Fast Approximate Similarity Join in Vector Databases

JIADONG XIE, The Chinese University of Hong Kong, China
JEFFREY XU YU, The Chinese University of Hong Kong, China
YINGFAN LIU*, Xidian University, China

Recent advancements in deep learning, particularly in embedding models, have enabled the effective representation of various data types such as text, images, and audio as vectors, thereby facilitating semantic analysis. A large number of massive vector datasets are maintained in vector databases. Approximate similarity join is a core operation in vector database systems that joins two datasets, and outputs all pairs of vectors from the two datasets, if the distance between such a pair of two vectors is no more than a specified value. Existing state-of-the-art approaches for approximate similarity join are selection-based such that they treat each data point in a dataset as an individual query point to search data points by an approximate range query in another dataset. Such methods do not fully capitalize on the inherent properties of the join operation itself. In this paper, we propose a new join algorithm, SimJoin. Our join algorithm aims to boost join processing efficiency by leveraging relationships between partial join results (e.g., join windows). In brief, our join algorithm accelerates the join processing to process a join window by utilizing the join windows from the processed data points. Then, we discuss optimizing join window order to minimize join costs. In addition, we discuss how to support k-similarity join, and how to maintain proximity graph index based on k-similarity join. Extensive experiments on real-world and synthetic datasets demonstrate the significant performance superiority of our proposed algorithms over existing state-of-the-art methods.

CCS Concepts: • **Information systems → Information retrieval query processing**.

Additional Key Words and Phrases: Approximate Similarity Join, High-Dimensional Vector

ACM Reference Format:
Jiadong Xie, Jeffrey Xu Yu, and Yingfan Liu. 2025. Fast Approximate Similarity Join in Vector Databases. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 158 (June 2025), 26 pages. https://doi.org/10.1145/3725403

1 Introduction

Recent advances in deep learning, particularly embedding models, have revolutionized the representation of various data types such as text [32], images [33], audio [6] and graphs [17]. These models encode data as vectors, capturing crucial information in a high-dimensional space for analysis. And many large datasets of vectors need to be maintained in the vector databases. Similarity join is one of the fundamental operations in vector database systems that join two datasets, X and Y, and output all pairs of vectors of X and Y, if they are similar, e.g., the distance between their vectors is no more than a specified threshold. As an example, auto-tagging [11, 43] is a real-world application that needs similarity join. Here, auto-tagging is to label a large number of documents unlabeled based on certain documents labeled. And an unlabeled document can be labeled by the

*Yingfan Liu is the corresponding author.

Authors' Contact Information: Jiadong Xie, The Chinese University of Hong Kong, China, jdxie@se.cuhk.edu.hk; Jeffrey Xu Yu, The Chinese University of Hong Kong, China, yu@se.cuhk.edu.hk; Yingfan Liu, Xidian University, China, liuyingfan@xidian.edu.cn.

ACM 2836-6573/2025/6-ART158
https://doi.org/10.1145/3725403

label of another labeled document, if the corresponding two document embeddings (vectors) are sufficiently similar.

In this paper, we study ϵ-similarity join for two d-dimensional datasets, X and Y, denoted as $X \bowtie_\epsilon Y$, for $d > 0$. The ϵ-similarity join finds all pairs of vectors (x_i, y_j) from $X \times Y$, if the distance $\delta(x_i, y_j)$, for $x_i \in X$ and $y_j \in Y$, is no more than the given parameter ϵ. Due to the curse of dimensionality [21], the exact ϵ-similarity join is computationally intensive [23, 24, 36]. We focus on the approximate ϵ-similarity join in this paper, which has not been well studied, whereas the k-NN (nearest neighbor) search and k-ANN (approximate k-NN) search have been extensively studied. In general, ϵ-similarity join, $X \bowtie_\epsilon Y$, can be done by k-NN/k-ANN search if it takes every vector $x_i \in X$ as a query vector to search its nearest neighbors in the dataset Y, as a ϵ-range query. We call them a selection-based approach for ϵ-similarity join. The existing state-of-the-art approaches for approximate ϵ-similarity join are all selection-based methods [44, 46, 51]. Such existing approaches focus on optimizing the join processing time by refining the selection operations, like skipping tuples with zero or fewer join results [46], bypassing the limitations of a top k-only interface to support ϵ-range search [51]. However, these approaches do not fully leverage the inherent property of the ϵ-similarity join. That is, they do not utilize the join results from processed data points to accelerate the processing time of other data points that have not been processed, which is the focus we study in this paper.

The main contributions of this work are summarized below. ❶ We propose a new ϵ-similarity join algorithm for $X \bowtie_\epsilon Y$ based on two key issues in join processing, which are join window sliding and join window order selection. Here, a join window is the join results in Y that are ϵ-similar to a data point x in X. Join window sliding is to leverage the join window results of some processed data points to accelerate the join results for other data points. And join window order selection is to determine the processing order of join windows to slide, in order to minimize the overall join processing time. ❷ To allow join window sliding, we discuss the continuous sliding from a data point to another adjacent data point in d-dimensional space. Such adjacent data points are difficult to handle in d-dimensional space, as there are 2^d directions to slide and there is no easy way to slide in order. We discuss how to capture such adjacent data points in d-dimensional space, and how to maintain such adjacent data points in an adjacent graph to be constructed. Furthermore, we discuss the differences between the adjacency issue and k-NN based similarity, and we show that the probability of using a proximity graph (e.g., k-NN based graph) as an approximate adjacent graph in practice. ❸ We give the optimal solution to determine the join window order selection, for $X \bowtie_\epsilon Y$. To find such optimal join window order, we show how to estimate the join window sliding cost on Y based on the information we have on X. We give our efficient approach to find the optimal solution. ❹ Based on our ϵ-similarity join algorithm, SimJoin, we further discuss how we support k-similarity join, and how we maintain a proximity graph index based on the k-similarity join algorithm we present. ❺ We report on extensive experiments on both real-world and synthetic datasets. (a) In general, SimJoin demonstrates significant performance improvements, achieving reductions in join time of over 1 order of magnitude compared to state-of-the-art methods for approximate similarity joins, along with superior join result quality. Moreover, it achieves more than 2 orders of magnitude reduction in join time compared to the state-of-the-art approach for exact similarity joins with less than 1% loss in join results. (b) Regarding index maintenance, our approach accelerates index maintenance significantly, consuming less peak memory, and enhancing index quality for k-approximate nearest neighbor search, compared to the state-of-the-art methods.

The paper is organized as follows. We discuss ϵ-similarity join and its key issues in Section 2. In Section 3, we discuss the main difficulties for selection-based approaches to support ϵ-similarity join by finding nearest neighbors in one dataset for any point in another dataset. In Section 4,

we propose a new approximate ϵ-similarity join algorithm that is based on join window sliding and join window order selection. Sections 5 and 6 delve into the specifics of addressing the two issues, join window sliding and join window order selection, respectively. In Section 7, we show that our techniques can be used to support k-similarity join, and to support maintenance of k-ANN based proximity graph indices. We conduct extensive experiments to confirm the effectiveness and efficiency of our approach in Section 8. We discuss the related works in Section 9, and conclude our work in Section 10.

2 The ϵ-Similarity Join

Given two sets of points, $X = \{x_1, x_2, \cdots, x_n\}$ and $Y = \{y_1, y_2, \cdots, y_m\}$, in Euclidean d-dimensional space, denoted as $\mathbb{R}^d$, for $d > 0$, the **ϵ-similarity join** $X \bowtie_\epsilon Y$ is defined as $X \bowtie_\epsilon Y = \{(x_k, y_l) \in X \times Y \mid \delta(x_k, y_l) \leq \epsilon\}$, where $\delta(x_k, y_l)$ is L_2 norm (i.e., Euclidean distance) between $x_k \in X$ and $y_l \in Y$, and ϵ is a user-given threshold where $\epsilon > 0$. The ϵ-similarity join, $X \bowtie_\epsilon Y$, can also be formalized based on join windows. Let J_i be the **join window** for $x_i \in X$ such that $J_i = \{y_l \in Y \mid \delta(x_i, y_l) \leq \epsilon\}$. $X \bowtie_\epsilon Y = \bigcup_{x_i \in X}(\{x_i\} \times J_i)$. We also refer the ϵ-similarity join, as **ϵ-similarity self-join**, if the two given datasets are identical.

