new ideas are needed to overcome the challenges mentioned above. First, we propose (in Section 3) a general framework for building an index for similarity join and then describe how to

implement it efficiently. There are two main ingredients. First, we store the similarity join pairs *implicitly* so that on one hand, they can be enumerated without probing using every input tuple, and on the other hand, the representation can be updated quickly whenever A or B is updated. Second, we show how to ensure $\widetilde{O}(1)$ delay during enumeration.

Exact and approximate index for ℓ_2; fixed r. Next (in Section 5) we extend these ideas to construct an index for similarity join under the ℓ_2 metric using an index for ball range searching. However, the update time and delay bound are $\widetilde{O}(n^{1-1/d})$. The known lower bounds on ball range searching [1, 22] imply that there is no index for this case with $\widetilde{O}(1)$ update time and delay bound. We therefore shift our attention to ε-approximate similarity join and describe an index of $O(n)$ size with $\widetilde{O}(\varepsilon^{2-4d})$ amortized update time and $\widetilde{O}(\varepsilon^{1-d})$ delay bound. Our main idea is that a ball of radius r can be approximated by a collection of $O(\varepsilon^{1-d})$ hypercubes of size εr and use a variant of quadtrees to build the index.

Approximate index for any ℓ_p; variable r. The above two sets of results assume the similarity threshold r to be known in advance and thus exploit the geometry of the shape of the ball. But in many applications a user may wish to perform enumeration queries with different thresholds. Addressing this issue, our third result (Section 6) is an index for ε-approximate similarity join that works for any ℓ_p metric, and the similarity threshold r can be specified as part of the query. The size of the index is $\widetilde{O}(n)$. The update time is $\widetilde{O}(1)$, and the delay bound is $\widetilde{O}(1)$ under a mild assumption that the *spread* (ratio of the farthest to the closest distance) of $A \cup B$ is polynomially bounded. Our index relies on the *well-separated pair decomposition* (WSPD) of input points, a concept that has been useful to a wide range of applications.

Approximate index for ℓ_1, ℓ_2, Hamming metric in high dimensions. The complexities of these indexes increase exponentially with dimension d, so they are not effective in high dimensions. Our final results (Section 7) are indexes for higher values of d using *locality sensitive hashing* (LSH) [30]. Since LSH is designed for capturing similar join results as much as possible, two challenging questions remain: (1) the delay is not guaranteed; (2) there are many duplicated results. We start with the *uniform assumption* that points are chosen from the universe space randomly, and points to be deleted are chosen from existing ones randomly. Under this assumption, we propose an index of size $\widetilde{O}(nd)$ that can be constructed in $\widetilde{O}(nd)$ time and updated in $\widetilde{O}(d)$ time, while supporting $\widetilde{O}(d)$-delay enumeration for similarity join under any ℓ_p norm, with probability at least $1 - 1/n$.

In general, without uniform assumption, we achieve the following result under the Hamming metric, which can be extended to ℓ_1, ℓ_2 metrics with the same complexity. Let $\varepsilon > 0$ be the approximation ratio and $\rho \leq \frac{1}{1+\varepsilon}$ be the quality of the LSH family in Hamming space. An index of $\widetilde{O}(nd + n^{1+\rho})$ size can be built in $\widetilde{O}(dn^{1+\rho})$ time and updated in $\widetilde{O}(dn^{\rho})$ amortized time, while with high probability supporting $(1 + 2\varepsilon)$-approximate enumeration with $\widetilde{O}(n^{\rho})$ delay. We note that our framework can benefit automatically from any improvement over the LSH and it works for other metrics, such as ℓ_1 and ℓ_2 metrics for which efficient LSH methods exist. Furthermore, our index works even in the case where the similarity threshold r is part of the query. Finally we can show some lower bound by relating similarity join to the *approximate nearest neighbor* (ANN) query. Due to lack of space, we omit the extensions from this version.

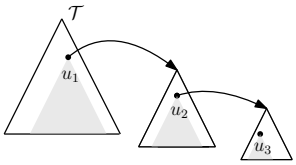

Figure 1: An illustration of range tree $\mathcal{T}$ in $\mathbb{R}^3$.

2 PRELIMINARIES

In this section, we introduce a few useful tools including indexes for range searching and known techniques for transforming a static data structure to a dynamic one.

2.1 Range search indexes

Range tree. A range tree [11] on a set P of points in $\mathbb{R}$ is a balanced binary search tree of $O(\log n)$ height. The points are stored in the leaves of the tree while each internal node v stores the smallest and the largest value, x_v^-, x_v^+, respectively contained in its subtree. The node v is associated with an interval $I_v = [x_v^-, x_v^+]$ and the subset $P_v = I_v \cap P$. The intervals associated with v are called *canonical* intervals. A d-dimensional range tree $\mathcal{T}$ on a set P of points in $\mathbb{R}^d$ is a recursively defined d-level balanced binary search tree. Each level of the tree is a binary search tree on one of the d dimensions. More precisely, the top level of the $\mathcal{T}$ is a 1D range tree on the x_d-coordinates of points in P. Each node v of this tree recursively stores a $(d-1)$-dimensional range tree on P_v^*, the x_d-projection of points in P_v on the hyperplane $x_d = 0$. See [25] for details.

For each level u of the d-th level (bottom level) of $\mathcal{T}$, we associate a d-tuple of $\mathcal{T}$ $\pi(u) = \langle u_1, u_2, \ldots, u_d = u \rangle$, where u_i is the node at the i-th level of $\mathcal{T}$ to which the level-d subtree of $\mathcal{T}$ containing u_i is connected, see Figure1. Recall that each node w of $\mathcal{T}$ at any level is associated with an interval I_w. We associate the rectangle $\square_u = \prod_{j=1}^{d} I_{u_j}$ with the node u. For a given rectangle $b = \prod_{i=1}^{d} [b_i^-, b_i^+]$, a d-level node is called a *canonical node* if for $i = 1, \ldots, d$, $I_{u_i} \subseteq [b_i^-, b_i^+]$ and $I_{p(u_i)} \nsubseteq [b_i^-, b_i^+]$, where $\pi(u) = \langle u_1, \ldots, u_d = u \rangle$ and $p(u_i)$ is the parent of node u_i in the level-i tree rooted at u_{i-1}. Let $\ell(b)$ denote the set of canonical nodes of the rectangle b. The characterization of *canonical* nodes will be critical for our index. A key property of range trees is that for any rectangle b, there are $O(\log^d n)$ canonical nodes in $\mathcal{T}$, and they can be computed in $O(\log^d n)$ time. Furthermore $P \cap b = \bigcup_{u \in \ell(b)} P \cap \square_u$.

Range trees use $\widetilde{O}(n)$ space, can be constructed in $\widetilde{O}(n)$ time, and support $\widetilde{O}(k+1)$-time range-reporting queries [25], where k is the size of the output, and ranges are boxes.

Quadtree. A d-dimensional quadtree [32, 51] over a point set P is a tree data structure in which each node u is associated with a box $\square_u$ in $\mathbb{R}^d$ and each internal node has exactly 2^d children. It recursively subdivide the space into 2^d equal size boxes until a box contains at most one point from P. If $\mathcal{S} = \frac{\max_{x,y \in P} \|x-y\|_2}{\min_{x,y \in P} \|x-y\|_2}$ is the *spread* of point set P, $\mathcal{T}$ can be constructed in $O(n \log \mathcal{S})$ time, it has $O(n \log \mathcal{S})$

space, and $O(\log \mathcal{S})$ height. See [32] for details on quadtrees. Given a sphere $\mathcal{B}$, the quadtree can be used to find $O(\log \mathcal{S} + \varepsilon^{d-1})$ nodes such that the union of their regions completely covers $\mathcal{B} \cap P$ and might cover some parts of $(1+\varepsilon)\mathcal{B}$.

Partition tree. A partition tree on a set P of points in $\mathbb{R}^d$ [18, 42, 57] 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 [18]. 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 [18] 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.

2.2 Dynamization techniques

The following is the standard technique proposed by Bentley and Saxe [13] (see also [27]) for transforming a static index to a dynamic one, for any decomposable query. In this work, we apply it to several indexes for range searching, including range tree, partition tree, and a variation of the quadtree.

Suppose we wish to build an index on a set X of n objects that can be updated efficiently under insertion/deletion. The idea is to partition X into $m = \lceil \log_2 n \rceil$ groups $L_0, \ldots, L_{m-1}$, such that (1) $L_i \subseteq X$; (2) $L_i \cap L_j = \emptyset$; (3) $\bigcup_{i=1,\ldots,m-1} L_i = P$; (4) $|L_i| = 0$ or $|L_i| = 2^i$. A static index $\mathcal{T}_i$ is built for each group L_i. In order to answer a query over the new index we visit each non-empty group L_i and we run the query procedure of the static index $\mathcal{T}_i$. In the end we get the overall result of the query by combining the results. The total query time is $\sum_{i=0}^{l-1} T(2^i) < m \cdot T(n) = O(T(n) \log n)$.

When a new object x is inserted, let i be the smallest index such that $L_i = \emptyset$. Set $L_i = \{x\} \cup \bigcup_{j<i} L_j$. We delete all previous static indexes in $L_0, \ldots, L_{i-1}$, and construct a new static index $\mathcal{T}_i$ for L_i. A key observation is that each object will take part in the construction of $\log n$ indexes. By charging $\sum_{i=0}^{\log n} P(2^i)/2^i$ time to each insertion, we see that the amortized time is $O(\frac{P(n)}{n} \log n)$.

When an object is deleted, we perform *weak deletions*: delete it from the static index $\mathcal{T}_i$ containing it, without changing the structure of $\mathcal{T}_i$. We only require that the cost of a query after the weak deletion is no higher than the cost of a query before it. When more than half number of objects are deleted, we just rebuild everything from scratch. Hence, the amortized cost of a deletion is $O(P(n)/n + D(n))$.

Using the lazy rebuilding technique [48], one can also obtain a new index with the same guarantees in the worst case.

Lemma 2.1 ([13]). *Let X be a set of n objects. For a decomposable query, if there is an static index $\mathcal{T}$ for X which uses $S(n)$ space, can be built in $P(n)$ time, and answers the query in $T(n)$ time, then there is a new index which uses $O(S(n))$ space, can be built in $O(P(n))$ time, can be updated in $O(\frac{P(n)}{n} \log n)$ time for an insertion and $O(\frac{P(n)}{n} + D(n))$ for an deletion where $D(n)$ is the weak deletion time, and answers the query in $O(T(n) \log n)$ time.*

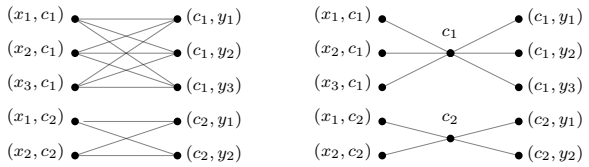

Figure 2: An example of equi-join $R_1(X, C) \bowtie R_2(C, Y)$.

3 FRAMEWORK

All our algorithms for solving similarity join in different settings are based on a common framework. Intuitively, we model the similarity join as a bipartite graph $G' = (A \cup B, E)$, where an edge $(a, b) \in E$ exists if and only if a can be joined with b, i.e., $\phi(a, b) \leq r$. To obtain a data structure for poly-logarithmic delay enumeration, it suffices to find a compact representation of G' with a set $\mathcal{F} = \{(A_1, B_1), (A_2, B_2), \ldots, (A_u, B_u)\}$ 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 $i \neq j$. It remains to store and maintain these bi-cliques efficiently under update.

Take equi-join as an example. After representing input tuples as sets of vertices A and B, there is an edge between $a \in A$ and $b \in B$ if and only if a and b have the same join attribute value. In this way, the bipartite graph itself is a set of edge-disjoint bi-cliques. However, this bipartite representation is not resilient to update, i.e., an insertion or deletion of any tuple may incur $\Omega(n)$ changes in the worst case. A simple idea of getting around this problem is to introduce a middle layer of vertices as C, each representing a distinct join attribute value and there is an edge between $a \in A$ (resp. $b \in B$) and $c \in C$ if and only if the join attribute value of a (resp. b) is equal to c. An example is illustrated in Figure 2.

Let $A_c \subseteq A, B_c \subseteq B$ be the set of vertices in A, B with join attribute value c. In our construction, A_c, B_c are exactly the list of vertices which share an edge with c, thus the biclique $A_c \times B_c \subseteq E$ is implicitly represented by the cross product of two neighbor lists of c. To achieve constant-delay enumeration, we need to maintain a list of *active* values in C, i.e., $c \in C$ is active if and only if c is connected to at least one value in A and one value in B. Obviously, we only visit active values in C, each emitting at least one result. This improves the original representation in the following way: (1) it decreases the space complexity as well as the pre-processing time complexity from $O(n + k)$ to $O(n)$, where k is the output size; (2) the update time is decreased from $O(n)$ to $O(1)$.

We extend this simple example to general similarity joins with $r > 0$. A formal description of the framework is defined as follows.

Definition 3.1 (Tripartite Graph Representation). A tripartite graph $G = (A \cup C \cup B, E_1 \cup E_2)$ where $E_1 \subseteq A \times C$ and $E_2 \subseteq C \times B$ is a representation of the similarity join over A, B under $\phi(\cdot)$ metric with threshold r if for each pair of points $(a, b) \in A \times B$ with $\phi(a, b) \leq r$, there exists unique $c \in C$ such that $(a, c) \in E_1, (c, b) \in E_2$.