As the exact ϵ-similarity join is time-consuming [23, 24, 36] due to the curse of dimensionality [21], in this paper, we study an approximate ϵ-similarity join algorithm, and its quality is measured by the average *recall* and average *precision*. Here, let $\tilde{J}_i$ be the results from an approximate ϵ-similarity join algorithm and $J_i = \{y_l \in Y \mid \delta(x_i, y_l) \leq \epsilon\}$, the average recall is defined as $\frac{1}{|X|} \cdot \sum_{x_i \in X} \frac{|\tilde{J}_i \cap J_i|}{|J_i|}$ and the average precision is defined as $\frac{1}{|X|} \cdot \sum_{x_i \in X} \frac{|\tilde{J}_i \cap J_i|}{|\tilde{J}_i|}$.

In the design of an approximate ϵ-similarity join algorithm, in this work, we focus on two key issues, namely, **join window sliding** and **join window order selection**. The join window, $J_i \subseteq Y$ of $x_i \in X$ is a d-ball centered at x_i with a radius of ϵ in the d-dimensional space. Suppose J_i has been processed for a given x_i. Join window sliding refers to the process of sliding the join window J_i to the join window J_j to be processed next for x_j. And join window order selection is how to select such x_j among the points in X that have not yet been processed, in order to minimize the join processing cost.

Here, we discuss the join cost of $X \bowtie_\epsilon Y$. Let $\aleph$ be the join window order over X such that $\aleph = \langle(\kappa_1, x_1), (\kappa_2, x_2), \cdots, (\kappa_n, x_n)\rangle$, where a pair of (κ_i, x_i) indicates that the join window, J_i, to be processed next for x_i is slide from the join window represented by $\kappa_i \in X$. Regarding κ_i, there are two cases. One is that κ_i represents x_j for $1 \leq j < i$ in $\aleph$, and the other is $\kappa_i = \emptyset$ as it is possible that the cost of finding J_i for x_i by an individual ϵ-range search is smaller if it does not slide from any join window found already. Note that $\kappa_1 = \emptyset$. The total join cost for $X \bowtie_\epsilon Y$ is defined as follows given a join window order $\aleph$:

$$C(X \bowtie_\epsilon Y) = \sum_{x_i \in X} \begin{cases} c_\epsilon(x_i) & (\emptyset, x_i) \in \aleph \\ c_\kappa(\kappa_i, x_i) & \text{otherwise} \end{cases} \tag{1}$$

where $c_\epsilon(x_i)$ is the cost of processing an ϵ-range search to find J_i for x_i, and $c_\kappa(\kappa_i, x_i)$ is the cost of finding J_i by sliding from the join window represented by κ_i. The problem to be studied is how to reduce such join processing cost.

The two issues can be easily handled for $X \bowtie_\epsilon Y$ when X and Y are in d-dimensional space for $d = 1$. We show its join algorithm in Algorithm 1. In Algorithm 1, both X and Y are sorted in ascending order. For x_i in the ascending order, it finds J_i by finding the leftmost and rightmost boundary point in J_i, denoted as $l(J_i)$ and $r(J_i)$, following the linear order of Y. The join window J_i forms a continuous interval in 1-dimensional space. The continuous interval is formed by sliding in Y from a point y_p to another point y_q on the condition that y_p and y_q are adjacent points such that

Algorithm 1: Join1D (X, Y, ϵ)

Input : two 1-dimensional datasets $X = \{x_1, \cdots, x_n\}$ and $Y = \{y_1, \cdots, y_m\}$ in ascending order and a parameter ϵ
Output : the result of $X \bowtie_\epsilon Y$

```
1 XY ← ∅; l ← 1; r ← 0;
2 for each x_i from 1 to n in the ascending order do
3     while l ≤ m and y_l < x_i − ε do l ← l + 1;
4     while r + 1 ≤ m and y_{r+1} ≤ x_i + ε do r ← r + 1;
5     J_i ← {y_l, ··· , y_r};
6     if l ≤ r then XY ← XY ∪ ({x_i} × J_i);
7 return XY
```

there is no y_w between y_p and y_q. The following is obvious. That is, $x_i \le x_{i+1}$ implies $l(J_i) \le l(J_{i+1})$ and $r(J_i) \le r(J_{i+1})$. The best to slide join window is to slide from the join window J_i to the join window J_{i+1}, and the join window order is to select x_{i+1} from x_i. The time complexity of Algorithm 1 is in $O(n + m)$.

The issue of join window sliding is that the continuous in join window sliding in d-dimensional space for $d > 1$ becomes challenging. First, there are 2^d directions to slide from a join window J_i to another join window J_j. In other words, two join windows, J_i and J_j, or the two d-balls, may overlap in any of 2^d ways. Second, the adjacent points, y_p and y_q in Y, are arbitrary as there are 2^d directions to slide, and there is no easy way to slide in order. In this paper, we propose a new efficient approximate ϵ-similarity join algorithm with high quality of the join results.

3 Selection-based Approaches

The ϵ-similarity join, $X \bowtie_\epsilon Y$, can be processed using the state-of-the-art k-ANN (Approximate Nearest Neighbor) search. That is, for every point $x_i \in X$, it issues an approximate ϵ-range query [46, 51] to find its join window J_i in Y. In [51], it presents a method to deal with ϵ as a filter on k-ANN search. In [46], it utilizes learning techniques to predict the number of join results for each data point, enabling the skipping of those with zero or fewer results. By such a selection-based approach, there are no issues on join window sliding and join window selection, because each of join window is treated individually. We discuss its main problem to slide exact/approximate join window, $J_i \subseteq Y$, for $x_i \in X$, below using an example.

Example 3.1: Let $X = \{x_1, x_2, x_3\}$ and $Y = \{y_1, y_2, \cdots, y_{10}\}$. Fig. 1 shows a k-NN graph over Y for $k = 2$, where a point is connected to its 1-NN and 2-NN, as indicated by blue solid/dashed edges. The X points are not explicitly shown in Fig. 1. Instead, we show every join window J_i for x_i in X, where x_i is the center of the corresponding d-ball, illustrated by a circle in 2-dimensional space. Here, $J_1 = \{y_8, y_9, y_{10}\}$, $J_2 = \{y_2, y_3\}$, and $J_3 = \{y_3, y_4, y_5\}$. As shown in Fig. 1, the two clusters of Y points, namely, $\{y_1, \cdots, y_7\}$ and $\{y_8, y_9, y_{10}\}$, are not connected over the k-NN graph. To enforce all points connected in a graph, in k-ANN graph ($k = 2$), it adds some red edges to make it connected and prunes some dashed blue edges to make the graph small for efficiency.

As shown in Example 3.1 with Fig. 1, first, on the k-NN graph, the join window J_1 cannot slide to the join window J_2, as there is no point in J_1 that has an blue edge to a point in J_2. Suppose that there is x_4 in X, and its join window $J_4 = \{y_8, y_1\}$, where y_1 and y_8 are not connected in the k-NN graph. The selection-based approaches start the search from a single entry point, which results in the inability to simultaneously locate both y_1 and y_8, since they belong to separate connected components. Second, on the k-ANN graph with red/blue solid edges, consider finding J_2 for x_2

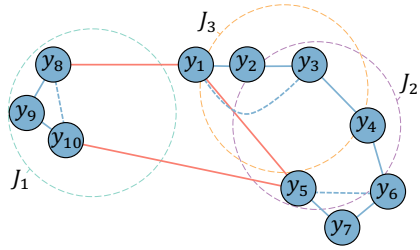


Fig. 1. k-NN/k-ANN graph

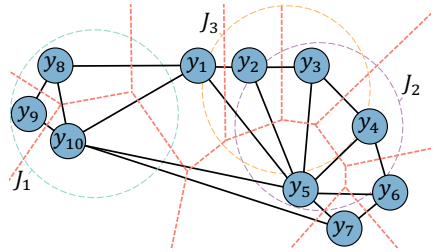


Fig. 2. An adjacent graph

from J_3. Staring from y_2 and y_3, it finds y_4 included in J_2, then it finds that y_6 cannot be in J_2, and accordingly, it stops search and misses y_5 which should be in J_2. In short, it is difficult for a selection-based k-NN or k-ANN approach to ensure high recall for ϵ-similarity join.

4 A New ϵ-Similarity Join Algorithm

In this section, we address the two key issues: join window sliding (Section 4.1) and join window order selection (Section 4.2). Subsequently, we integrate the techniques proposed to present our similarity join algorithm, SimJoin (Section 4.3).

4.1 Join Window Sliding

First, we define adjacent points, y and y', in a d-dimensional dataset Y. The definition applies to any d-dimensional spaces.

Definition 4.1: [**Adjacent points**] For any point set Y in a d-dimensional space for $d > 0$, two points y and y' in Y are adjacent if there exists a d-ball such that y and y' are two points on the boundary of d-ball and there is no any point y'' in Y inside d-ball.