It should be noted that G is only conceptual, instead of being stored explicitly. The only extra information we will maintain for G is the *status* of vertices in C. Let A_c, B_c be the set of vertices in A, B connected to $c \in C$, respectively. To support constant-delay enumeration, we define a vertex $c \in C$ as *active* if it witnesses at least one result, i.e., $A_c \neq \emptyset$ and $B_c \neq \emptyset$, and *inactive* otherwise. All active vertices are maintained in $\boldsymbol{C} \subseteq C$.

Lemma 3.2 (δ-delay Enumeration). *In $G = (A \cup C \cup B, E_1 \cup E_2)$ with a set of active vertices in C as $\boldsymbol{C}$, if for any $c \in C$, vertices in A_c (resp. B_c) can be enumerated with δ delay, then all join results can be enumerated with δ delay.*

The proof of Lemma 3.2 directly follows the algorithm: We visit each vertex $c \in \boldsymbol{C}$, and enumerate every pair $(a, b) \in A_c \times B_c$ by a nested-loop over A_c, B_c. It remains to show how to compute and update the active/inactive status of vertices C. The proof of Lemma 3.3 is given in Appendix A.1.

Lemma 3.3 (Status Maintenance). *In $G = (A \cup C \cup B, E_1 \cup E_2)$ with $|C| = O(\eta)$, if (1) for any $c \in C$, whether $A_c = \emptyset$ and $B_c = \emptyset$ can be decided in δ time; (2) for any $c \in C$, A_c and B_c can be counted in ζ time; (3) for any $a \in A$ (reps. $b \in B$), C_a (or C_b) can be reported in λ time, then we have two possible options for $\boldsymbol{C}$ as below:*

- *$\boldsymbol{C}$ can be computed in $O(\eta\delta)$ time and updated in $O(\lambda\delta)$ time;*
- *$\boldsymbol{C}$ can be computed in $O(\eta\zeta)$ time and updated in $O(\lambda)$ time.*

In the remaining of this paper, we will see different instantiations of this framework. The details might be slightly different from what have been described here, due to the inherent difficulties of similarity joins in different settings.

4 SIMILARITY JOIN IN ℓ_1/ℓ_∞ METRICS

In this section, we study similarity joins under the ℓ_1/ℓ_∞ metric, which is captured by a more general problem, namely the *rectangle-containment* problem. Here we are given a set of points as A and a set of orthogonal rectangles as B, with $|A| + |B| = n$. The join result is the set of all pairs $(a, b) \in A \times B$ such that $a \in b$. Note that a similarity join with ℓ_∞ metric is equivalent to a rectangle-containment problem where each side of the rectangles has length $2r$. We first present our results for the ℓ_∞ metric in $\mathbb{R}^d$ for constant d. Then we show that similarity join under ℓ_1 metric in $\mathbb{R}^d$ can be reduced to similarity join under the ℓ_∞ metric in $\mathbb{R}^{d+1}$. For the ease of understanding we illustrate the main idea by an one-dimensional example, which reduces to the *interval-containment* problem.

4.1 Index

We build a d-dimensional dynamic range tree $\mathcal{T}_A$ on the points in A. Figure 3 illustrates a range tree built in 1D. Each rectangle $b = \prod_{i=1}^{d} [b_i^-, b_i^+]$ defines a point $\bar{b} = (b_1^-, b_1^+, \ldots, b_d^-, b_d^+)$ in $\mathbb{R}^{2d}$. [1] Let $\bar{B} = \{\bar{b} \mid b \in B\}$. We also build a $2d$-dimensional dynamic range tree $\mathcal{T}_B$ on $\bar{B}$. The dynamic property of $\mathcal{T}_A$ is maintained by applying the transformation described in Section 2.2. In particular, we partition the points in A into $m = \log |A|$ groups $A^{(1)}, A^{(2)}, \ldots, A^{(m)}$ and construct $\log |A|$ static range trees $\mathcal{T}^{(i)}$, for each $i = 1, 2, \ldots, m$. The construction of $\mathcal{T}_B$ resorts to the standard techniques in [19, 59],

[1] For our similarity join query we have squares of equal side lengths instead of general rectangles. In this case, each square defines a point in $\mathbb{R}^d$.

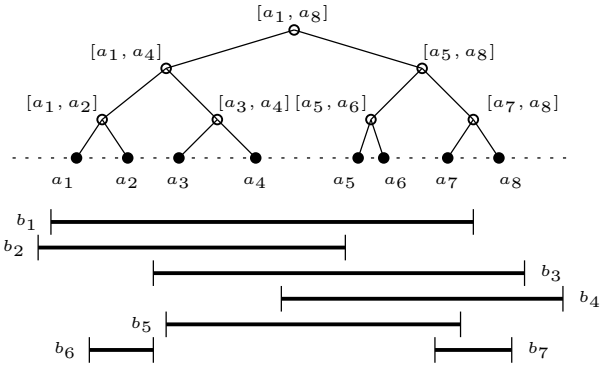

Figure 3: An example of interval-containment query with $A = \{a_1, a_2, \cdots, a_8\}$ and $B = \{b_1, b_2, \cdots, b_7\}$. The range tree $\mathcal{T}_A$ built on A has only one level because $d = 1$, as a balanced binary search tree, where each leaf node stores an input point $a \in A$, and each internal node u stores an interval $[a_i, a_j]$ such that a_i, a_j are the smallest and largest points stored in the subtree rooted at u. Interval b_1, covering all points except a_8, has three canonical nodes, $[a_1, a_4]$, $[a_5, a_6]$ and $[a_7, a_7]$.

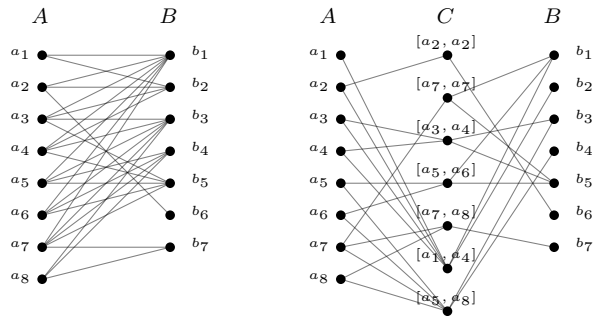

Figure 4: The bipartite (left) and tripartite (right) graph representation of the interval-containment query, where inactive vertices in C are omitted.

which supports $\widetilde{O}(1)$ amortized update time. These two indexes use $\widetilde{O}(n)$ space and can be built in $\widetilde{O}(n)$ time.

Tripartite graph representation. Next, we show how to define the tripartite graph representation $G = (A \cup C \cup B, E_1 \cup E_2)$. Let C be the set of all nodes in the d-th level of $\mathcal{T}_A$. Each node $u \in C$ is associated with a rectangle $\square_u$.For any point $a \in A$, edge $(a, u) \in E_1$ exists if and only if $a \in \square_u$. For any rectangle $b \in B$, edge $(b, u) \in E_2$ exists if and only if u is a *canonical node* of b in $\mathcal{T}_A$ (see Section 2 for the definition of canonical nodes).

Figure 4 illustrates the tripartite graph G built for the interval-containment query in Figure 3. It can be easily checked that G is a valid tripartite graph representation, with proof in Appendix A.2.

Preprocessing. It should be reminded that the only information stored for G is the set of active nodes in C, denoted as $\boldsymbol{C}$. To help define the active/inactive status of nodes in C, we store two additional values $\mu_A(u) = |A_u|$ and $\mu_B(u) = |B_u|$ for each $u \in C$.

To find points in A_u efficiently, the range tree stores and maintains the corresponding points in every node u, so all points can be reported with $O(1)$ delay or counted in $O(1)$ time. To report rectangles in B_u, we issue a range query to $\mathcal{T}_B$ with a rectangle $\square'_u$ [2] and report all rectangles b of B for which u is a canonical node of b. Finally, we define a node $u \in C$ *active* if $\mu_A(u) > 0$ and $\mu_B(u) > 0$, and *inactive* otherwise.

Implied by the property of range tree, each query takes $\widetilde{O}(1)$ time, which is also summarized in Lemma 4.1.

Lemma 4.1. *For any node u in the d-th level range tree of $\mathcal{T}_A$, the next hold: (1) it takes $O(1)$ time to compute $|A_u|$; (2) points in A_u can be enumerated with $O(1)$ delay; (3) it takes $\widetilde{O}(1)$ time to compute $|B_u|$; and (4) points in B_u can be enumerated with $\widetilde{O}(1)$ delay.*

4.2 Update and Enumeration

There are two cases of updates. In the first case when an update happens for rectangles in B, say b, we first insert/delete $\bar{b}$ to/from $\mathcal{T}_B$ in $\widetilde{O}(1)$ time [19, 59]. Then, we visit each group i with $A^{(i)} \neq \emptyset$ and perform the following operations.

Let $C_b \subseteq C$ be the set of canonical nodes of rectangle b. Note that $|C_b| = O(\log^d n)$. Moreover, all nodes in C_b can be found in $O(\log^d n)$ time, by issuing a range query b to the range tree $\mathcal{T}^{(i)}$ [25]. The high-level idea is to update the active/inactive status for each canonical node $u \in C_b$. In particular when a rectangle b is inserted we visit each node $u \in C_b$ and increase $\mu_B(u)$ by 1. If $u \notin \boldsymbol{C}$ and $\mu_A(u) > 0$, we add u to $\boldsymbol{C}$. When a rectangle b is deleted we visit each node $u \in C_b$, and decrease $\mu_B(u)$ by 1. If $u \in \boldsymbol{C}$ and $\mu_B(u) = 0$ after the deletion of b, we remove u from $\boldsymbol{C}$.

In the second case when a update happens for points in A, say a, we resort to the standard techniques in [13, 27]. Let $C_a \subseteq C$ be the set of canonical nodes containing a. Note that $|C_a| = \widetilde{O}(1)$. Moreover, all nodes in C_a can be found in $\widetilde{O}(1)$ time, by issuing a query point a to the range tree $\mathcal{T}^{(i)}$, and for each leaf containing a, taking all nodes up to the root of the d-th level tree of $\mathcal{T}^{(i)}$. We then proceed as follows depending on the type of the update.

Insertion of a. Let i be the smallest index of groups such that $A^{(i)} = \emptyset$. Set $A^{(i)} = \{a\} \cup (\bigcup_{j<i} A^{(j)})$. We first delete the range tree built for $A^{(j)}$ for any $j < i$. Then we build a new range tree for points in $A^{(i)}$. It takes $\widetilde{O}(2^i)$ time to construct $\mathcal{T}^{(i)}$, and for each node u in $\mathcal{T}^{(i)}$, we need $\widetilde{O}(1)$ time to decide whether $B_u = \emptyset$. Following the results in [13, 27] and our discussion in Section 2, the amortized update time for an insertion is $\widetilde{O}(1)$.

Deletion of a. Assume $a \in A^{(i)}$. We only delete point a from all d-level nodes of $\mathcal{T}^{(i)}$ lying on the path from root to the leaf containing a. Then, we visit each node $u \in C_a$ and decrease $\mu_A(u)$ by 1. If $u \in \boldsymbol{C}$ and $\mu_A(u) = 0$ we remove u from $\boldsymbol{C}$. In total, this step takes $\widetilde{O}(1)$ time. When the number of deleted points in A is more than the number of the remaining points in A, we just stop updating the existing index and build the set of $O(\lceil \log_2 |A| \rceil)$ static range trees from scratch. Note that this index can be constructed in $\widetilde{O}(|A|)$ time, so the amortized time for deletion is $\widetilde{O}(1)$.

[2] The condition in Section 2 of finding if node u is a canonical node of a rectangle b can be rewritten as a set of inequalities: for each $i \in \{1, \cdots, d\}$, if $x^-_{u_i} = x^-_{p(u_i)}$, then $b^-_i \leq x^-_{u_i}$ and $x^+_{u_i} \leq b^+_i < x^+_{p(u_i)}$; and if $x^+_{u_i} = x^+_{p(u_i)}$, then $x^-_{p(u_i)} < b^-_i \leq x^-_{u_i}$ and $x^+_{u_i} \leq b^+_1$, which together define an open rectangle $\square'_u$ in $\mathbb{R}^{2d}$. For example assume that $x^-_{u_i} = x^-_{p(u_i)}$ for any $i \in \{1, \cdots, d\}$. Observe that u is a canonical node for b if and only if point $\bar{b} = (b^-_1, b^+_1, \cdots, b^-_d, b^+_d)$ lies in the rectangle defined as $\square'_u = (-\infty, x^-_{u_1}] \times [x^+_{u_1}, x^+_{p(u_1)}) \times \ldots \times (-\infty, x^-_{u_d}] \times [x^+_{u_d}, x^+_{p(u_d)})$.

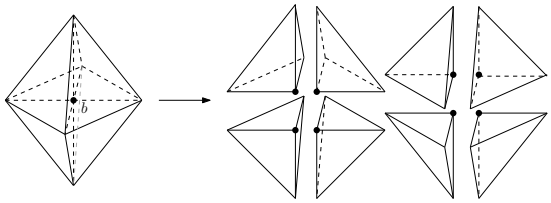

Figure 5: An illustration of ℓ_1 ball in $\mathbb{R}^3$. It is decomposed to $2^d = 8$ types of simplices.

Enumeration. We visit each group i with $A^{(i)} \neq \emptyset$ and apply the enumeration procedure in Section 3 for each group. More specifically, for each node $u \in \boldsymbol{C}$, we just enumerate pair (a, b) for each $a \in A_u$ and $b \in B_u$. By Lemma 4.1, this step has $\widetilde{O}(1)$ delay. By Lemma A.1, all join results will be enumerated without duplication.

Theorem 4.2. *Let A be a set of points and B be a set of rectangles in $\mathbb{R}^d$ with $|A| + |B| = n$. For constant d, an index of $\widetilde{O}(n)$ size can be constructed in $\widetilde{O}(n)$ time and updated in $\widetilde{O}(1)$ amortized time, while supporting $\widetilde{O}(1)$-delay enumeration of rectangle-containment query.*

4.3 Extension to ℓ_1 metric

Given an arbitrary instance of similarity join under the ℓ_1 metric in $\mathbb{R}^d$, we show how to reduce it to an instance of similarity join under ℓ_∞ metric in $\mathbb{R}^{d+1}$.

In the ℓ_1 metric, b can be mapped to an ℓ_1 ball with radius r defined as, $\sum_{j=1}^d |b_j - x_j| \leq r$ (please see Figure 5). Hence, we need to report a pair $(a, b) \in A \times B$ if and only if $\sum_{j=1}^d |b_j - a_j| \leq r$. Let E be the set of all vectors in $\mathbb{R}^d$ with coordinates either 1 or -1. We have $|E| = 2^d$. For each vector $e_i \in E$, we construct an instance of our problem. For each $e_i \in E$, we map each point $a = (a_1, \ldots, a_d) \in A$ to a point $\bar{a}_i = (a_1, \ldots, a_d, \sum_{j=1}^d e_{ij} a_j) \in \mathbb{R}^{d+1}$. Let $\bar{A}_i = \{\bar{a}_i \mid a \in A\}$. In addition, for each point $b = (b_1, b_2, \ldots, b_d) \in B$, we construct the axis-align rectangle in $\mathbb{R}^{d+1}$, $\bar{b}_i$ which is defined as the Cartesian product of the intervals $[b_j, \infty)$ if $e_{ij} = 1$ and $(-\infty, b_j]$ if $e_{ij} = -1$ for each $j = 1, \ldots, d$, and $(-\infty, r + \sum_{j=1}^d e_{ij} b_j]$. Let $\bar{B}_i = \{\bar{b}_i \mid b \in B\}$. For each $e_i \in E$, we construct the index for the ℓ_∞ metric by taking $\bar{A}_i$ as the set of points and $\bar{B}_i$ as the set of rectangles. Equivalently, if b is the ℓ_1 ball of radius r then it can be decomposed to 2^d simplices (Figure 5). Each vector $e_i \in E$ corresponds to a type of simplex of these balls. Each type of simplex has a fixed orientation and can be processed independently by orthogonal range searching in $\mathbb{R}^{d+1}$.

Let $(a, b) \in A \times B$ be an arbitrary pair of points such that $\|a - b\|_1 \leq 1$. Then we show that there is a unique vector in $e_i \in E$ such that $\bar{a}_i \in \bar{b}_i$. Let e_i be the vector such that $\|a - b\|_1 = \sum_{j=1}^d e_{ij}(a_j - b_j)$. Notice that the first d coordinates of $\bar{a}_i$ lie inside the first d intervals defining $\bar{b}_i$: (1) If $e_{ij} = 1$, then $a_j \geq b_j$, i.e., $a_j \in [b_j, +\infty)$; (2) If $e_{ij} = -1$, then $a_j \leq b_j$, i.e., $a_j \in [-\infty, b_j)$. Observe that $\sum_{j=1}^d e_{ij}(a_j - b_j) \leq r$ can be equivalently rewritten as $\sum_{j=1}^d e_{ij} a_j \leq$

$r + \sum_{j=1}^{d} e_{ij} b_j$, or $\sum_{j=1}^{d} e_{ij} a_j \in (-\infty, r + \sum_{j=1}^{d} e_{ij} b_j]$. It is easy to see that for any other vector $e_{i'} \neq e_i$, at least one of the constraints above does not hold. With the same argument, we can show that if $\|a - b\|_1 > r$, then there is no vector $e_i \in E$ such that $\bar{a}_i \in \bar{b}_i$.

Overall, we build $O(2^d) = O(1)$ indexes for the reduced instance. Plugging to Theorem 4.2, we conclude to the following result.

Theorem 4.3. *Let A be a set of points and B be a set of ℓ_1 balls in $\mathbb{R}^d$, with $|A| + |B| = n$. For constant d, an index of $\widetilde{O}(n)$ size can be constructed in $\widetilde{O}(n)$ time and updated in $\widetilde{O}(1)$ amortized time, while supporting $\widetilde{O}(1)$-delay enumeration of the ℓ_1-ball-containment query.*

5 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. Given a threshold r, this is exactly the *sphere-containment* problem. Consider any two points $a = (a_1, \cdots, a_d) \in A$ and $b = (b_1, \cdots, b_d) \in B$. The two points join under the ℓ_2 distance if $(a_1 - b_1)^2 + \cdots + (a_d - b_d)^2 \leq r^2$, or, a lies in the sphere centered at b with radius r. We first present an index for exact sphere-containment join problem and prove a lower bound. Next, we present a more efficient index for approximate sphere-containment join.

5.1 Exact Sphere-Containment

Reduction to halfspace containment. We use the *lifting transformation* [25] to convert an instance of the sphere-containment join problem to the *halfspace-containment* problem in $\mathbb{R}^{d+1}$. The join condition above can be rewritten as

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

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

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

Note that a, b join if and only if $a' \in b'$. Thus, in the following, we study the halfspace-containment problem. Set $A' = \{a' \mid a \in A\}$ and $B' = \{b' \mid b \in B\}$.

Index. 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 index for halfspace-containment join on A, B. The overall structure of the index is the same as for rectangle containment described in Section 4, so we simply highlight the difference.

Instead of constructing a range tree, we construct a dynamic partition tree $\mathcal{T}_A$ 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 [25] 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 index 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 with the update procedure of Section 4, however the query time now on $\mathcal{T}_B$ or $\mathcal{T}_A$ 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 4 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 5.1. *Let A be a set of points and B be a set of halfspaces in $\mathbb{R}^d$ with $|A| + |B| = n$. An index 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 5.1 and the lifting transformation described at the beginning of this section we have the following corollary for the similarity join under the ℓ_2 metric.

Corollary 5.2. *Let A, B be two sets of points in $\mathbb{R}^d$ with $|A| + |B| = n$. An index 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 exact enumeration of similarity join under ℓ_2 metric with $\widetilde{O}(n^{1-\frac{1}{d+1}})$ delay.*

Lower bound. We show a lower bound based on the hardness of sphere reporting problem. Let P be a set of n points in $\mathbb{R}^d$ for $d > 3$ and a parameter r. The sphere reporting problem asks for an index on the points in P, such that given any sphere b report all points of $P \cap b$. If the space is limited to $\widetilde{O}(n)$, there is no known index for answering a sphere reporting query in $\widetilde{O}(k + 1)$ time, where k is the output size, under the pointer machine model for $d \geq 4$ [22].

For any instance of sphere reporting problem, we construct an instance of similarity join over two sets, with $A = \emptyset$ and $B = P$. Given a query sphere $\mathcal{B}(q, r)$, we insert point q to A and issue an enumeration query with threshold r, and then remove q from A. All results enumerated if exist are the results of the sphere reporting problem. If there exists an index 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 5.3. *Let A, B be two sets of points in $\mathbb{R}^d$ for $d > 3$, with $|A| + |B| = n$. If only using $\widetilde{O}(n)$ space, there exists no index that can be updated in $\widetilde{O}(1)$ time, while supporting $\widetilde{O}(1)$-delay enumeration for the sphere-containment query.*

5.2 Approximate Sphere-Containment

The lower bound on the range reporting query rules out the possibility of obtaining an index with the same complexity as in the ℓ_1/ℓ_∞ metric. However, if the accuracy of join results could be sacrificed slightly, we can break the current barrier and achieve a minimal index. In this case, we consider the ε-approximate sphere-containment problem, where all pairs of (a, b) where a lies in the

sphere centered at b with radius r must be reported, together with some pairs of (a', b) where a' lies in the sphere centered at b with radius $(1+\varepsilon)r$. For simplicity, we assume all spheres have radius 1, by scaling all coordinates with r. We present an index with linear space $O(n)$, without any dependency on ε, d, fast update time and small enumeration delay.

Due to lack of space, we only describe the high level idea of our index here and the details can be found in Appendix A.3. Let C be a grid in $\mathbb{R}^d$, where the size of each grid cell is $\varepsilon/\sqrt{d}$. We choose the size of the grid cells to be small enough so that the distance between two points in a grid cell is small, and big enough so that we can bound the number of cells intersected by a unit ball. A grid cell $\square_c \in C$ is *active* if and only if $A \cap \square \neq \emptyset$ and there exists $b \in B$ such that $\phi(b, \square_c) \leq 1+\varepsilon$. Let $\mathcal{C} \subseteq C$ be the set of active grid cells. Intuitively, a cell is active if it contains at least one point from A that is "close" to some points in B.

Note that we need to handle the following query efficiently: Given a ball $\mathcal{B}(x,t) \in \mathbb{R}^d$ centered at a point x with radius t, find all grid cells in C that intersect the ball. For these queries we use the *BBD tree* which is a variation of the quadtree we described in Section 2. A BBD tree has $O(n)$ size and can answer such queries in $\widetilde{O}(\varepsilon^{1-d}+k)$ time, where k is the output size. In particular, it returns all cells within distance t from x and might return cells with distance at most $(1+\varepsilon)t$.

We construct a dynamic BBD tree $\mathcal{T}_A$ for the points in A (implemented by constructing $O(\log|A|)$ static BBD trees as explained in Section 2.2) and a dynamic BBD tree $\mathcal{T}_B$ for points in B [26, 45]. When a new point in $a \in A$ is inserted/deleted we find the cell $c \in C$ such that $a \in A \cap \square_c$ and we check, if needed, if c is still active. When a new point $b \in B$ is inserted/deleted we run a query in $\mathcal{T}_A$ and find all grid cells of C that intersect the ball $\mathcal{B}(b,(1+\varepsilon)^2)$ and change the status of the cells as needed. For the enumeration procedure, we visit every active cell $c \in \mathcal{C}$ and for each point $a \in A \cap \square_c$ we report all points in B within distance $(1+\varepsilon)^3$ from a, using the BBD tree. We note that we use different exponents of $(1+\varepsilon)$ on the threshold distances to ensure that the search procedure of the BBD tree will report all the necessary grid cells or points, due to its error ε. We can ensure ε'-approximate enumeration by setting $\varepsilon' = \varepsilon/6$.

Theorem 5.4. *Let A, B be two sets of points in $\mathbb{R}^d$, with $|A|+|B| = n$. Let $\varepsilon > \frac{1}{n}$ be a small parameter. An index of $O(n)$ space can be constructed in $\widetilde{O}(n\varepsilon^{1-d})$ time and updated in $\widetilde{O}(\varepsilon^{2-4d})$ amortized time, while supporting ε-approximate enumeration of similarity join under ℓ_2 metric with $\widetilde{O}(\varepsilon^{1-d})$ delay.*

6 SIMILARITY JOIN WITH UNKNOWN THRESHOLD

The previous solutions assumed the threshold r to be fixed in advance. However, after receiving join results of multiple queries, users may wish to vary the similarity threshold r. A more challenging question is, whether we can design an index oblivious to r that supports efficient updates and that answers enumeration queries for a given threshold r. All previous data structures fail in this setting, since they are designed to capture the *geometry* of similarity join with r. In the worst case, one has to build a new index from scratch for every incoming query.

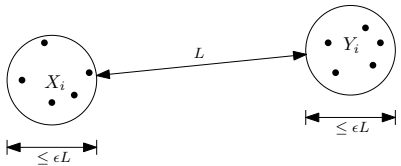

Figure 6: An illustration of well-separated pair in the WSPD.

In this section, we give a unified data structure to solve the approximate similarity join with varying r. For simplicity, we focus on the ℓ_2 metric; the algorithm extends to any ℓ_p norm in an obvious manner. We first introduce the concept of well separated pair decomposition (WSPD) [16, 32, 34], which will be crucial for our index. The high-level idea of WSPD is to divide input points into a few clusters such that for each pair of clusters, one can determine if there is anything joinable under given r by simply checking their bounding boxes. Indexing these pairs of point sets by distance of boxes allows these candidate pairs to be joined efficiently. In addition, we make a mild assumption on the input tuples: the *spread* is bounded by a polynomial in terms of n, i.e, $\frac{\max_{x,y\in A\cup B}\|x-y\|_2}{\min_{x,y\in A\cup B}\|x-y\|_2} = O(\text{poly}(n))$.

6.1 Well-Separated Pair Decomposition

WSPD can be built on a quadtree (that we defined in Section 2) for storing input points. Recall that if the spread of points P is bounded by a polynomial with respect to n the height of the quadtree is $O(\log n)$. For a set of points P, let $\text{diam}(P) = \max_{x,y\in P}\|x-y\|_2$. For two sets of points P_1, P_2, let $\phi(P_1,P_2) = \min_{x\in P_1, y\in P_2}\|x-y\|_2$.