A point y in Y may have multiple adjacent points, for instance, as illustrated in Fig. 2, y_2 is adjacent to y_1, y_3 and y_5.

Definition 4.2: [**Continuous Window**] A join window J is a continuous window if any two points, y_p and y_q, in J are connected by a sequence of pairs of adjacent points.

Definition 4.3: [**Continous Sliding**] A join window J_i slides to a join window J_j in a continuous manner, if there are two adjacent points, y_p and y_q, for $y_p \in J_i$ and $y_q \in J_j$.

From a different view, a point y_p sliding to its adjacent point y_q may cause a join window slides to a new join window. Note that the d-ball in Definition 4.1 is used to define two adjacent points in a d-dimensional space, and is different from the d-ball that corresponds to a join window or a continuous join window. It is important to note that the continous sliding ensures that we do not miss any points in Y for $X \bowtie_\epsilon Y$.

Example 4.1: Reconsider Example 3.1. Fig. 2 shows points in Y, where there is a line between two adjacent points. With the adjacent points in Fig. 2, the join window J_1 can slide to J_3 as $y_8 \in J_1$ and $y_1 \in J_3$ are adjacent points, and the join window J_3 can slide to J_2. All the three join windows are continuous windows. There are no missing points in any join windows.

Similarity vs Adjacency: First, regarding the k-NN based similarity and adjacency in one d-dimensional space (e.g., Y), the adjacency defined is different from the k-NN based similarity. It does not necessarily mean that two adjacent points are similar, and it does not necessarily mean two similar points are adjacent. Consider Example 4.1 and its Fig. 2. for y_1, its 1-NN, 2-NN, and 3-NN are y_2, y_3, and y_5, respectively. y_3 is more similar to y_1 than y_5. But, y_3 is not adjacent point

of y_1 but y_5 is. This is because y_1 can slide to y_5 directly, but y_1 cannot slide to y_3 without passing through y_2. Second, regarding $X \bowtie_\epsilon Y$ in two d-dimensional spaces, if y_p and y_q are adjacent in Y, it means that it is possible that there exists a point x_i in X, which takes y_p and y_q as its 1-NN and 2-NN to join. In general, if x_i can ϵ-similarity join y_p but cannot ϵ-similarity join its adjacent point y_q, it does not necessarily to explore ϵ-similarity join between x_i and any other adjacent points of y_q in Y. In other words, if y_p is in the join window J_i for x_i, but its adjacent point y_q not, then it does not need to slide from y_q to any other adjacent points of y_q in Y.

Next, we show that a join window, J_i, for arbitary data point x_i is continuous window (Definition 4.2).

First, for an arbitrary y_x, its 1-NN y_p and 2-NN y_q are adjacent since the d-ball centered at y_x contains only y_p and y_q in it. However, in general, for an arbitrary y_x, it does not necessarily mean that (k-1)-NN y_p and k-NN y_q are adjacent from the viewpoint of y_x. In Fig. 3, suppose y_x is the large circle center, the distance between y_i and y_x is smaller than the distance between y_j and y_x if $i < j$, and the adjacent points are connected by blue lines. Here, y_1 and y_2 are adjacent, and are 1-NN and 2-NN of y_x, whereas y_5 and y_6 are 5-NN and 6-NN of y_x, but y_5 and y_6 are not adjacent.

For easy discussion, in the following, we use y_p to show that y_p is p-NN of y_x. To show that a join window, J_i is continuous, we show that y_k is adjacent to one of y_j, for $1 \leq j < k$. Consider a d-ball centered at the segment from y_k to y_x, passing through y_k. By sliding the d-ball center along the segment from y_k towards y_x, y_j is the adjacent to y_k if y_j is the first on such a d-ball. Note that y_j must be closer to y_x. This is because the sliding d-ball remains within the large d-ball centered at y_x and passing through y_k, and y_j is on the sliding d-ball. We explain it using Fig. 3. Consider y_8 in Fig. 3. We show that y_8 must be adjacent to y_5. The segment is depicted as a yellow line between y_8 and y_x. By sliding a d-ball (depicted as a red circle), it encounters y_5, and y_5 and y_8 are adjacent. And y_5 must be closer to y_x than y_8. This is because the red d-ball remains within the large d-ball centered at y_x and passing through y_8, and y_5 is on the red d-ball. This fact implies that the join window of an arbitrary data point y_x is continuous.

Lemma 4.1: *Given a dataset Y, regarding an arbitrary data point y_x, its k-NN y_k in Y are adjacent with at least one of the j-NN of y_x in Y, for $1 \leq j < k$.*

We omit the proof as it can be done as discussed above.

To handle the adjacent points in a d-dimensional space Y, we define an adjacent graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

Definition 4.4: [**Adjacent Graph**] An adjacent graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is for a given d-dimensional dataset Y. Here, $\mathcal{V}$ is a set of points in Y, and $\mathcal{E}$ is a set of edges where $(y_p, y_q) \in \mathcal{E}$ if y_p and y_q are adjacent points by Definition 4.1.

Based on Lemma 4.1, we further show that a join window forms a connected subgraph $\mathcal{G}_i$ in $\mathcal{G}$ for Y, i.e., it is continous window.

Corollary 4.1: *For an adjacent graph $\mathcal{G}$ of dataset Y, for an arbitrary data point y_x, the nodes within its join window form a connected subgraph in $\mathcal{G}$.*

Proof Sketch: It is evident that the join window of y_x contains its top-k nearest data points, $\{y_1, y_2, \cdots, y_k\}$, in Y, if $\delta(y_i, y_x) \leq \epsilon$ for $1 \leq i \leq k$ and $\delta(y_{k+1}, y_x) > \epsilon$. We prove that its top-$k$ data points in Y form a connected subgraph by induction. For the base case where $k = 2$, 1-NN and 2-NN of y_x are adjacent, hence there is an edge to connect both in $\mathcal{G}$. Suppose that the subgraph for top-(k-1) nearest data points, $\{y_1, y_2, \cdots, y_{k-1}\}$ are connected in $\mathcal{G}$. Based on Lemma 4.1, we have

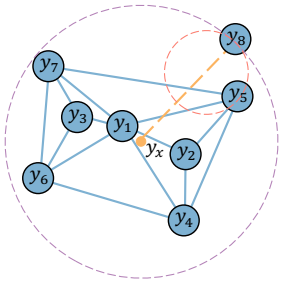


Fig. 3. Example for adjacency

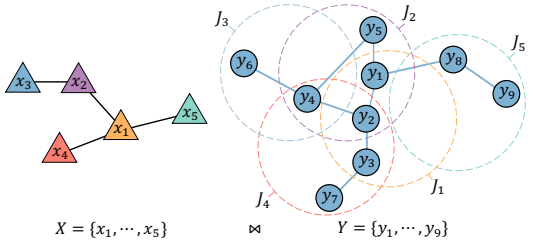


Fig. 4. Example for similarity join window order

the k-NN y_k is connected to one of its j-NN y_j, where $1 \leq j < k$. This implies that the sugraph for the top-k data points are connected. □

As a join window, J_i, is a connected subgraph, $\mathcal{G}_i$, in $\mathcal{G}$, we can slide a join window J_i to another join window J_j over $\mathcal{G}$ by continous sliding (Definition 4.3).

Adjacent Graph Construction: To construct an adjacent graph, $\mathcal{G}$, for a d-dimensional dataset, Y, it needs to check if two points, y_p and y_q, in Y are adjacent. In doing so, it needs to check if there exists a d-ball, B_{pq}, where y_p and y_q are on its boundary and no any points are in the ball. Let the center of the d-ball, B_{pq}, be y_c, which is located at the bisector hyperplane between y_p and y_q. As shown in Fig. 2, y_1 and y_2 are adjacent, because there is a circle (i.e., 2-ball) whose center is on the perpendicular bisector (the red line between y_1 and y_2) of y_1 and y_2, only y_1 and y_2 are on the boundary of such circle, and no other points are in the circle. On the other hand, y_1 and y_3 are not adjacent, because it is impossible to find a circle that does not have y_2 inside the circle. The cost of checking the existence of a d-ball for every pair of points is too high.