Definition 6.1 (Well-separated Pair Decomposition). Given a set P of n points in $\mathbb{R}^d$ and a parameter $0 < \varepsilon < \frac{1}{2}$, a set of s pairs $W = \{(X_1,Y_1),(X_2,Y_2),\cdots,(X_s,Y_s)\}$ is a ε-WSPD if the following hold: (1) for any $i \in \{1,2,\cdots,s\}$, $X_i, Y_i \subseteq P$ and $X_i \cap Y_i = \emptyset$; (2) for each pair of points $x, y \in P$, there exists a unique pair $(X_j, Y_j) \in W$ such that $x \in X_j$ and $y \in Y_j$; (3) $s = O(\frac{n}{\varepsilon^d})$ and (4) for any $i \in \{1,2,\cdots,s\}$, $\max\{\text{diam}(X_i), \text{diam}(Y_i)\} \leq \varepsilon \cdot \phi(X_i, Y_i)$.

Each pair (X_i, Y_i) is *well-separated* in the sense that any two points across X_i, Y_i have their distance being at least $\frac{1}{\varepsilon}$-factor of the largest distance between points inside X_i or Y_i; see Figure 6. In [32, 34], a compressed quadtree (quadtree for bounded spread as in our case) is used to construct an ε-WSPD efficiently. In particular, a ε-WSPD W can be constructed in $O(n\log n + \varepsilon^{-d}n)$ time such that each pair $(X_i, Y_i) \in W$ is a pair of nodes in a quadtree over the points set of points P. The sets of points corresponding to X_i, Y_i are denoted as $P \cap \square_{X_i}, P \cap \square_{Y_i}$ respectively. Beyond the points in P, some stronger properties indeed hold on each pair of nodes in $\mathcal{T}$, which corresponds to a pair in W. More specifically, for each pair $(X_i, Y_i) \in W$, the following hold: (1) $\square_{X_i} \cap \square_{Y_i} = \emptyset$; (2) $\max\{\text{diam}(\square_{X_i}), \text{diam}(\square_{Y_i})\} \leq \varepsilon\phi(\square_{X_i}, \square_{Y_i})$. Another important property that will be frequently used on W and $\mathcal{T}$ is stated as below.

Lemma 6.2 ([32]). *Given a set P of n points with $O(\text{poly}(n))$ spread, and its WSPD W built on a quadtree $\mathcal{T}$, $\mathcal{T}$ has height $O(\log n)$ and every node in $\mathcal{T}$ participates in $\widetilde{O}(1)$ pairs of W.*

It is known that the ε-WSPD can be maintained under insertion/deletion of points in $\widetilde{O}(1)$ time [15, 29]. In the following, we

assume that ε is a small constant, so the complexity dependency on ε is hidden in the big-Oh notation.

6.2 Index, Update and Enumeration

Index. First we construct two dynamic range trees $\mathcal{T}_A, \mathcal{T}_B$ for the points in A, B respectively, following [19, 59]. We also construct a quadtree $\mathcal{T}$ on all input points in $A \cup B$ for a $\frac{\varepsilon}{2}$-WSPD, denoted as $W = \{(X_1, Y_1), \ldots, (X_s, Y_s)\}$.

Tripartite graph representation. The tripartite graph $G = (A \cup C \cup B, E_1 \cup E_2)$ is defined as follows. Set $C = W$. Consider an arbitrary pair $c = (X_i, Y_i) \in W$. For a point $a \in A$, an edge $(a, c) \in E_1$ exists if $a \in X_i \cup Y_i$. Similar definition applies for points in B.

Preprocessing. Without knowing the value of r in advance, there is no way to determine whether there is any join result witnessed by one specific pair in W. Below, we slightly relax the notion of *active*/*inactive* pair.

For any pair $(X_i, Y_i) \in W$, if $X_i \cup Y_i$ only contain points from the same relation, no join results will be enumerated from (X_i, Y_i). A pair $(X_i, Y_i) \in W$ is *active* for join result if $(X_i \cup Y_i) \cap A \neq \emptyset$ and $(X_i \cup Y_i) \cap B \neq \emptyset$, and *inactive* otherwise. Obviously, no join results will be enumerated from an inactive pair. Note that whether a pair (X_i, Y_i) is active can be decided[3] in $\widetilde{O}(1)$ time. Moreover, for each active pair (X_i, Y_i), we store the distance $\phi(\square_{X_i}, \square_{Y_i})$.[4] All active pairs are maintained in $\boldsymbol{C}$ as a balanced search tree, in increasing order of $\phi(\square_{X_i}, \square_{Y_i})$. Overall, this step takes $\widetilde{O}(n)$ time.

Update. After inserting or deleting an input point, the quadtree $\mathcal{T}$ and W can be updated in $\widetilde{O}(1)$ time, following the standard techniques in [15, 29]. The range trees $\mathcal{T}_A, \mathcal{T}_B$ can also be updated in $\widetilde{O}(1)$ time [19, 59]. Next, we will explain how to update $\boldsymbol{C}$. Without loss of generality, assume the update is from A, say a.

Let W_1, W_2 be the set of pairs deleted from and inserted to W respectively. For each pair $(X_i, Y_i) \in W_1$, if $(X_i, Y_i) \in \boldsymbol{C}$, we remove it from $\boldsymbol{C}$. For each pair $(X_i, Y_i) \in W_2$, we first check whether it is active. If yes, we compute $\phi(\square_{X_i}, \square_{Y_i})$ and insert this pair to $\boldsymbol{C}$.

It remains to check whether an old pair of W should be added to or removed from $\boldsymbol{C}$, after the update of a. Without loss of generality, assume a is inserted (the deletion can be handled symmetrically). Let $u \in T$ be the new leaf node containing a. Consider an arbitrary node v lying on the path from root to u. For each pair including v in W, say (v, t), if $(v, t) \notin \boldsymbol{C}$, we check whether (v, t) turns active after inserting a. If yes, we compute the value of $\phi(\square_v, \square_t)$ and insert it in $\boldsymbol{C}$. We prove that $\boldsymbol{C}$ can be updated in $\widetilde{O}(1)$ time; the detailed proof is given in Appendix A.4.

Enumeration. We start an in-order traversal of the balanced search tree for $\boldsymbol{C}$. Consider an arbitrary pair $(X_i, Y_i) \in \boldsymbol{C}$ visited. We check whether $\phi(\square_{X_i}, \square_{Y_i}) \leq r$; if yes, we enumerate all pairs (a, b) in the cross product $(A \cap \square_{X_i}) \times (B \cap \square_{Y_i})$ as well as $(A \cap \square_{Y_i}) \times (B \cap \square_{X_i})$. If $\phi(\square_{X_i}, \square_{Y_i}) > r$, we stop the enumeration procedure.

Note that points in $A \cap \square_{X_i}$ can be enumerated by issuing a range query $\square_{X_i}$ to the range tree $\mathcal{T}_A$, with $\widetilde{O}(1)$ delay. The similar argument applies for $B \cap \square_{Y_i}$, $A \cap \square_{Y_i}$ and $B \cap \square_{X_i}$. Moreover, observe that $|\boldsymbol{C}| \leq |W| = O(\varepsilon^{-d} n) = O(n)$, as long as ε is a constant. The in-order traversal of the balanced binary search tree has a delay $O(\log |\boldsymbol{C}|) = \widetilde{O}(1)$. Overall, this enumeration has $\widetilde{O}(1)$ delay.

Moreover, all pairs of points enumerated from this index form an $(1+\varepsilon)$-approximation of the join result. The proof of correctness is provided in the Appendix A.4.

Theorem 6.3. *Let A, B be two sets of points in $\mathbb{R}^d$ for some constant d, with $O(poly(n))$ spread and $|A| + |B| = n$. Let $0 < \varepsilon \leq 1$ be a small constant. An index of $\widetilde{O}(n)$ space can be constructed in $\widetilde{O}(n)$ time and updated in $\widetilde{O}(1)$ time, while supporting ε-approximate enumeration for similarity join under any ℓ_p metric with $\widetilde{O}(1)$ delay, for any query similarity threshold r.*

7 SIMILARITY JOIN IN HIGH DIMENSIONS

So far, we have treated the dimension d as a constant. In this section we describe an index for approximate similarity join using the *locality sensitive hashing* (LSH) technique so that the dependency on dimension is a small polynomial in d, by removing the exponent dependency on d from the hidden poly-log factor. For simplicity, we describe our index assuming that r is fixed, and in the end we extend it to the case where r is also part of the similarity join query.

For $\varepsilon > 0$, $1 \geq p_1 > p_2 > 0$, recall that a family 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, we have (1) $\Pr[h(x) = h(y)] \geq p_1$ if $\phi(x, y) \leq r$; and (2) $\Pr[h(x) = h(y)] \leq p_2$ if $\phi(x, y) \geq (1 + \varepsilon)r$. The quality of a hash function family is measured by $\rho = \frac{\ln p_1}{\ln p_2} < 1$, which is bounded by a constant that depends only on ε, but not the dimensionality; and $\rho = \frac{1}{1+\varepsilon}$ for many common distance functions [7, 24, 30, 33]. In a standard hash family H, by concatenating multiple hash functions independently chosen from H, we can make p_1 and p_2 arbitrarily small, whereas ρ is kept fixed.

The essence of LSH is to hash "similar" points in P into the same buckets with high probability. To use LSH for similarity join, the standard way is to (1) hash input tuples into buckets; (2) probe each bucket and check, for each pair of points falling into the same bucket, whether their distance is smaller than r; and (3) report every pair of tuples within distance r found in (2). However, two challenges arise here in answering enumeration query under worst-case delay guarantee. Firstly, without any knowledge of false positive results inside each bucket, there can be a huge delays in checking every pair of points before finding a true result. One may wonder whether this can be resolved by building additional data structure inside each bucket. But, hash functions only decrease the number of points inside each bucket, instead of dimensionality, which does not help solve our original problem. Secondly, one pair of points within distance r may collide under multiple hash functions, so an additional deduplication step is necessary in the enumeration.

We start with a strong *uniform* assumption that points to be inserted or deleted are randomly chosen; this randomness help resolve above challenges. Then, we present our main result for the general case without any assumption on input tuples.

For simplicity, we consider the Hamming space $\mathbb{H}^d$, and our techniques can be extended to other metrics. The LSH family H consists of all hash functions h_i, which samples the i-th bit of the

[3] We check if $A \cap \square_{X_i} \neq \emptyset$ (by issuing a range query $\square_{X_i}$ to $\mathcal{T}_A$) and $B \cap \square_{Y_i} \neq \emptyset$ (by issuing a range query $\square_{Y_i}$ to $\mathcal{T}_B$), or $A \cap \square_{Y_i} \neq \emptyset$ and $B \cap \square_{X_i} \neq \emptyset$.

[4] The value of $\phi(\square_{X_i}, \square_{Y_i})$ can be computed in time that depends only on dimension d by computing the distance between every pair of corners and faces of two cubes $\square_{X_i}, \square_{Y_i}$. This is still a constant as long as d is a constant.

input point. Hence, we have $p_1 = 1 - r/d$ and $p_2 = 1 - r(1+\varepsilon)/d$ [32, 36]. We sample k hash functions randomly with replacement, and concatenate them as a new composite hash function g. This new hash family H' is $(r, (1+\epsilon)r, p_1^k, p_2^k)$-sensitive, with fixed ρ.

7.1 With Uniform Assumption

Under this strong assumption, LSH technique can be used with a slight modification.

Index. Set $k = O(\log n)$. We just choose $\tau = \frac{3}{p_1^k} \ln n$ hash functions randomly and independently from H', denoted as $g_1, g_2, \cdots, g_\tau$.

Tripartite graph representation. Next, we define the tripartite graph representation $G = (A \cup C \cup B, E_1 \cup E_2)$. Let C be the set of all buckets over all τ hash functions. There is an edge between tuple $a \in A$ and bucket $\square \in C$ from hash function g_i if $g_i(a) = \square$, which can be decided in $O(\log n)$ time. Let $A_\square, B_\square$ be the set of points from A, B falling into bucket $\square$, respectively. A nice property on $A_\square$ and $B_\square$ is stated in the following lemma, which is directly followed by the balls-into-bins result.

Lemma 7.1. *If input points are randomly and uniformly chosen from the domain universe, by choosing $k = O(\log n)$, with probability at least $1 - \frac{1}{n}$, every bucket receives $O(\log n / \log\log n)$ points.*

As the number of points colliding in each bucket can be bounded by $O(\log n)$, it is affordable to check all pairs of points inside one bucket in $O(\log^2 n)$ time, thus resolving the challenge (1). Moreover, we introduce a variable $\square_{\text{out}}$ for each bucket $\square \in C$ indicating the number of the pair of tuples within distance r colliding inside $\square$. Obviously, a bucket $\square$ is *active* if $\square_{\text{out}} > 0$, and *inactive* otherwise. All active buckets are maintained in $\boldsymbol{C} \subseteq C$, in increasing order of the index of the hash function it comes from.

Update. Assume one point $a \in A$ is inserted. For each hash function g_i where $i \in \{1, 2, \cdots, \tau\}$, we first compute $g_i(a)$ and insert it into the bucket $\square \in C$ if $g_i(a) = \square$. Then, we count the number of pair of points $(a, b \in \square \cap B)$ with $\phi(a, b) \le r$, and add this quantity to $\square_{\text{out}}$. The case of deletion can be handled similarly.

Enumeration. Note that we need an an additional de-duplication step in enumerating pairs of points. We visit each bucket in $\boldsymbol{C}$ sequentially. Consider the bucket $\square$ from the hash function g_i. For each pair of points $(a \in \square \cap A, b \in \square \cap B)$ with $\phi(a, b) \le r$, we check whether a, b have ever collided into any bucket previously. If there exists no index $j < i$ such that $g_j(a) = g_j(b)$, we report it. Then, we need to notify every bucket which also witnesses (a, b) but comes after $\square$. More specifically, for every $j > i$, if $g_j(a) = g_j(b)$ in bucket $\square'$, we decrease $\square'_{\text{out}}$ by 1, and remove $\square'$ from $\boldsymbol{C}$ if $\square'_{\text{out}}$ becomes 0. The pseudocode is given in Appendix A.6.

Theorem 7.2. *Let A, B be two sets of points in $\mathbb{R}^d$, with $|A| + |B| = n$, and ε, r be positive parameters. Under uniform assumption, an index of $\widetilde{O}(nd)$ size can be constructed in $\widetilde{O}(nd)$ time and updated in $\widetilde{O}(d)$ time, while with probability $1 - 2/n$ supporting exact enumeration of similarity join with $\widetilde{O}(d)$ delay.*

7.2 Without Uniform Assumption

In general, without this uniform assumption, we need to explore more properties of the LSH family for an efficient index. Our key insight is that after checking some pairs of points in one bucket (the specific numbers of pairs will be determined later), we can safely skip the bucket, since with high probability any join result missed in this bucket will be found in another one. In this way, we avoid spending too much time in one bucket before finding any join result. Recall that the LSH uses τ random hash functions of the family H'. The new hash family of LSH [32, 36] is $(r, (1+\varepsilon)r, 1-(1-p_1^k)^\tau, 1-(1-p_1^k)^\tau)$-sensitive. From [32, 36] by choosing $k = O(\log_{1/p_2^k} n)$ and $\tau = 4\left\lceil 1/p_1^k \right\rceil$ we have $\tau = O(n^{1/\rho})$ and $k = O(\log n)$ for $\rho = \frac{1}{1+\varepsilon}$ and some nice properties are proved for answering ANN query, which will be the starting point of our index. The first part of the next lemma follows from [36]. We prove the second part in Appendix A.5.

Lemma 7.3. *For a set P of n points in hamming space $\mathbb{H}^d$ and a distance threshold r, by choosing $k = O(\log n)$ and $\tau = O(n^\rho)$, the following two properties hold for a point $p \in P$:*

- *Let q be a point such that $\phi(p, q) \le r$. With probability at least $3/4$, the point p will collide in some bucket with point q;*
- *For any $x \in \mathbb{H}^d$ and $t > 0$, let $\overline{\mathcal{B}}(x, t) = \mathbb{H}^d \setminus \mathcal{B}(x, t)$.With probability at least $3/4$, the number of points in $P \cap \overline{\mathcal{B}}(p, (1+\varepsilon)r)$ colliding in any bucket with p is at most $M = O(\log n)$.*

7.2.1 Index

Recall that $\rho = \frac{1}{1+\varepsilon}, \tau = O(n^\rho), k = O(\log n)$, and $M = O(\log n)$. We construct $m = \frac{3}{\log(4/3)} \log n$ copies of the index as $\mathbb{I}_1, \mathbb{I}_2, \cdots, \mathbb{I}_m$. An important property on this set of indexes is stated by the following lemma. All proofs in this section are given in Appendix A.5.

Lemma 7.4. *With probability at least $1 - 1/n$, for any pair of points $(a, b) \in A \times B$ with $\phi(a, b) \le r$, there exists an index $\mathbb{I}_j$ such that*

- *a, b will collide in some bucket of $\mathbb{I}_j$;*
- *The number of points in $A \cap \overline{\mathcal{B}}(a, (1+\epsilon)r)$ and $B \cap \overline{\mathcal{B}}(a, (1+\epsilon)r)$ colliding with a in any bucket of $\mathbb{I}_j$ is at most M;*
- *The number of points in $A \cap \overline{\mathcal{B}}(b, (1+\epsilon)r)$ and $B \cap \overline{\mathcal{B}}(b, (1+\epsilon)r)$ colliding with b in any bucket of $\mathbb{I}_j$ is at most M;*

Lemma 7.4 implies that for each join result (a, b), there exists at least one bucket such that a, b collides, and the number of "noisy" points (with distance more than $(1+\epsilon)r$ from a or b) inside this bucket is also bounded by $M = O(\log n)$. We denote such a bucket as *proxy bucket* for (a, b). Later, we will see that our enumeration phase only reports each join result in one of its proxy buckets. This guarantees the completeness of query results, but de-duplication is still necessary if a pair of points has more than one proxy buckets.

The tripartite graph $G = (A \cup C \cup B, E_1 \cup E_2)$ can be defined similarly as the previous case with uniform assumption. The only difference comes from the definition of active/inactive bucket.

Preprocessing. We pick arbitrary M points from $A_\square, B_\square$ respectively and denote them as $A_\boxtimes, B_\boxtimes$. If $|A_\square| \le M$, then $A_\boxtimes = A_\square$. The similar applies for $B_\boxtimes$. A bucket $\square$ is *active* if there exists a pair of points $(a, b) \in A_\boxtimes \times B_\boxtimes$ within distance $2(1+\varepsilon)r$, and *inactive* otherwise. Note that each active bucket is stored with a pair of points within distance $2(1+\varepsilon)r$, as its *representative*. All active buckets are maintained in the list $\boldsymbol{C} \subseteq C$. Later we will see that join results are only enumerated from buckets in $\boldsymbol{C}$.

Lemma 7.5. *For any bucket $\square$ and $a \in A_\square$, if there are more than M points in $B_\square$ with distance larger than $2(1+\varepsilon)r$ from a, $\square$ is not a proxy bucket for any pair of points $(a, b \in B_\square)$ within distance r.*

Lemma 7.6. *For any bucket $\square$, if there exist M points from A as $A_\boxtimes$ and M points from B as $B_\boxtimes$, such that all pairs of points in $A_\boxtimes \times B_\boxtimes$ have their distances larger than $2(1+\varepsilon)r$, then $\square$ is not a proxy bucket for every pair of points $(a', b') \in A_\square \times B_\square$ within distance r.*

7.2.2 Update

We handle insertion and deletion separately. Without loss of generality, assume the update comes from A, say a. All details are illustrated with pseudocodes in Appendix A.6.

Insertion of a. We compute $g(a)$ for each chosen hash function g. Assume $\square$ is the bucket with hash value $g(a)$. We first insert a to $A_\square$. If $\square$ is inactive, we check the distances between a and arbitrary M points in $B_\square$. Note that if there are fewer than M points in $B_\square$, then we need to compute distances between a and all points in $B_\square$. If a point b is found with $\phi(a, b) \le 2(1+\epsilon)r$, we add $\square$ to $\boldsymbol{C}$ and store (a, b) as its representative.

Deletion of a. Similarly, we compute $g(a)$ for each chosen hash function g. Assume $\square$ is the bucket with hash value $g(a)$. We first remove a from $\square_A$. If $\square$ is active and a participates in the representative pair of $\square$, we check whether $\square$ is active or inactive after the deletion of a, and update its representative if necessary.

When there are $n/2$ updates, we just reconstruct the entire index from scratch.

7.2.3 Enumeration

The high-level idea is to enumerate the representative pair of points for each bucket in $\boldsymbol{C}$. Moreover, results are enumerated in groups of value a. Assume a representative pair (a, b) is found in a bucket $\square \in \boldsymbol{C}$. Now, it is going to enumerate all results associated with a.

Initially, all buckets containing a are maintained in $\boldsymbol{C}(a) \subseteq \boldsymbol{C}$. Algorithm 1 visits every bucket $\square \in \boldsymbol{C}(a)$ and starts to check the distances between a and points in $B_\square$. Each time when a pair (a, b) within distance $2(1+\varepsilon)r$ is found, it just reports this pair and calls the procedure Deduplicate on (a, b) (details will be given later). If there are more than M points far away from a, we just stop enumerating results with point a in this bucket, which is safe by Lemma 7.5, and remove the bucket[5] $\square$ from $\boldsymbol{C}(a)$. Once the enumeration is finished on a, i.e., when $\boldsymbol{C}(a)$ becomes empty, it can be easily checked that a has been removed from all buckets.

Next, we explain more details on the de-duplication step presented as Algorithm 2. Once a pair of points (a, b) within distance $2(1+\varepsilon)r$ is reported, Algorithm 2 goes over all buckets witnessing the collision of a, b, and marks b with $X(\square, a)$ to avoid repeated enumeration (line 2). Moreover, for any bucket $\square$ with $a \in A_\square$ and $b \in B_\square$, if (a, b) is also its representative pair, Algorithm 2 performs more update for $\square$. Algorithm 2 first needs to decide whether $\square$ is still an active bucket for a by checking the distances between a and M points unmarked by a in $B_\square$. If such a pair within distance $2(1+\varepsilon)r$ is found, it will set this pair as new representative for $\square$ (line 5-7). Otherwise, it is safe to skip all results with point a in this bucket (line 9-10), implied by Lemma 7.5. In this case, it needs to further update a new representative pair for $\square$ (line 11-15). Moreover, if no representative pair can be found (line 16-18), it is safe to skip all results with bucket $\square$, implied by Lemma 7.6.

Algorithm 1: EnumerateLSH

```
1  while C ≠ ∅ do
2      (a, b) ← the representative pair of any bucket in C;
3      C(a) ← {□ ∈ C : a ∈ A_□};
4      while C(a) ≠ ∅ do
5          Pick one bucket □ ∈ C(a); i ← 0;
6          foreach b ∈ B_□ − X(□, a) do
7              if φ(a, b) ≤ 2(1 + ε)r then
8                  Emit(a, b);
9                  Deduplicate(a, b);
10             else
11                 i ← i + 1;
12                 if i > M then break;
13         A_□ ← A_□ − {a};
14         C(a) ← C(a) − {□};
```

Algorithm 2: Deduplicate(a, b)

```
1  foreach □ ∈ C with a ∈ A_□ and b ∈ B_□ do
2      X(□, a) ← X(□, a) ∪ {b};
3      if (a, b) is the representative pair of □ then
4          B_⊠ ← M arbitrary points in B_□ − X(□, a);
5          if there is b' ∈ B_⊠ with φ(a, b') ≤ 2(1 + ε)r then
6              Set (a, b') as new representative of □;
7          else
8              C(a) ← C(a) − {□};
9              A_□ ← A_□ − {a};
10             A_⊠, B_⊠ ← M arbitrary points in A_□, B_□;
11             if there is a pair (a', b') ∈ A_⊠ × B_⊠ with
                φ(a', b') ≤ 2(1 + ε)r then
12                 Set (a', b') as new representative of □;
13             else
14                 C ← C − {□};
```

For any bucket $\square$, we can maintain the points in $A_\square, B_\square, X(\square, a)$ in balanced binary search trees, so that points in any set can be listed or moved to a different set with $O(\log n)$ delay. Moreover, to avoid conflicts with the markers made by different enumeration queries, we generate them randomly and delete old values by lazy updates [27, 47, 48] after finding new pairs to report.

In Appendix A.5 we show that our index supports $(1+2\varepsilon)$-approximate enumeration. We next analyze the complexity of index. The size of the index is $\widetilde{O}(dn + nk\tau) = \widetilde{O}(dn + n^{1+\rho})$. The insertion time is $\widetilde{O}(d\tau M) = \widetilde{O}(dn^\rho)$. Using that, we can bound the construction time of this index as $\widetilde{O}(dn^{1+\rho})$. The deletion time is $\widetilde{O}(d\tau M^2) = \widetilde{O}(dn^\rho)$. The delay is bounded by $\widetilde{O}(d\tau M^2) = \widetilde{O}(dn^\rho)$ since after reporting a pair (a, b), we may visit $\widetilde{O}(\tau)$ buckets and spend $O(M^2)$ time for each in updating the representative pair. Putting everything together, we conclude to the next theorem.

[5] In the enumeration phase, the "remove" always means conceptually marked, instead of changing the index itself.

Theorem 7.7. *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 $\rho = \frac{1}{1+\varepsilon}$, an index of $\widetilde{O}(dn+n^{1+\rho})$ size can be constructed in $\widetilde{O}(dn^{1+\rho})$ time, and updated in $\widetilde{O}(dn^{\rho})$ amortized time, while supporting $(1+2\varepsilon)$-approximate similarity join enumeration over A, B with $\widetilde{O}(dn^{\rho})$ delay.*

Due to lack of space, in Appendix A.7 we extend our index to work when r is part of the query. Furthermore, we discuss how our index extends to the ℓ_1, ℓ_2 metric and we give a lower bound reducing the ANN problem to our query.

REFERENCES

[1] P. Afshani. Improved pointer machine and i/o lower bounds for simplex range reporting and related problems. In *Proceedings of the twenty-eighth annual symposium on Computational geometry*, pages 339–346, 2012.

[2] P. K. Agarwal. Simplex range searching and its variants: A review. In *A Journey Through Discrete Mathematics*, pages 1–30. Springer, 2017.

[3] P. K. Agarwal, J. Erickson, et al. Geometric range searching and its relatives. *Contemporary Mathematics*, 223:1–56, 1999.

[4] P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13(4):325–345, 1995.

[5] P. K. Agarwal, J. Xie, J. Yang, and H. Yu. Monitoring continuous band-join queries over dynamic data. In *Intern. Symp. on Alg. and Comp.*, pages 349–359, 2005.

[6] P. K. Agarwal, J. Xie, J. Yang, and H. Yu. Scalable continuous query processing by tracking hotspots. In *Proc. inter. conf. on Very large data bases*, pages 31–42, 2006.

[7] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06)*, pages 459–468. IEEE, 2006.