Here, we construct $\mathcal{G}$ by space partitioning. We partition the d-dimensional space, Y, into $|Y|$ subspaces, where each subspace, S_p, contains only one point, y_p, in Y, which implies that any $x_i \in X$ point in the subpace, S_p, will take y_p as its 1-NN. Such space partitioning is illustrated in Fig. 2 by the red lines. With such space partitioning, y_p and y_q are adjacent points if their subspaces share at least one point (note that the point mentioned here is an arbitrary data point in the space, which may not appear in the dataset Y). In other words, there must exist a d-ball centered at the boundary of the two subspaces for y_p and y_q. Such space partitioniong is equivalent to the Voronoi diagram in the hyperspace [13]. Note that a Voronoi diagram partitions space into subspaces that are closer to each of a specified set of data points. Its dual graph, Delaunay triangulations (DT) [12, 26] can be identified. For example, In Fig. 2, the DT is shown by the black lines to connect two adjacent points if their subspaces share at least one point. In this work, we construct $\mathcal{G}$ for Y by constructing DT over Y.

4.2 Join Window Order Selection

We have discussed join window sliding in Section 4.1, and give a join cost function (Eq. (1)) for $X \bowtie_\epsilon Y$ which we need to minimize. In this section, we delve into the discussion of the join cost and strategies for minimizing it.

In processing $X \bowtie_\epsilon Y$, suppose that we have processed some points, $\overrightarrow{X} = \{x_p, \cdots, x_i, \cdots, x_q\}$ in X, and the join windows processed are $\overrightarrow{J} = \{J_p, \cdots, J_i, \cdots, J_q\}$. To minimize the total join cost is to minimize the cost of processing a new join window. And the join window order selection is to select a $x_i \in \overrightarrow{X}$ to process $x_j \in (X \setminus \overrightarrow{X})$ such that the cost to process the next join window J_j is

Algorithm 2: SimJoin (X, Y, ϵ)

Input : two d-dimensional datasets X and Y, and a parameter ϵ
Output : The ϵ-similarity join results

```
1  G_C ← the complete weighted graph of X; 𝒢 ← the adjacent graph of Y; y_0 ← the entry point of 𝒢;
   initialize w to control the sliding size;
2  ℵ ←WindowOrder (G_C, y_0);
3  XY ← ∅;
4  for each (κ_i, x_i) ∈ ℵ do
5      if κ_i = y_0 then
6          J_i ←JoinSlide (𝒢, y_0, x_i, {y_0}, ϵ, w);
7      else
8          J_i ←JoinSlide (𝒢, κ_i, x_i, J_{κ_i}, ϵ, w);
9      XY ← XY ∪ J_i;
10 return XY;
```

minimized. It is important to note that a join window is a continuous window. In general, the cost of sliding from a join window J_i to a join window J_j to be processed next is small, if $|J_i \cap J_j|$ is large, because we can process a part of J_j while sliding from J_i to J_j. The difficulty is that we do not know J_j, and we do not know $|J_i \cap J_j|$ before processing J_j. Hence, we solve this selection problem by selecting $x_i \in \overrightarrow{X}$ to process $x_j \in (X \setminus \overrightarrow{X})$ instead. It is important to note that if $\delta(x_i, x_j)$ is smaller than $\delta(x_l, x_j)$, it implies that cost of sliding from x_i to x_j is smaller than the sliding from x_l. In other words, here, we use J_i for x_i to process J_j for x_j if $\delta(x_i, x_j)$ is the smallest among all points in $\overrightarrow{X}$. It is easy to check whether $|J_i \cap J_j|$ is empty or not. When $|J_i \cap J_j| \neq \emptyset$, we continuously slide from J_i to J_j. When $|J_i \cap J_j| = \emptyset$, we have two choices to slide to J_j. One is to continuously slide from a point in J_i towards J_j, and the other is to slide from the entry point in the adjacent graph $\mathcal{G}$ for Y, i.e., seen finding J_j as an individual ϵ-range query. We take the smaller one when $|J_i \cap J_j| = \emptyset$.

As an example, consider $X \bowtie_\epsilon Y$ for $X = \{x_1, \cdots, x_5\}$ and $Y = \{y_1, \cdots, y_9\}$ as shown in Fig. 4. In Fig. 4, we show the adjacent graph $\mathcal{G}$ on Y, and d-balls for the join windows, $\{J_1, \cdots, J_5\}$. Assume that we have processed $J_4 = \{y_2, y_3, y_4, y_7\}$ for x_4. Suppose x_1 is 1-NN of x_4, and we know $J_1 \cap J_4 \neq \emptyset$. Then we continuously slide from y_2 towards y_1 to process J_1 where $J_1 = \{y_1, y_2, y_3\}$. Next, suppose x_5 is 1-NN of x_1, and we will select J_1 to process J_5 for x_5. By checking points in J_1, we know that $J_1 \cap J_5 = \emptyset$. Next, we compare $\delta(x_1, x_5)$ with $\delta(y_0, x_5)$, where y_0 is a fixed entry point in $\mathcal{G}$. If $\delta(x_1, x_5) < \delta(y_0, x_5)$, we continuously slide from y_1 towards J_5 for x_5. Otherwise, we slide from the fixed y_0 towards J_5.

We give details of the cost function for $C(X \bowtie_\epsilon Y)$ (Eq. (1)). Suppose x_j is the next point to process, and x_i is the point closest to x_j among the processed data points, i.e., in $\overrightarrow{X}$. When $J_i \cap J_j = \emptyset$, we estimate the cost by $c_\epsilon(x_j) = \min\{\delta(x_i, x_j), \delta(y_0, x_j)\}$ where y_0 is a fixed point in $\mathcal{G}$ for Y. When $J_i \cap J_j \neq \emptyset$, we estimate its cost by $c(x_i, x_j) = \delta(x_i, x_j)$.

To solve the join window selection problem, we construct a weighted graph $G_C = (V_C, E_C)$. Here, $V_C = X \cup \{y_0\}$, and for every edge $(u, v) \in E_C$, its weight is $\delta(u, v)$. The minimum cost of $C(X \bowtie_\epsilon Y)$ is the cost of the minimum spanning tree (MST) of the weight graph, G_C, rooted at y_0.

4.3 The Join Algorithm

The ϵ-similarity join algorithm is given in Algorithm 2, which takes X, Y, and ϵ as its inputs. Here, $\mathcal{G}$ is the adjacent graph constructed over Y, and G_C is the weighted graph for join window selection over X together with a fixed point $y_0 \in \mathcal{G}$. To control the size of the sliding window, a parameter w

is given. A higher value of w will enhance the quality of join results but at the expense of efficiency. In Algorithm 2, we identify a join window order, $\aleph$, by calling WindowOrder (line 2), and we process $X \bowtie_\epsilon Y$ by sliding join windows following the order given in $\aleph$ (lines 4-9).

Note that our join algorithm can be extended to support parallelism. In brief, the join order $\aleph$ we use for approximate similarity join forms a tree, as discussed in the preceding subsection, where for every $(\kappa_i, x_i) \in \aleph$, κ_i is the parent node of x_i within the tree. Here, a child node can process the join window sliding, when its parent node completes its join processing. And all the child nodes of the same parent can be processed in parallel. Based on this, different strategies can be employed for parallel processing in SimJoin. For example, we can utilize the idea of parallel breadth-first search to parallelly process the join window sliding within the same layer and subsequently process them in a layer-wise fashion.

5 Join Window Sliding

We give an adjacent graph $\mathcal{G}$ where the edges in $\mathcal{G}$ connecting adjacent points in Y. However, in a high dimensional space, $\mathcal{G}$ becomes a complete graph [19, 26, 38], which implies that all points can possibly be adjacent in a d-dimensional space when d becomes larger (e.g., $d > 100$). Thus, it becomes impractical for $\mathcal{G}$ to be used to slide over the join windows. As a result, we need to construct an approximate adjacent graph that can be used in practice.

For an approximate adjacent graph to be constructed, we consider that it is impractical in a system to maintain two large graph indices. One is based on k-ANN for k-ANN search or approximate ϵ-range query, and the other one is based on adjacency for ϵ-similarity join. We explore if we can use one graph to serve all the purposes. In other words, we explore if a proximity graph constructed for k-ANN search can be used as an approximate adjacent graph $\mathcal{G}$. The question here is how likely a pair of near neighbors become adjacent. To answer this question, we consider the adjacency in Y from the angle of an arbitrary point in X. To be more precise, consider the adjacency for 3 data points, y_u, y_v, and y_w in Y, for an arbitrary data point x_q in X. Here, x_q is a point to join with points in Y. Suppose y_u is the nearest point to x_q, and y_v has a smaller distance to y_u in comparison with the distance between y_w and y_u. How likely is it for y_u and y_v to be adjacent?