[8] S. Arya and D. M. Mount. Approximate range searching. *Computational Geometry*, 17(3-4):135–152, 2000.

[9] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[10] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

[11] J. L. Bentley. Decomposable searching problems. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1978.

[12] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409, 1979.

[13] J. L. Bentley and J. B. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *J. algorithms*, 1(4):301–358, 1980.

[14] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 303–318, 2017.

[15] P. B. Callahan. *Dealing with higher dimensions: the well-separated pair decomposition and its applications.* PhD thesis, Citeseer, 1995.

[16] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)*, 42(1):67–90, 1995.

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

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

[19] T. M. Chan and K. Tsakalidis. Dynamic orthogonal range searching on the ram, revisited. In *33rd International Symposium on Computational Geometry (SoCG)*, volume 77, pages 28:1–28:13, 2017.

[20] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proc. of Intern. Conf. on Very Large Databases*, pages 203–214. Elsevier, 2002.

[21] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Inter. Conf. on Data Eng. (ICDE)*, pages 5–5. IEEE, 2006.

[22] B. Chazelle and B. Rosenberg. Simplex range reporting on a pointer machine. *Computational Geometry*, 5(5):237–247, 1996.

[23] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390, 2000.

[24] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[25] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 1997.

[26] D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 296–305, 2005.

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

[28] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 115–126, 2001.

[29] J. Fischer and S. Har-Peled. Dynamic well-separated pair decomposition made easy. In *CCCG*, volume 5, pages 235–238, 2005.

[30] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[31] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. Inter. Conf. on Man. of Data (SIGMOD)*, page 157–166, 1993.

[32] S. Har-Peled. *Geometric approximation algorithms.* Number 173. American Mathematical Soc., 2011.

[33] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of computing*, 8(1):321–350, 2012.

[34] S. Har-Peled and M. Mendel. Fast construction of nets in low-dimensional metrics and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006.

[35] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1259–1274, 2017.

[36] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[37] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Transactions on Database Systems (TODS)*, 33(2):1–38, 2008.

[38] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Counting triangles under updates in worst-case optimal time. *arXiv preprint arXiv:1804.02780*, 2018.

[39] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 375–392, 2020.

[40] C. Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 87–98, 2010.

[41] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278, 2014.

[42] J. Matoušek. Efficient partition trees. *Disc. & Comp. Geom.*, 8(3):315–334, 1992.

[43] J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete & Computational Geometry*, 10(2):157–182, 1993.

[44] R. Motwani, A. Naor, and R. Panigrahi. Lower bounds on locality sensitive hashing. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 154–157, 2006.

[45] D. M. Mount and E. Park. A dynamic data structure for approximate range searching. In *Proceedings of the twenty-sixth annual symposium on Computational geometry*, pages 247–256, 2010.

[46] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 International Conference on Management of Data*, pages 511–526, 2016.

[47] M. H. Overmars. *The design of dynamic data structures*, volume 156. Springer Science & Business Media, 1987.

[48] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, 1981.

[49] R. Paredes and N. Reyes. Solving similarity joins and range queries in metric spaces with the list of twin clusters. *Journal of Disc. Alg.*, 7(1):18–35, 2009.

[50] S. Patnaik and N. Immerman. Dyn-fo: A parallel, dynamic complexity class. *Journal of Computer and System Sciences*, 55(2):199–209, 1997.

[51] H. Samet. Spatial data structures: Quadtree, octrees and other hierarchical methods, 1989.

[52] T. Schwentick and T. Zeume. Dynamic complexity: recent updates. *ACM SIGLOG News*, 3(2):30–52, 2016.

[53] L. Segoufin. Enumerating with constant delay the answers to a query. In *Proc. of the 16th Inter. Conf. on Database Theory*, pages 10–20, 2013.

[54] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *26th Inter. Conf. on Data Eng. (ICDE)*, pages 892–903. IEEE, 2010.

[55] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering? an adaptive framework for similarity join and search. In *Intern. Conf. Manag. of Data*, pages 85–96, 2012.

[56] Q. Wang and K. Yi. Maintaining acyclic foreign-key joins under updates. In *Proc. of the ACM SIGMOD Intern. Conf. on Manag. of Data*, pages 1225–1239, 2020.

[57] D. E. Willard. Polygon retrieval. *SIAM Journal on Comp.*, 11(1):149–165, 1982.

[58] D. E. Willard. Applications of range query theory to relational data base join and selection operations. *Journal of Comp. and Syst. Sciences*, 52(1):157–169, 1996.

[59] D. E. Willard and G. S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM (JACM)*, 32(3):597–617, 1985.

[60] K.-L. Wu, S.-K. Chen, and P. S. Yu. Interval query indexing for efficient stream processing. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 88–97, 2004.

A APPENDIX

A.1 Missing proofs in Section 3

Proof of Lemma 3.2. By the definition of tripartite graph representation, the union of $A_c \times B_c$'s is exactly the result of the similarity join over A, B. Then, it suffices to enumerate the cross product of $A_c \times B_c$ for each $c \in \boldsymbol{C}$. The enumeration algorithm is straightforward. We visit vertices in $\boldsymbol{C}$ sequentially; and for each $c \in \boldsymbol{C}$, we just perform a simple nested-loop over A_c, B_c to emit every pair $(a, b) \in A_c \times B_c$ with δ delay. □

Proof of Lemma 3.3. We will mention two different ways of computing and maintaining $\boldsymbol{C}$.

Firstly, we use the properties (1) and (3). For each vertex $c \in C$, we check whether $A_c \neq \emptyset$ and $B_c \neq \emptyset$; if not, add c to $\boldsymbol{C}$. When an update occurs, say a point a is inserted to A (the case of deletion is similar), the high-level idea is to check if some vertex in C_a turns active after the insertion of a. We first find C_a by a reporting query. For each $c \in C_a$, if $c \notin \boldsymbol{C}$, we check whether $B_c \neq \emptyset$; if yes, add c to $\boldsymbol{C}$. This option computes $\boldsymbol{C}$ in $O(|C| \cdot \delta) = O(\eta \cdot \delta)$ time and updates $\boldsymbol{C}$ in $O(|C_a| \cdot \delta) = O(\lambda \cdot \delta)$ time.

Secondly, we use the properties (2) and (3). For each vertex $c \in C$, we spend $2 \cdot \lambda$ time to compute $|A_c|$ and $|B_c|$ and store them as $\mu_A(c), \mu_B(c)$ respectively. If $|A_c| > 0$ and $|B_c| > 0$, we add c to $\boldsymbol{C}$. The total time is $O(|C| \cdot \lambda) = O(\eta \cdot \zeta)$. When an update occurs, say a point a is inserted to A (the case of deletion is similar), the high-level idea is to check if some vertex in C_a turns active after the insertion of a. We first find C_a by a reporting query. For each $c \in C_a$, we increase $\mu_A(c)$ by 1. If $c \in \boldsymbol{C}$, we check whether $\mu_B(c) > 0$; if yes, add c to $\boldsymbol{C}$. The update takes $O(|C_a|) = O(\lambda)$ time. □

A.2 Missing Proofs in Section 4

Lemma A.1. *G is a valid tripartite graph representation.*

Proof. We need to show that for each pair $a, b \in A \times B$ such that a lies inside the rectangle b then there is a unique $c \in C$ such that $(a, c) \in E_1$ and $(c, b) \in E_2$. The canonical nodes of a rectangle in a range tree form a partition of points in $O(\log^d n)$ groups. From the definition of the set of canonical nodes in $\mathcal{T}_A$ for rectangle b, there exists a unique d-dimensional node u of $\mathcal{T}_A$ such that u is a canonical node of b and $a \in \square_u$. □

A.3 Approximate Similarity Join under ℓ_2 metric

In this section, we use the *BBD* tree to derive our new index. We start by introducing BBD trees.

BBD tree. A balanced box decomposition (or BBD for short) tree [8, 9] is a variant of the *quadtree* [32, 51]. A BBD tree $\mathcal{T}$ on a set P of n points in $\mathbb{R}^d$, is a balanced binary tree of size $O(n)$ and height $O(\log n)$ with n leaves. It can be constructed in $O(n \log n)$ time. Every box has a bounded aspect ratio and every $2d$ levels of descent in the tree, the size of the associated cells decreased by at least a factor of $1/2$, where the size of a cell is defined as the length of its longest side. Each node is associated with a box in $\mathbb{R}^d$ or the set theoretic difference of two nested boxes $\square_v^O, \square_v^I$ (rectangular annulus) with $\square_v^I$ possibly being empty. Hence each box v is associated with an annular region $\square_v = \square_v^O \setminus \square_v^I$.

Given a sphere B of radius 1, the BBD tree in [8, 9] returns $O(\log n + \varepsilon^{d-1})$ nodes such that the union of their regions completely covers $B \cap P$ and might cover some parts of $(1+\varepsilon)B$. Given $x \in \mathbb{R}^d, r \geq 0, \varepsilon > 0$, the BBD tree $\mathcal{T}$ can be

- all points of P within distance r from x will be returned;
- no point of P farther than $(1+\epsilon)r$ from x will be returned;

Points that are at distance from x between r and $(1+\varepsilon)r$ from x may or may not be reported. The time complexity of such a query is $O(\log n + \varepsilon^{1-d} + k)$, where k is the output size.

In [26, 45], the dynamic version of BBD tree was proposed, with $\widetilde{O}(1)$ time for maintenance under per update.

Index We build two dynamic BBD trees for points in A, B as $\mathcal{T}_A, \mathcal{T}_B$ separately. The dynamic property of $\mathcal{T}_A$ will be maintained by applying the transformation described in Section 2. In particular, we partition the points in A into $m = \log |A|$ groups $A^{(1)}, A^{(2)}, \ldots, A^{(m)}$ and construct $\log |A|$ static BBD trees $\mathcal{T}^{(i)}$, for each $i = 1, 2, \ldots, m$. The construction of $\mathcal{T}_B$ will use the standard techniques in [26, 45], which supports $\widetilde{O}(1)$ amortized update time. These two indexes uses $\widetilde{O}(n)$ space and can be built in $\widetilde{O}(n)$ time.

Tripartite graph representation. Next, we will show how to define a tripartite graph representation $G = (A \cup C \cup B, E_1 \cup E_2)$. For a node u in $\mathcal{T}_A$, let $p(u)$ be the parent of u, $\square_u$ be the box associated with u and size($\square_u$) be the largest side of the box $\square_u$. Define

$$C = \{u \in \mathcal{T}_A \mid \text{size}(\square_u) \leq \varepsilon/\sqrt{d} < \text{size}(\square_{p(u)})\},$$

i.e., the set of nodes that have size smaller than $\varepsilon/\sqrt{d}$ but their parents have size strictly greater than $\varepsilon/\sqrt{d}$. Notice that C defines a grid on the point set A with small enough cells. The BBD tree is actually used in our case to efficiently derive the cells of C that are contained in a query ball.

The high-level idea why we choose these nodes from $\mathcal{T}_A$ as C can be explained as follows. When a new ball is inserted or deleted from B, we need to find all nodes in C whose union cover the ball (because of the error ε of the BBD tree, their union can also contain areas out of the ball). The size of nodes in C is small enough so that the area out of the ball covered will be bounded, and also big enough so that for any unit ball $b \in B$, the number of nodes in C to cover b is bounded.

Consider an arbitrary node $u \in C$. For any point $a \in A$, edge $(a, u) \in E_1$ exists if $a \in \square_u$. However, for any point $b \in B$, edge (b, u) is not constructed in a deterministic way; instead, it obeys the following rules due to the approximation nature of range queries over BBD tree: if $\|x - b\|_2 \leq (1+\varepsilon)^2$ for some corner x of $\square_u$, (b, u) exists; if $\|x - b\|_2 > (1+\varepsilon)^3$, (b, u) does not exist; otherwise, (b, u) may or may not exist.

Preprocessing. Let $Q_\varepsilon(x, 1+\varepsilon)$ be an approximate range query on a BBD tree, where the query range is a ball of center x and radius $1+\varepsilon$, and the error parameter is ε. We assume that $Q_\varepsilon(x, 1+\varepsilon)$ returns the set of nodes of the BBD tree whose union cover the ball $\mathcal{B}(x, 1+\varepsilon)$.

We define the inactive/active status of nodes in C. Node $u \in C$ is *active* if $\square_u \cap A \neq \emptyset$, and there exists a corner x of $\square_u$ such that running the query $Q_\varepsilon(x, 1+\varepsilon)$ in $\mathcal{T}_B$ it returns at least one point from B, and *inactive* otherwise. Notice that if a node $u \in C$ is active

then $\square_u$ contains at least a point in A and there exists at least one corner x of $\square_u$ and a point $b \in B$ with $\|x-b\|_2 \leq (1+\varepsilon)^2$. All active nodes are maintained in $\boldsymbol{C}$.

Lemma A.2. *For any node $u \in C$, if u is active, for any point $a \in A \cap \square_u$, there exists a point $b \in B$ such that $\|a-b\|_2 \leq (1+\varepsilon)^2+\varepsilon$; otherwise, there exists no pair of points ($a \in \square_v \cap A, b \in B$) such that $\|a-b\|_2 \leq 1$.*

Proof. If v is active, there exists a corner x of $\square_v$ such that $Q_\varepsilon(x, 1+\varepsilon) \neq \emptyset$. Let $b \in Q_\varepsilon(x, 1+\varepsilon)$ be an arbitrary point. Moreover, $\|x-b\|_2 \leq (1+\varepsilon)^2$. As size$(\square_v) \leq \varepsilon/\sqrt{d}$, the distance between any pair of points inside $\square_v$ is at most ϵ. Then, for any point $a \in \square_v \cap A^{(i)}$, $\|a-b\|_2 \leq \|x-b\|_2 + \|x-a\|_2 \leq (1+\varepsilon)^2 + \varepsilon$.

If v is inactive, assume there is a pair of points $a \in \square_v \cap A_i, b \in B$ with $\|a-b\|_2 \leq 1$. As size$(\square_v) \leq \varepsilon/\sqrt{d}$, any pair of points in $\square_v$ has its distance bounded by ϵ. Let x be an arbitrary corner of $\square_v$. We will have $\|x-b\|_2 \leq \|a-b\|_2 + \|x-a\|_2 \leq 1+\varepsilon$, contradicting the fact that v is inactive, i.e., no corner of $\square_v$ has a near neighbor in B within distance $1+\varepsilon$. □

Note that the status of u can be decided in $O(\varepsilon^{1-d} \log n)$ time, so this step takes $O(n\varepsilon^{1-d} \log n)$ time in total.

A.3.1 Update

Note that the dynamic BBD tree $\mathcal{T}_B$ can be updated in $O(\log n)$ time, following the standard techniques in [26, 45]. Next, we focus on the update on $\mathcal{T}$ and G.

In the first case when an update happens for points in A, say a, we follow the standard techniques in [13, 27].

Insertion of a. Let i be the smallest index such that $A^{(i)} = \emptyset$. We delete all BBD trees for every group $j \leq i$ and create a new BBD tree $\mathcal{T}^{(i)}$ on points in $\{a\} \cup (\bigcup_{j \leq i} A_j)$. Following the observations in [13, 27], the amortized cost is $O(\varepsilon^{1-d} \log n)$.

Deletion of a. Assume $a \in A^{(i)}$. Let u be the leaf node in $\mathcal{T}^{(i)}$ containing a. We just remove a from u. Let $v \in C$ be the node lying on the path from root to u. If v is active and $\square_v \cap A^{(i)} = \emptyset$, i.e., there is no other point lying inside $\square_u$ except a, we remove v from $\boldsymbol{C}$. Note that $\mathcal{T}^{(i)}$ is static, so no operations are performed on the tree structure.

When the number of deleted points in A is more than the number of the remaining points in A, we just stop updating the existing index and build the set of $O(\lceil \log |A| \rceil)$ static BBD trees from scratch. Note that this index can be constructed in $O(n \cdot \varepsilon^{1-d} \log n)$ time, the amortized cost is $O(\varepsilon^{1-d} \log n)$.

In the second case when an update comes from B, say b, we perform the following procedure for each group $A^{(i)}$ if $A^{(i)} \neq \emptyset$. The high-level idea is to find all nodes in C whose status would be changed by b, and update them if necessary. Let U be the set of nodes returned by issuing a range query $Q_\varepsilon(b, (1+\varepsilon)^2)$ to $\mathcal{T}^{(i)}$. The following Lemma states that every node in U is located "above" the nodes in C.

Lemma A.3. *For each $u \in U$, size$(\square_{p(u)}) \geq \varepsilon/\sqrt{d}$.*

Proof. Let $v \in C$ be a node visited by the search procedure $Q_\varepsilon(b, (1+\varepsilon)^2)$ in $\mathcal{T}^{(i)}$. We argue that the search algorithm will not continue searching at the children of v. The search procedure in [8, 9] first checks if $\square_v \subset \mathcal{B}(b, (1+\varepsilon)^3)$. If that is true then the search procedure does not consider the children of v and $v \in U$. If $\square_v \not\subset \mathcal{B}(b, (1+\varepsilon)^3)$ then it checks if $\square_v \cap \mathcal{B}(b, (1+\varepsilon)^2) = \emptyset$. If it is true then it does not consider the children of v and $v \notin U$. Hence, the only remaining case to consider is when $\square_v \not\subset \mathcal{B}(b, (1+\varepsilon)^3)$ and $\square_v \cap \mathcal{B}(b, (1+\varepsilon)^2) \neq \emptyset$. However, this case will not happen because size$(\square_v) \leq \varepsilon/\sqrt{d}$: if $\square_v \not\subset \mathcal{B}(b, (1+\varepsilon)^3)$ then $\square_v$ is so small that it does not intersect $\mathcal{B}(b, (1+\varepsilon)^2)$. In particular, any pair of points in $\square_v$ has distance at most ε and the distance of $\mathcal{B}(b, (1+\varepsilon)^2)$ and $\mathcal{B}(b, (1+\varepsilon)^3)$ is more than ε, so if $\square_v \not\subset \mathcal{B}(b, (1+\varepsilon)^3)$ then $\square_v \cap \mathcal{B}(b, (1+\varepsilon)^2) = \emptyset$. □

For each $u \in U$, we find all nodes in C which reside in the subtree of $\mathcal{T}^{(i)}$ rooted at u, denoted as V_u. Note that we will check the status for each node $v \in \bigcup_{u \in U} V_u$. To see why this procedure is correct, it suffices to show that every node of C whose status will be changed by the update of b must be included by $\bigcup_{u \in U} V_u$. In particular, it suffices to show that when a point b is deleted from B, we will visit all nodes $u \in C$ such that there exists some corner x of $\square_u$ with $\|x-b\|_2 \leq (1+\varepsilon)^2$. Let u be such a node in C. Notice that $\square_u \cap \mathcal{B}(b, (1+\varepsilon)^2) \neq \emptyset$. As size$(\square_u) \leq \varepsilon/\sqrt{d}$, we observe that $\square_u \subset \mathcal{B}(b, (1+\varepsilon)^3)$. Hence the search procedure for the query range $Q_\varepsilon(b, (1+\varepsilon)^2)$ in $\mathcal{T}^{(i)}$ will always find a node w that lies in the path from the root to u such that $\square_w \subseteq \mathcal{B}(b, (1+\varepsilon)^3)$. Notice that $\square_u \subseteq \square_w$. Hence, the search procedure always visits node u.

Insertion of b. If $\square_v \cap A^{(i)} \neq \emptyset$ and v is inactive, we compute the distances between all corners of $\square_v$ with b. If any one of them has distance smaller than $1+\varepsilon$ from b, we add v to $\boldsymbol{C}$.

Deletion of b. If v is active and $\square_v \cap A^{(i)} \neq \emptyset$, we issue a range query $Q_\varepsilon(x, 1+\varepsilon)$ to $\mathcal{T}_B$ for every corner x of $\square_v$. If all queries have empty results, we remove v from $\boldsymbol{C}$.

We next analyze the complexity of this update procedure with b. A first observation on the number of nodes we have checked is stated in Lemma A.4.

Lemma A.4. $|\bigcup_{u \in U} V_u| = O\left(\varepsilon^{-2d}(\log 2^i + \varepsilon^{1-d})\right)$.

Proof. From [8, 9], $|U| = O(\log 2^i + \varepsilon^{1-d})$. From each node $u \in U$ we need to bound $|V_u|$. Notice that $size(\square_u) \leq 2(1+\varepsilon)^3$. Recall, that every $2d$ levels of the BBD tree $\mathcal{T}^{(i)}$ the size of the cells are decreased by at least a factor of $1/2$. Hence, $2d \log(\frac{2(1+\varepsilon)^3\sqrt{d}}{\varepsilon})$ levels below the level of u the size of the cells will be at most $\varepsilon/\sqrt{d}$. So, $|V_u| = O\left(2^{2d \log(\frac{2(1+\varepsilon)^3\sqrt{d}}{\varepsilon})}\right) = O\left((\frac{2(1+\varepsilon)^3\sqrt{d}}{\varepsilon})^{2d}\right)$. □

The range query over $\mathcal{T}^{(i)}$ takes $O(\log 2^i + \varepsilon^{1-d})$ time, following [8, 9]. Over all groups, it takes $O(\log n(\log n + \varepsilon^{1-d}))$ to find all nodes in U. From Lemma A.4 the insertion time is $O(\varepsilon^{-2d} \log n(\log n + \varepsilon^{1-d}))$. Moreover, each range query over $\mathcal{T}_B$ takes $O(\varepsilon^{1-d} \log n)$ time. Implied by the Lemma A.4 we will execute $O(\varepsilon^{-2d} \log n(\log n + \varepsilon^{1-d}))$ such queries when we delete a point in B so the overall update time for deleting a point is $O(\varepsilon^{1-3d} \log^2 n(\log n + \varepsilon^{1-d}))$.

A.3.2 Enumeration

We visit each group $A^{(i)}$ if $A^{(i)} \neq \emptyset$. For any node $u \in \boldsymbol{C}$, we invoke the following procedure. For each point $a \in A^{(i)} \cap \square_u$, let B_a be the result of issuing a ball query $\boldsymbol{Q}_\varepsilon(a, (1+\varepsilon)^3)$ to $\mathcal{T}_B$. We just report (a, b) for all b in B_a.

Each range reporting query over $\mathcal{T}_B$ takes $O(\varepsilon^{1-d} \log n)$ time first, and then all results can be reported with $O(1)$ delay. The delay of this enumeration phase is $O(\varepsilon^{1-d} \log n)$.

Lemma A.5. *The index supports $(4\varepsilon + 6\varepsilon^2 + 4\varepsilon^3 + \varepsilon^4)$-approximate enumeration.*

Proof. It suffices to show that all pairs of points within distance r must be reported, and any pair of points within distance strictly larger than $(1+\varepsilon)^4$ will not be reported.

Consider an arbitrary pair of points $(a, b) \in A \times B$ with $\|a-b\|_2 \leq 1$. Let $u \in C_\varepsilon^i$ be the unique node such that $a \in \square_u$. Observe that there exists a corner x of $\square_u$ such that $\|x - b\|_2 \leq \|a - b\|_2 + \|x - a\|_2 \leq 1+\varepsilon$, thus u is active. So the enumeration query will visit node u and we will run a query $\boldsymbol{Q}_\varepsilon(a, (1+\varepsilon)^3)$ in $\mathcal{T}_B$ finding the point b. When u is active, for a node $u \in C$ then we will always return at least a pair $(a \in \square_u, b \in B)$ since there exists b with $\|x - b\|_2 \leq (1+\varepsilon)^2$, where x is a corner of $\square_u$ and $\|a - b\|_2 \leq (1+\varepsilon)^2 + \varepsilon < (1+\varepsilon)^3$.

Finally, from the definition, a reporting query $\boldsymbol{Q}_\varepsilon(a, (1+\varepsilon)^3)$ in $\mathcal{T}_B$ will never return a point b with $\|a - b\|_2 > (1+\varepsilon)^4$. □

A.4 Missing proofs in Section 6

Lemma A.6. *$\boldsymbol{C}$ can be updated in $\widetilde{O}(1)$ time.*

Proof of Lemma A.6. It has been shown in [15, 29]that $|W_1| + |W_2| = O(\text{polylog}n)$. The update for W_1, W_2 takes $O(\text{polylog}n)$ time in total, since removing a pair from the balanced search tree of W' takes $O(\log |W'|) = O(\log n)$ time, and checking whether a pair is active or not takes $O(\text{polylog}n)$ time. From Lemma 6.2, each node $u \in \mathcal{T}$ participates in $O(\log n)$ pairs in W, and each point is included by $O(\log n)$ nodes in $\mathcal{T}$. In this way, there are at most $O(\log^2 n)$ pairs to be checked in an addition step (no matter for insertion and deletion), and each check takes $\widetilde{O}(\text{polylog}n)$ time. Overall, the total time for updating W' is $\widetilde{O}(1)$. □

Proof of correctness of the enumeration algorithm in Section 6. It suffices to show that every pair of points within distance r must be reported, while any pair of points with distance larger than $(1+\varepsilon)r$ will not be reported.

Let $(a, b) \in A \times B$ be a pair with $\|a - b\|_2 \leq r$. Implied by Definition 6.1, there exists a unique pair (X_i, Y_i) such that $a \in X_i$ and $b \in Y_i$ or $a \in Y_i$ and $b \in X_i$, so $(X_i, Y_i) \in W'$. Moreover, $\phi(\square_{X_i}, \square_{Y_i}) \leq \|a - b\|_2 \leq r$. Thus, all join results across X_i, Y_i will be reported, including (a, b).

Next, let (X_i, Y_i) be a pair that is found by the enumeration procedure in W', with $\phi(\square_{X_i}, \square_{Y_i}) \leq r$. For any pair of points $x \in \square_{X_i}, y \in \square_{Y_i}$, their distance can be bounded as

$$\begin{aligned}\|x - y\|_2 \leq& \phi(\square_{X_i}, \square_{Y_i}) + \text{diam}(\square_{X_j}) + \text{diam}(\square_{Y_j}) \\ \leq& (1 + 2 \cdot \frac{\varepsilon}{2}) \cdot \phi(\square_{X_i}, \square_{Y_i}) \leq (1+\varepsilon)r,\end{aligned}$$

thus any pair of points with distance strictly larger than $(1+\varepsilon)r$ will not be reported.

A.5 Missing Proofs in Section 7

Proof of Theorem 7.2. We first prove the correctness of the algorithm. It can be easily checked that any pair of points with their distance larger than r will not be emitted. Consider any pair of points (a, b) within distance r. Let i be the smallest index such that $g_i(a) = g_i(b)$ in bucket $\square$. In the algorithm, (a, b) will be reported by $\square$ and not by any bucket later. Thus, each join result will be enumerated at most once without duplication.