Theorem 5.1: *Assume that the data points are uniformly distributed in an infinite d-dimensional space. Suppose that there are three data points, y_u y_v, and y_w in Y, and suppose y_u is the nearest data point to a given data point x_q, and $\delta(y_u, y_v) < \delta(y_u, y_w)$. Then, y_v has a higher probability to have a smaller distance to x_q compared to the distance from y_w to x_q.*

Proof Sketch: Given that y_u is the nearest data point to x_q, x_q resides within the region delineated by the two perpendicular hyperplanes between y_u and y_v and between y_u and y_w, regarding y_u. This region can be further segmented by the perpendicular hyperplane of y_v and y_w. If y_v is closer to x_q than y_w, then x_q lies between the perpendicular hyperplanes of y_u and y_v, and y_v and y_w. The probability of x_q being in this region is given as $\angle uvw/(\angle uvw + \angle uwv)$, where $\angle uvw$ is the angle centered at v. As $\delta(u, v) < \delta(u, w)$, we have $\angle uvw > \angle uwv$, which implies that y_v is more likely to be closer to x_q than the distance from y_w to x_q. □

Based on Theorem 5.1, because y_u is the 1-NN to x_q, the data points which have a smaller distance to y_u have a higher probability to have a smaller distance to x_q, and in other words, have a higher probability to become the 2-NN to x_q. Hence, there is a d-ball centered at x_q that contain only y_u and y_v, indicating that y_v has higher probability to be adjacent with y_u in the d-dimensional space, from the angle of such x_q. Recall that we have shown that for two adjacent points, y_u and y_v in Y, there exists an data point that take y_u and y_v as their 1-NN and 2-NN. Here, we show that if y_u and y_v are 1-NN and 2-NN of a data point in X, then it is likely that y_u and y_v are adjacent.

Algorithm 3: JoinSlide ($\mathcal{G}, x_i, x_j, J_i, \epsilon, w$)

Input : an adjacent graph $\mathcal{G}$ of dataset Y, two data points x_i, x_j in dataset X, the join window J_i of x_i and two parameters ϵ, w
Output : the join window J_j of x_j in Y

1 $Q \leftarrow J_i$ is a priority queue, sorted in ascending order based on $\delta(y_p, x_j)$ for every $y_p \in Q$;
2 **if** $\delta(Q.top(), x_j) > \epsilon$ **then**
3 $Visit \leftarrow J_i$;
4 **while** Q is not empty **do**
5 **if** $\delta(Q.top(), x_j) \leq \epsilon$ **then** break;
6 $u \leftarrow Q.pop()$;
7 **for** each $v \in N_{\mathcal{G}}(u) \setminus Visit$ **do**
8 **if** $\delta(v, x_j) < \delta(u, x_j)$ **then**
9 $Q.push(v)$; $Visit \leftarrow Visit \cup \{v\}$;
10 **while** $Q.size() > w$ **do**
11 remove the last element from Q;
12 **if** Q is empty **then return** $J_j = \emptyset$;
13 $J_j \leftarrow \{y_p | y_p \in Q \wedge \delta(y_p, x_j) \leq \epsilon\}$;
14 push all elements in J_j into an empty queue $\boldsymbol{Q}$;
15 **while** $\boldsymbol{Q}$ is not empty **do**
16 $y_p \leftarrow \boldsymbol{Q}.pop()$;
17 **for** each $y_q \in N_{\mathcal{G}}(y_p)$ **do**
18 **if** $y_q \notin J_j$ and $\delta(y_q, x_j) \leq \epsilon$ **then**
19 $\boldsymbol{Q}.push(v)$; $J_j \leftarrow J_j \cup \{y_q\}$;
20 **return** J_j;

Based on our analysis, we show that a single proximity graph designed for k-ANN search can be utilized for the purpose of ϵ-similarity join. Suppose there is a k-ANN graph with $k = 20$, where y_v is the 20-NN of y_u. This implies that, from the angle of data points in dataset X, there might exist a data point x_q to join, which takes y_u and y_v as its 1-NN and 2-NN.

In the following, the adjacent graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is constructed as a proximity graph (e.g., k-ANN graph). We use $N_{\mathcal{G}}(y_u)$ to indicate the neighbor list of node y_u in $\mathcal{G}$, i.e., $N_{\mathcal{G}}(y_u) = \{y_v | (y_u, y_v) \in \mathcal{E}\}$. In a proximity graph, a degree constraint of m is enforsed to limit the maximum of m out-neighbors for each node in aG [14, 28, 30, 34] such that $|\mathcal{E}| = O(nm)$, where n is the size of dataset Y.

The join window sliding algorithm, named JoinSlide is given in Algorithm 3. It takes 5 inputs, namely, the adjacent graph $\mathcal{G}$ over Y, two points, x_i, x_j in X, the join window J_i for x_i, together with two parameters, ϵ and w. The algorithm ouputs the join window J_j for x_j. We discuss two cases, namely, $J_i \cap J_j = \emptyset$ and $J_i \cap J_j \neq \emptyset$.

(i) $J_i \cap J_j = \emptyset$**:** Furthermore, there are two cases, ❶ to slide towards J_j from J_i, or ❷ to slide towards J_j from y_0.

❶ We slide the join window from the nodes in the join window J_i towards their neighbors, in order to ultimately locate at least one data point contained in the join window J_j. Specifically, we populate the priority queue by adding data points from J_i, where the priority queue is sorted according to the ascending order of distance to x_j (line 1). For each point at the top of the queue (lines 4-11), we explore its neighbors in $\mathcal{G}$ to find data points to the sliding direction, i.e., closer to x_j (lines 7-9). If closer data points are found, they are added to the queue (lines 8-9). This exploration

process continues until a data point within a distance no greater than ϵ from x_j is identified (line 5), or the search terminates when there are no closer data points to x_j (lines 4, 12). Note that a parameter w is utilized to cap the size of priority queue Q size (lines 10-11), thereby enhance the efficiency of join window sliding algorithms by reducing distance computations in practice. After this process, if there exists at least one data point within a distance not exceeding ϵ from x_j, we then turn to process to the case (ii) (lines 13-19).

❷ This is a case of an approximate ϵ-range query. We slide from an entry point y_0 in $\mathcal{G}$ towards their neighbors, in order to locate at least one data point contained in the join window J_j in a similar way as discussed for ❶. This case can be achieved by invoking JoinSlide $(\mathcal{G}, p_0, x_j, \{p_0\}, \epsilon, w)$.

Example 5.1: As illustrated in Fig. 4, the blue lines are edges in an approximate adjacent graph of $Y = \{y_1, \cdots, y_9\}$, with y_2 is the entry point of the graph. Considering the join windows J_1 for x_1 and J_5 for x_5, where $J_1 \cap J_5 = \emptyset$, our initial step involves acquiring at least one data point contained in J_5 by sliding the graph. This step entails two strategies: ❶ Starting from nodes y_1, y_2, and y_3, we discover that y_1's neighbor y_8 is included in J_5; ❷ Starting from node y_2, we find that its neighbor y_1 is closer to x_5, next identifying that y_1's neighbor y_8 is included in J_5.

(ii) $J_i \cap J_j \neq \emptyset$**:** In this case, acquiring the join window J_j is straightforward, given that we are already aware of at least one data point, y_p, contained in the join window J_j. We begin exploration from y_p. This is because the join windows of any data points in $\mathcal{G}$ are continuous windows (Corollary 4.1). To elaborate, we search from any y_p in $J_i \cap J_j$ and add it to the queue (line 13-14). Then, we expand nodes in the queue (line 16) by exploring their neighbors within a distance no greater than ϵ from y_p (lines 17-19). We continue the search process until no more neighbors can be added to the queue (line 15) and finally return J_j (line 20).

Example 5.2: Reconsider Example 5.1, we aim to slide the join window from J_1 to J_5, where we have found y_8 in J_5. Next, in the second step, we start from y_8, and explore its neighbors, which results in the discovery of y_8 and y_9 within J_5. For another instance, when sliding the join window from J_1 to J_4, upon finding that $J_1 \cap J_4 = \{y_2, y_3\}$ as outlined in line 13 of Algorithm 3. Hence we proceed to explore starting from these two data points, and find that y_4 and y_7 are also in J_4. Then the explore terminates as no additional neighbors are in J_4.

The Cost of Join Window Sliding: Recall that we give the join cost function for $C(X \bowtie_\epsilon Y)$ (Eq. (1)), and discuss how to use distance function, $\delta(,)$, to estimate the join costs in Section 4.2. A question that is unanswered is whether such cost estimation is feasible given the ϵ-similarity join algorithm, Algorithm 2, and in particular the JoinSlide algorithm (Algorithm 3). It is important to mention that, on the one hand, as shown in Algorithm 3, the part of the join cost of JoinSlide is the number of distance computations (NDC) while sliding over $\mathcal{G}$ within the join window J_j (case (ii)). On the other hand, the cost from sliding from J_i to J_j (case(i)), which can be estimated by $\delta(x_i, x_j)$, and it is the only cost related to the join window order.