In the case of hamming distance, we have $k = \log_2 n$ and $p_1^k = (1 - \frac{r}{d})^{\log n} \in [1/e, 1]$ since $d/r > \log n$ by padding some zeros at the end of all points[6], thus $\tau = 3 \cdot 1/p_1^k \cdot \ln n = \widetilde{O}(1)$.

We next analyze the complexity of our index. It can be built in $O(nk\tau)$ time with $O(nk\tau)$ space, since there are n vertices in $A \cup B$, at most $O(n\tau)$ non-empty buckets in C, and each tuple in $A \cup B$ is incident to exactly l buckets in C. With the same argument, it takes $O(nkl)$ time for construct the tripartite graph representation. Moreover, it takes $O(\sum_\square |A_\square| \cdot |B_\square|)$ time for computing the quantity $\square_{\text{out}}$ for all buckets, which can be further bounded by

$$\sum_\square |A_\square| \cdot |B_\square| < n \cdot \max_\square(|A_\square| + |B_\square|) = O(n \log n)$$

implied by Lemma 7.1.

Consider any bucket $\square$ from hash function g_j. If the algorithm visits it during the enumeration, at least one pair of points within distance r will be emitted, which has not been emitted by any bucket from hash function h_i for $i < j$. Checking all pairs of points inside any bucket takes at most $O((d + kl) \cdot \max_\square |A_\square| \cdot |B_\square|)$ time, where it takes $O(d)$ time to compute the distance between any pair of points and kl time for checking whether this pair has been emitted before or marking buckets which also witnesses this pair later. Thus, the delay between any two consecutive pairs of results is bounded by $O((d+kl) \cdot \max_\square |A_\square| \cdot |B_\square|)$, which is $\widetilde{O}(d)$ under the uniform assumption.

Moreover, for each such pair of tuples within distance r, it will be reported by any hash function with probability at least p_1^k. The probability that they do not collide on any one of hash function is at most $(1 - p_1^k)^{3 \cdot 1/p_1^k \cdot \ln n} \leq 1/n^3$. As there are at most n^2 such pairs of tuples, the probability that any one of them is not reported by our index is at most $1/n$.

By a union bound, the probability that either uniform assumption fails or one join result is not reported is at most $\frac{1}{n} + \frac{1}{n} = \frac{2}{n}$. Thus, the result holds with probability at least $1 - \frac{2}{n}$. □

Proof of Lemma 7.3. The first part of the lemma follows from [36].

Next we focus on the second part. Let h be one of the τ hash functions and let $q \in P \cap \overline{\mathcal{B}}(p, (1+\varepsilon)r)$. From [36] we have that $\Pr[h(p) = h(q)] \leq 1/n$. Let $X(q)$ be a random variable which is 1 if $h(p) = h(q)$, and 0, otherwise. Let $X = \sum_{q \in P \cap \overline{\mathcal{B}}(p,(1+\varepsilon)r)} X(q)$. We have $\mu = \mathbf{E}[X] \leq 1$. We have $\Pr[X \geq 2\log(4\tau) + 1] \leq \Pr[X \geq (1 + \frac{2\log(4\tau)}{\mu})\mu]$. From Chernoff inequality[7] we have $\Pr[X \geq (1 + \frac{2\log(4\tau)}{\mu})\mu] \leq e^{-\frac{2^2 \log^2(4\tau)/\mu}{2 + 2\log(4\tau)/\mu}}$. Notice that $2 + 2\log(4\tau)/\mu \leq$

[6] Similar assumption was made in the original paper [30] of nearest neighbor search in Hamming distance.

[7] We use the version of Chernoff inequality where $\Pr[X \geq (1+\delta)\mu] \leq e^{-\delta^2\mu/(2+\delta)}$, for $\delta \geq 0$

$4\log(4\tau)/\mu$, so $\Pr[X \geq 2\log(4\tau)+1] \leq e^{-\frac{\log^2(4\tau)/\mu}{\log(4\tau)/\mu}} = e^{-\log(4\tau)} = \frac{1}{4\tau}$.

Let G_i be the event which is true if point p has at most $2\log(4\tau)+1$ conflicts with points in $P \cap \overline{\mathcal{B}}(p,(1+\varepsilon)r)$ at hash function h_i. $\Pr[\cap_{i=1}^{\tau} G_i] = 1 - \Pr[\bigcup_{i=1}^{\tau} \bar{G}_i] \geq 1 - \frac{\tau}{4\tau} = 3/4$. □

Proof of Lemma 7.4. Consider any pair of points ($a \in A, b \in B$) within distance r and an arbitrary index constructed as described above. From Lemma 7.3, with probability at least $1/4$ there exists a bucket in the index that contains both a, b and the number of collisions of both a and b (with the rest of the points in $A \cup B$) is bounded by M.

Let F_j be the event that is true if there is a bucket in $\mathbb{I}_j$ that witnesses the collision of a, b and the number of collisions of both a, b is bounded by M. Since F_i, F_j are independent for $i \neq j$, we have $\Pr[\bar{F}_1 \cap \ldots \cap \bar{F}_C] = \Pr[\bar{F}_1] \cdot \ldots \cdot \Pr[\bar{F}_C] \leq (3/4)^{\frac{3}{\log(4/3)}\log n} \leq 1/n^3$. Let Z be the number of pairs with distance at most r. We have $Z \leq n^2$. Let G_i be the event which is true if for the j-th pair of points a', b' with distance at most r, there is at least a copy of the index such that there exists a bucket that contains both a', b' and the number of collisions of both a, b is bounded by M. Then $\Pr[G_1 \cap \ldots \cap G_Z] = 1 - \Pr[\bar{G}_1 \cup \ldots \cup \bar{G}_Z] \geq 1 - \Pr[\bar{G}_1] - \ldots - \Pr[\bar{G}_Z] \geq 1 - n^2/n^3 \geq 1 - 1/n$. Hence, with high probability, for any pair $a \in A, b \in B$ with distance at most r there will be at least one bucket in the index such that, both a, b are contained in the bucket and the number of collisions of both a, b in the bucket is bounded by M. □

Proof of Lemma 7.6. Consider a bucket □ in which all pairs of points in $A_{\boxtimes} \times B_{\boxtimes}$ have their distances larger than $2(1+\varepsilon)r$. From Lemma 7.5 follows that □ is not a proxy bucket for any pair ($a \in A_{\boxtimes}, b \in B_{\square}$) and ($a \in A_{\square}, b \in B_{\boxtimes}$). It remains to check the pairs $a' \in A_{\square} \setminus A_{\boxtimes}, b' \in B_{\square}$ and $a' \in A_{\square}, b' \in B_{\square} \setminus B_{\boxtimes}$. Let (a', b') be such a pair within r. W.l.o.g., assume $a' \in A_{\square} \setminus A_{\boxtimes}$ and $b' \in B_{\square}$ (the case with $a' \in A_{\square}$ and $b' \in B_{\square} \setminus B_{\boxtimes}$ is similar).

The case for $b' \in B_{\boxtimes}$ is directly implied by Lemma 7.5. It remains to consider the case that $b' \in B_{\square} \setminus B_{\boxtimes}$. If all points in $B_{\boxtimes}$ are at least $(1+\varepsilon)r$ away from a', or all points in $A_{\boxtimes}$ are at least $(1+\varepsilon)r$ away from a', then □ is not a proxy bucket for (a', b') by Lemma 7.4. Otherwise, there must exist at least one point $a'' \in A_{\boxtimes}$ as well as $b'' \in B_{\boxtimes}$ such that $\phi(a', a'') \leq (1+\varepsilon)r$ and $\phi(a', b'') \leq (1+\varepsilon)r$. By triangle inequality,

$$\phi(a'', b'') \leq \phi(a', a'') + \phi(a', b'') \leq 2(1+\varepsilon)r$$

Thus, $(a'', b'') \in A_{\boxtimes} \times B_{\boxtimes}$ is such a pair within distance $2(1+\varepsilon)r$, coming to a contradiction. □

Lemma A.7. *The index supports $(1+2\varepsilon)$-approximate enumeration.*

Proof of Lemma A.7. It can be easily checked that any pair of points with distance far more than $2(1+\varepsilon)r$ will not be enumerated. Also, each result is reported at most once by Algorithm 2. Next, we will show that with high probability, all pairs of points within distance r are reported. Consider any pair of points (a, b) within distance r. Implied by Lemma 7.4, there must exist a proxy bucket □ for (a, b). Observe that there exists no subset of M points from $A_{\square}$ as $A_{\boxtimes}$ and subset of M points from $B_{\square}$ as $B_{\boxtimes}$, where all pairs of points in $A_{\boxtimes} \times B_{\boxtimes}$ have their distances larger than $2(1+\varepsilon)r$, implied by Lemma 7.6, so □ is active. Moreover, there exists no subset of M points from $B_{\square}$ as $B_{\boxtimes}$, where all pairs of points ($a, b' \in B_{\boxtimes}$) have their distances larger than $2(1+\varepsilon)r$, implied by Lemma 7.5, so □ is an active bucket for a. In Algorithm 1, when visiting □ by line 7-18, (a, b) must be reported by □ or have been reported previously. □

A.6 Pseudocodes in Section 7

Algorithm 3: UniEnumLSH

```
1  All buckets in C are sorted by the index of hash functions;
2  foreach □ ∈ C do
3      foreach (a, b) ∈ □_A × □_B do
4          if φ(a, b) ≤ r then
5              flag = true;
6              foreach j ∈ {1, 2, ··· , i − 1} do
7                  if g_j(a) = g_j(b) then
8                      flag = false;
9              if flag = true then
10                 Emit (a, b);
11                 foreach j ∈ {i + 1, i + 2, ··· , τ} do
12                     if g_j(a) = g_j(b) in □ then
13                         □_out ← □_out − 1;
14                         if □_out = 0 then
15                             C ← C − {□};
```

Algorithm 4: Insert($a \in A$)

```
1  foreach hash function g in the index do
2      □ ← the bucket with hash value g(a);
3      insert a into A_□;
4      if □ ∈ C − C then
5          Choose M arbitrary points from B_□ as B_⊠;
6          if there is b ∈ B_⊠ with φ(a, b) ≤ 2(1 + ε) then
7              C ← C ∪ {□};
8              Store (a, b) as the representative of □;
```

Algorithm 5: Delete($a \in A$)

```
1  foreach hash function g in the index do
2      □ ← the bucket with hash value g(a);
3      Delete a from □_A;
4      if □ ∈ C and a is in the representative pair of □ then
5          Choose M arbitrary points from A_□ as A_⊠;
6          Choose M arbitrary points from B_□ as B_⊠;
7          if there exists a pair (a', b') ∈ A_⊠ × B_⊠ with
             φ(a', b') ≤ 2(1 + ε)r then
8              Store (a', b') as representative of □;
9          else
10             C ← C − {□};
```

A.7 Extensions of Section 7

There are three extensions from our results for the LSH:

Remark 1. Similar to the LSH [36] used for ANN query, we can extend our current index to the case where r is also part of the query. In $\mathbb{H}^d$, $1 \leq r \leq d$. Hence, we build $Z = O(\log_{1+\varepsilon} d) = O(\varepsilon^{-1} \log d)$ indexes as described above, each of them corresponding to a similarity threshold $r_i = (1+\varepsilon)^i$ for $i = 1, \ldots, Z$. Given a query with threshold r, we first run a binary search and find r_j such that $r \leq r_j \leq (1+\varepsilon)r$. Then, we use the j-th index to answer the similarity join query. Overall, the index has $\widetilde{O}(dn + \varepsilon^{-1} n^{1+\rho} \log d)$ size can be constructed in $\widetilde{O}(\varepsilon^{-1} dn^{1+\rho} \log d)$ time, and updated in $\widetilde{O}(\varepsilon^{-1} dn^{\rho} \log d)$ amortized time. After finding the value r_j in $O(\log(\varepsilon^{-1} \log d))$ time, the delay guarantee remains $\widetilde{O}(dn^{\rho})$.

Remark 2. This framework of query enumeration in Hamming distance can be extended to ℓ_2 or ℓ_1 metric, by resorting the LSH family constructed in [24] and [32] (Section 20.2), obtaining an index with exactly the same complexity.

Currently, the best LSH family constructed for the ANN problem under ℓ_2 metric is presented in [7], with $\rho \leq \frac{1}{(1+\varepsilon)^2} + o(1)$. Our framework can benefit automatically from any improvement over the construction of LSH family. On the other hand, it is shown in [44] that $\rho \geq 0.462/(1+\varepsilon)$ for Hamming space and ℓ_1 metric, and $\rho \geq 0.462/(1+\varepsilon)^2$ for the ℓ_2 metric, which also implies some limitations of our approach.

Remark 3. It is known that the algorithm for similarity join can be used to answer the ANN query. Let P be a set of points in $\mathbb{R}^d$, where d is a large number, and ε, r be parameters. The ANN query asks that (1) if there exists a point within distance r from q, any one of them should be returned with high probability; (2) if there is no point within distance $(1+\varepsilon)r$ from q, it returns "no"with high probability. For any instance of ANN query, we can construct an instance of similarity join by setting $A = P$ and $B = \emptyset$. Whenever a query point q is issued for ANN problem, we insert q into B, invoke the enumeration query until the first result is returned (if there is any), and then remove q from B. Our index of $\widetilde{O}(dn + n^{1+\rho})$ size can answer $(1+2\varepsilon)$-approximate ANN query in $\widetilde{O}(dn^{\rho})$ time, which is not worse from the best index for answering ε-approximate ANN query.