We conduct some experimental studies using Msong dataset. Here, we randomly divide Msong into two sets in the same size, denoted as X and Y, and conduct the ϵ-similarity join on such X and Y. In particular, we select 1,000 pairs of $(x_i, x_j) \in X \times X$ where $x_i \neq x_j$, and obtain NDC during lines 1-12 using the JoinSlide algorithm (Algorithm 3) by sliding from the join window J_i of x_i to the join window J_j of x_j on the adjacent graph $\mathcal{G}$, which is constructed by NSG [14] of Y. The results are presented in Fig. 5. As shown in Fig. 5, the further apart x_i and x_j are, the more sliding is required, which leads to a larger NDC. The results indicate that the NDC increases as the distances between the two points increase, and it is almost linear when the distance is under 0.9.

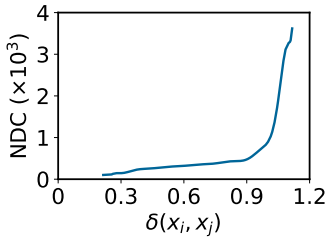


Fig. 5. NDC v.s. $\delta(x_i, x_j)$ on Msong dataset

Table 1. ϵNG graphs

Dataset	ϵ	d_{avg}
Rand	0.57690	50.966
Gauss	0.85741	11.201
Msong	0.70702	17.277
GIST	0.80105	17.026
GloVe	0.71288	45.205
Crawl	0.56437	27.796
SIFT	0.69817	27.769
Higgs	0.55096	28.755

6 Join Window Order Selection

In this section, we discuss join window order selection for $X \bowtie_\epsilon Y$. Here we assume the set of X is smaller than that of Y in size.

6.1 The Weighted Graph G_C

To determine the join window order, we discuss a weighted graph $G_C = (V_C, E_C)$ where $V_C = X \cup \{y_0\}$ in Section 4.2 with which the optimal join window order can be determined by an MST on G_C. Such G_C constructed is a complete weight graph, and needs to construct online because it needs dataset X and an entry point of y_0 taken from $\mathcal{G}$ of Y for $X \bowtie_\epsilon Y$. Therefore, the G_C construction time is a part of ϵ-similarity join. Therefore, it is impractical to construct G_C as it takes $O(dn^2)$ for a d-dimensional dataset X where $n = |X|$. This construction cost can be even larger than the join processing time given G_C is constructed.

Instead of constructing a completed weighted graph, $G_C = (V_C, E_C)$, we construct a ϵ-neighbor proximity graph (ϵNG), and denote it as $G_\epsilon = (V_\epsilon, E_\epsilon)$, where $V_\epsilon = X \cup \{y_0\}$. In G_ϵ, a data point is connected to the nodes in V_ϵ which distance between them is no greater than ϵ. The weight of an edge, (u, v), in E_ϵ is the distance $\delta(u, v)$. The following lemma establishes that the total weight of the MST on a connected ϵNG is equal to the minimum join cost.

Lemma 6.1: *The minimum join cost of $X \bowtie_\epsilon Y$ is equal to the total MST weight of G_ϵ if G_ϵ is connected.*

Proof Sketch: Since the $G_\epsilon = (V_\epsilon, E_\epsilon)$ is the subgraph of the complete graph $G_C = (V_C, E_C)$, where $V_C = V_\epsilon = X \cup \{y_0\}$ and $E_\epsilon \subseteq E_C$). The total MST weight over G_ϵ, denoted as $w(G_\epsilon)$, is no less than the total MST weight over G_C, denoted as $w(G_C)$. Assume that $w(G_\epsilon) > w(G_C)$, there must exist an edge (u, v) such that $(u, v) \notin G_\epsilon$ and $\delta(u, v) < w_{\max}(u \rightsquigarrow v)$, where $w_{\max}(u \rightsquigarrow v)$ is the maximum edge weight along the path from u to v in the MST of G_ϵ. However, it becomes evident that for ϵNG, the value of ϵ must satisfy $\epsilon > w_{\max}(u \rightsquigarrow v)$, whereas $\delta(u, v) < w_{\max}(u \rightsquigarrow v)$. This implies that (u, v) must be included in such ϵNG, leading to a contradiction. Therefore, we conclude that $w(G_\epsilon) = w(G_C)$. □

Lemma 6.1 states that ϵNG we require is a graph with a sufficiently large ϵ value to guarantee connectivity. In Table 1, we show ϵNG with the smallest ϵ value that guarantees G_ϵ connected, where d_{avg} represents the average node degree in this minimal connected ϵNG. The results indicate that the graph size needed remains consistently small across all evaluated datasets.

In practice, constructing an exact ϵNG, $G_\epsilon = (V_\epsilon, E_\epsilon)$ for $V_\epsilon = X \cup \{y_0\}$ is still time-intensive. We do as follows. First, we construct an approximate ϵNG, $\tilde{G}_\epsilon = (\tilde{V}_\epsilon, \tilde{E}_\epsilon)$, over X, which can be done offline. Here, an approximate ϵNG can be constructed based on the proximity graphs by considering both in-neighbors and out-neighbors for each node to identify its neighbors for ϵNG. It is because many state-of-the-art approaches for proximity graphs rely on an approximate k-NN graph and

Algorithm 4: MST (G)

Input : a proximity graph $G = (V, E)$ with each node $u \in V$ having neighbors sorted in ascending order by their distances to u
Output : the MST of proximity graph G

```
1  T ← ∅; l[u] ← 1 for each u ∈ V;
2  assign u ∈ V to the set only contains itself;
3  while T contains less than |V| − 1 edges do
4      E_min[s] ← ∅ for each set s that contains any data point u ∈ V;
5      for each u ∈ V do
6          v ←the l[u]-th node in N_G(u);
7          while l[u] ≤ |N_G(u)| and u, v are in the same set do
8              l[u] ← l[u] + 1; v ←the l[u]-th node in N_G(u);
9          if l[u] ≤ |N_G(u)| then
10             s ← the set contains data point u;
11             if E_min[s] = ∅ or δ(u, v) < weight of E_min[s] then
12                 E_min[s] ← (u, v);
13     for each set s contains any data point u ∈ V do
14         (u, v) ← E_min[s];
15         if u and v are in different sets then
16             merge the sets containing u and v;
17             add edge (u, v) into T with weight δ(u, v);
18 return T;
```

incorporate diverse pruning strategies [14, 34, 47]. These strategies preserve most short-distance edges within approximate k-NN graph and further add edges to ensure graph connectivity. Next, based on $\tilde{G}_\epsilon$ constructed over X, we construct $G_\epsilon = (V_\epsilon, E_\epsilon)$ by adding a new node, y_0, taken from $\mathcal{G}$ of Y, and add an edge between x_i and y_0, for every $x_i \in \tilde{V}_\epsilon$ with the edge-weight of $\delta(x_i, y_0)$ and satisfies $\delta(x_i, y_0) \leq \epsilon$.

6.2 The Algorithm for Order Selection

The MST can be computed for $G_\epsilon = (V_\epsilon, E_\epsilon)$ using the Kruskal's algorithm efficiently if E_ϵ has been sorted. We can sort edges in $\tilde{G}_\epsilon$ constructed for X offline before any $X \bowtie_\epsilon Y$. But, we have to sort all edges in E_ϵ when y_0 is added together with new edges between every x_i and y_0 online, in order to process $X \bowtie_\epsilon Y$. Therefore, the time complexity to find MST using the Kruskal's algorithm becomes $O(m|V_\epsilon| \log(m|V_\epsilon|))$, where m is the degree limit of G_ϵ. We call it as baseline algorithm. Such a baseline algorithm is time-consuming as illustrated in **Exp. 4** in Section 8, where it consumes over 20% of the total time to determine join window selection GloVe dataset.

Thus, we need a new algorithm for MST. At a high level, in the first phase, we find MST for $\tilde{G}_\epsilon$ over X, which can be done offline. In the second phase, for MST to be found over G_ϵ with a new node y_0 and $|X|$ edges added, we insert $|X|$ edges into the MST of $\tilde{G}_\epsilon$ to substitute certain existing edges. We give the details as follows.

The first phase: It finds MST of the proximity graph $\tilde{G}_\epsilon = (\tilde{V}_\epsilon, \tilde{E}_\epsilon)$, using the Kruskal's algorithm in $O(m|X| \log(m|X|))$. We give a more efficient algorithm in Algorithm 4, which finds MST, under the assumption that the neighbors of each node $u \in \tilde{V}_\epsilon$ are sorted in ascending order based on their

distance to u, which can be seamlessly integrated during the proximity graph ($\tilde{G}_\epsilon$) construction without requiring additional storage for such information.

Initially, each data point is allocated to an individual set containing only itself (line 2). Then, in each iteration, we identify the edge with the minimum weight from each set to others (lines 5-12) by exploring unvisited edges from each data point (lines 6-8). When we have found such edges across sets, we add them into the MST if the two nodes of the edges are not already connected (lines 13-17). The construction process concludes when the MST contains $|X| - 1$ edges (Line 5). To store the MST in the proximity graph, we can assign a label to the edges within the MST (line 18).

Checking whether two elements are in the same set (lines 7, 15) and merging the set operation (line 16) can be accomplished using the techniques proposed in [15]. Each operation requires $O(\alpha(N))$ time in total, where N is the total number of elements, and $\alpha(\cdot)$ denotes the inverse function of Ackermann's function, which is less than 5 for any practical input, can be seen as a constant [3]. Additionally, in each iteration, if there are $|S|$ distinct sets containing any data point $x \in X$, at least $\lfloor |S|/2 \rfloor$ edges will be added into the MST at this iteration. Hence, the algorithm comprises a maximum of $\log |X|$ rounds. Therefore, this phase runs in $O(m|X| + |X| \log |X|)$ time. In practice, $\log |X|$ is lesser or comparable to m, hence the time complexity aligns with the size of the proximity graph G.

Then, we provide the subsequent theorem to prove the correctness of this phase for constructing MST of proximity graph of X.

Theorem 6.1: *Given a proximity graph G of dataset X, Algorithm 4 obtains the MST of graph G.*

Proof Sketch: In Algorithm 4, each added edge connects two previously unconnected sets, culminating in a connected graph with $|V(G)| - 1$ edges, satisfying the tree property. To establish that the resultant tree is MST of G, we prove that the minimum-weight edge bridging any two disjoint subgraphs G_1 and G_2 ($V(G_1) \cap V(G_2) = \emptyset$) in G is indeed part of the MST. We prove it by contradiction. Assume the edge (u, v) with the least weight crossing G_1 and G_2 is absent from MST of G. It implies $\delta(u, v) > w_{\max}(u \rightsquigarrow v)$, where $w_{\max}(u \rightsquigarrow v)$ denotes the highest edge weight along the path from u to v within the MST of G. This contradicts the initial premise that (u, v) is the minimum-weight edge spanning G_1 and G_2, since one of the edges along the path from u to v is crossing G_1, G_2. □

The second phase: Let y_0 denote the entry point of proximity graph of Y. The second phase involves adding $|X|$ edges (x_i, y_0) for each $x_i \in X$ with weights $\delta(x_i, y_0)$ to MST of $\tilde{G}_\epsilon$, resulting in the final MST of G_ϵ. We denote the new $|X|$ edges into a sorted list $e_1, \cdots, e_{|X|}$ (line 1). Then, we traverse the edges in the MST of the proximity graph G based on ascending edge weights (line 4). Before inserting each edge, we check the sorted list $e_1, \cdots, e_{|X|}$ by considering edges with weights falling within the range of the previously traversed edge and the current edge (line 5-9). During both the insertion process of edges in the sorted list and MST, we verify that the two nodes of an edge if they do not belong to the same connected components (lines 6-8, 10-12). Following this, a Depth-First Search (DFS) is executed on the MST to obtain the window order list (lines 13-14).

Note that, at line 1, the edge from y_0 to $x_i \in X$ that is larger than $\max_T$ should not be included in the MST unless it is the shortest edges from y_0 to all data points in $x_i \in X$, where $\max_T$ is the maximum weight of edges in MST of $\tilde{G}_\epsilon$. To expedite the sorting process, a $\max_T$-range search can be conducted on $\tilde{G}_\epsilon$, with the query point set as y_0. We can find that the value of $\max_T$ needed is equal to the minimal value of ϵ for a connected ϵNG, as depicted in Table 1, the value of $\max_{MST}$ is typically small. Hence, the time complexity of this approach is similar to a k-nearest neighbor search in practice, approximately $O(\log |X|)$ [34]. Thereby, since lines 2-14 require $O(|X|)$ time, the overall time for the second phase is $O(|X|)$.

Algorithm 5: WindowOrder (G, y_0)

Input : a proximity graph $G = (V, E)$ of X and the entry point y_0 of the proximity graph of Y
Output : the window order list

1 sort edges (x_i, y_0) with ascending order of $\delta(x_i, y_0)$ for each $x_i \in V$, and denote them as $e_1, e_2, \cdots, e_{|X|}$;
2 $i \leftarrow 1$; $T \leftarrow \emptyset$;
3 assign each data point $u \in V \cup \{y_0\}$ to the set only contains itself;
4 **for** each edge (p, q) in G's MST in ascending order of $\delta(p, q)$ **do**
5 **while** $i \leq |V|$ and weight of edge $e_i = (u, v) \leq \delta(p, q)$ **do**
6 **if** u and v are in different sets **then**
7 merge the sets containing u and v;
8 add edge (u, v) into T;
9 $i \leftarrow i + 1$;
10 **if** p and q are in different sets **then**
11 merge the sets containing p and q;
12 add edge (p, q) into T;
13 use DFS to derive parent node $p[x_i]$ for each $x_i \in V$ in T rooted at y_0;
14 **return** $(p[x_i], x_i)$ **for** each $x_i \in X$ according to the DFS order;

The following theorem proves the correctness of this algorithm for obtaining the optimal join window order list.

Theorem 6.2: *Given the entry point y_0 of the proximity graph of Y, Algorithm 5 returns the optimal window order list with the minimum join cost when the given proximity graph G of X is a connected ϵ*NG.

Proof Sketch: We need to prove that the T acquired in Algorithm 5 is the MST of the ϵNG of dataset $|X| \cup y_0$, then we can utilize Lemma 6.1 to establish that the weight of the *MST* is equivalent to the minimum join cost. Since lines 4-5 guarantee the traversal of edges in $\{e_1, e_2, \cdots, e_{|X|}\}$ and the MST of G in ascending order, T is MST among these edges. Assuming that T is not MST of the connected ϵNG of dataset $|X| \cup \{y_0\}$, there must be at least one edge (u, v) in the ϵNG that is not in the MST of G but is included in the MST of the ϵNG. This implies $\delta(u, v) < w_{\max}(u \rightsquigarrow v)$, where $w_{\max}(u \rightsquigarrow v)$ is the maximum edge weight along the path from u to v in T. Since the edge (u, v) is absent in T, the path from u to v in T must involve two edges from y_0, denoted as (y_0, p) and (y_0, q), where $\delta(y_0, p) \leq \delta(y_0, q)$. Hence, we have $\delta(u, v) < \delta(y_0, q)$. However, because (u, v) is not part of the MST of G, and (y_0, q) must replace one of the edges from p to q in the MST of G to be present in T, we have $\delta(y_0, q) < w'_{\max}(p \rightsquigarrow q) \leq \delta(u, v)$, where $w'_{\max}(p \rightsquigarrow q)$ is the maximum edge weight along the path from p to q in the MST of G. This leads to a contradiction. □

7 k-similarity Join

In this section, we discuss how our ϵ-similarity join algorithm supports k-similarity join. Then, we discuss how our k-similarity join addresses maintenance issue for the proximity graph index.

7.1 k-similarity Join Algorithm

The k-similarity join $X \bowtie_k Y$ between two d-dimensional dataset X and Y is defined as $X \bowtie_k Y = \{(x_i, y_j) \in X \times Y \mid y_j \in TopK(x_i)\}$, where $TopK(x_i)$ is the set of top-k nearest data points of x_i in Y.

Algorithm 6: kJoinSlide ($\mathcal{G}, x_i, x_j, J_i, k, w$)

Input : an adjacent graph $\mathcal{G}$ of dataset Y, two data points x_i, x_j, the join result J_i of x_i and two parameters k, w

Output : the k-join result J_j of x_j in Y

```
1  Q ← J_i is a priority queue, sorted in ascending order based on δ(y_p, x_j) for every y_p ∈ Q;
2  Visit ← J_i; mark each u ∈ Q unexplored;
3  while at least one of the elements in Q is unexplored do
4      u ← the first unexplored element in Q; mark u explored;
5      for each v ∈ N_G(u) \ Visit do
6          push v into priority queue Q;
7          Visit ← Visit ∪ {v}; mark v unexplored;
8          while Q.size() > w do
9              remove the last element from Q;
10 return the first k elements in Q;
```

The main ideas used in our ϵ-similarity join algorithm, SimJoin, can be utilized to support k-similarity join, based on join window sliding and join window order selection. We discuss the key difference between ϵ-similarity join and k-similarity join. For ϵ-similarity join, $X \bowtie_\epsilon Y$, a continuous join window, J_i, for x_i, is bounded while sliding. In other words, suppose it slides to y_p, for $\delta(y_p, x_i) \leq \epsilon$, then it does not need to slide further from y_p to y_q and beyond if for $\delta(y_q, x_i) > \epsilon$. For k-similarity join, $X \bowtie_k Y$, it is difficult to determine whether the data points are included in the join window or not, i.e., it is a challenge for the sliding to ensure that no additional points in Y can be included in $TopK(x_i)$ before terminating. Hence, we propose to explore all adjacent points of a join window J_i to ensure the points in J_i are top-k nearest data points by there are no closer neighbors to x_i. Based on this, for k-similarity join, the strategies used for join window sliding and join window order selection can be used effectively.

We give the join window sliding algorithm for k-similarity join in Algorithm 6, which is the only part we need to modify to support for k-similarity join based on our ϵ-similarity join algorithm, SimJoin. It starts by enqueuing all data points from the join window of J_i into the priority queue, Q (line 1). Next, it traverses each node in Q to explore closer data points to x_j by examining its neighbors (lines 5-9). The size of Q is restricted to a maximum of w (lines 8-9), and the iteration concludes when no nodes in the queue have closer neighbors to x_j (line 3).

ϵ-similarity join vs k-similarity join: The main difference between k and ϵ is that the former indicates the number of nearest neighbors whereas the latter indicates the nearest neighbors in a range. The two serve a similar purpose but are different. From the viewpoint of k, the values of ϵ can vary from one dataset to another. Consider k as nSelectivity given in Exp. 3 in Section 8. As illustrated in Fig. 7, to have nSelectivity = 10, ϵ needs to be greater than 0.5 in Crawl, and needs to be less than 0.4 in Higgs.

Determining the value of k for the join may be straightforward as it limits the number of results. For selecting the value of ϵ, one way is to explore the average distance in a known dataset. For example, in the application of auto-tagging discussed in our introduction, the ϵ value can be obtained by the average distance among the documents that have the same labels. Another way to select ϵ for joins is to explore its relationship with nSelectivity (e.g., k).

7.2 Maintenance of Proximity Graph Index

Proximity graphs are one of the state-of-the-art approaches for approximate nearest neighbor search in vector databases, which offers a multitude of applications including information retrieval and recommendations. And there is a significant demand for updating these indices due to the continuous influx of large volumes of vector data across different platforms, e.g., over 500 hours of content are uploaded to YouTube every minute [41]. Thus, the maintenance of proximity graphs has emerged as an important issue.

The maintenance issue for a proximity graph index, $G_Y = (V_Y, E_Y)$, for a dataset Y is to deal with data insertion/deletion [40, 42, 44]. To deal with new data point insertion, the existing approaches will maintain it in a dataset X, and build a different small delta proximity graph index, $G_X = (V_X, E_X)$, to maintain such new points; and to deal with data point deletion from Y, the existing approaches, for each deleted y_p from Y, will connect all its in-neighbors to its out-neighbors in G_Y. It is important to note that periodic global rebuilding of the entire G_Y is necessary and is triggered (1) when the cumulative number of data points in G_X surpasses a specific threshold (e.g., 1-10% of the total index size) and (2) the number of deleted data points in G_Y exceeds a substantial threshold. Such global rebuildings are resource-intensive and time-consuming. As shown in Table 4, global rebuilding of an index for Higgs dataset peaked at 22.4 GB memory and required 2,962 seconds when it is executed using 32 threads, despite the original size of Higgs is 3.8 GB. And the main cost is to rebuild G_Y with G_X.

To deal with data insertions: To rebuild a new proximity index $\mathbb{G}_Y$ for $Y \cup X$ for data points insertion, first, we conduct k-similarity join, $X \bowtie_k Y$, where G_X has been built for X and G_Y has been built for Y. The k-similarity join results are the potential out-neighbors of each node across Y and X. Second, together with G_X, G_Y and the k-similarity resultss, we can build a new $\mathbb{G}_Y$ easily, without rebuilding $\mathbb{G}_Y$ from scratch for $Y \cup X$. Third, we can apply various pruning techniques by different approaches to obtain the final index.

To deal with data deletions: When a data point is deleted, we mark it on the proximity graph, G_Y, to ensure its exclusion from k-ANN searches, but do not remove it from G_Y. Upon reaching a specified threshold of deletions, we perform batch updates on G_Y. For batch updates, it is to deal with those data points, y_q, in G_Y marked deleted, because the out-degree of the in-neighbors of y_q can drop below a threshold, e.g., m. The impacts on the out-neighbors of y_q are minor, as their k-ANN on G_Y remains unaffected. To swiftly address the decrease in the out-degree of the in-neighbors of such deleted nodes in G_Y, we employ the join window sliding algorithm instead of adding new edges from the in-neighbors of y_q to out-neighbors of y_q. Specifically, for each in-neighbor y_x of a deleted data point, y_q, on G_Y, we slide the join window $J_x = N_{G_Y}(y_x) \setminus Y_D$ to the k-similarity join window J_x of y_x, where Y_D is the set of data points deleted from Y, which J_x should exclude. This process can be accomplished by invoking kJoinSlide $(\mathcal{G}, y_x, y_x, N_{G_Y}(y_x) \setminus Y_D, k, w)$ for $k > m$. We can easily do it by modifying line 8 to"**while** $|Q \setminus Y_D| > w$ **do**" and line 10 to "**return** the first k elements in $Q \setminus Y_D$". Next, various pruning techniques, relying on those proposed in different proximity graph methods, can be employed on J_x for each in-neighbor y_x. For instance, when maintaining NSG [14], an edge (u, v) exists only if there is no edge (u, w) such that $\delta(u, w) < \delta(u, v)$ and $\delta(v, w) < \delta(u, v)$.

8 Experiments

In this section, we conduct extensive experiments on both real-world and synthetic datasets and report our findings.

Datasets: We employ 6 real-world and 2 synthetic datasets with different numbers of dimensions/-points, the real-world datasets are from diverse applications including image (SIFT [1], GIST [1]),

Table 2. Statistics of Datasets

Dataset	dim.	# of points	Type	Source	Dataset	dim.	# of points	Type	Source
Rand	64	100,000	Synthetic	U(0,1)	GloVe	100	1,183,514	Text	[35]
Gauss	64	100,000	Synthetic	N(0,1)	Crawl	300	1,989,995	Text	[2]
Msong	420	992,272	Audio	[6]	SIFT	128	10,000,000	Image	[1]
GIST	960	1,000,000	Image	[1]	Higgs	29	11,000,000	Particles	[5]

Table 3. Runtime and average recall of four algorithms on different datasets (Exp. 2)

Dataset	ϵ-self similarity join (ϵ=0.4)								ϵ-similarity join (ϵ=0.4)								
	XJoin		VBASE		FGF-Hilbert		SimJoin		XJoin		VBASE		FGF-Hilbert		SimJoin		
	recall	time (s)	recall	time (s)	recall	time (s)	recall	time (s)	recall	time (s)	recall	time (s)	recall	time (s)	recall	time (s)	pre-ratio
Rand	0.95688	19.368	0.99233	16.343	1	48.989	1	1.8626	0.95752	19.536	0.99126	16.293	1	49.035	0.99298	6.3837	0.00623%
Gauss	0.95763	15.788	0.99583	13.272	1	18.234	1	1.1332	0.95727	16.418	0.99583	13.394	1	18.011	1.0000	1.0223	0.03870%
Msong	0.97613	765.82	0.99055	528.57	1	4904.1	0.99368	135.89	0.97541	783.35	0.99035	528.54	1	4912.7	0.99052	178.20	0.00066%
GIST	0.97308	2097.1	0.98825	1827.8	1	10911	0.99091	120.62	0.97739	2196.7	0.99135	1775.7	1	11295	0.99256	355.38	0.00032%
GloVe	0.96599	280.70	0.99264	225.13	1	882.29	1	18.158	0.96586	287.76	0.99261	222.77	1	897.89	0.99788	15.203	0.01474%
Crawl	0.97304	1063.1	0.98753	838.91	1	6072.2	0.99998	47.114	0.97321	1048.7	0.98659	842.63	1	6138.9	0.99084	202.72	0.00201%
SIFT	0.98138	14819	0.99251	7679.5	1	166050	0.99564	5232.8	0.98023	15479	0.99222	9418.4	1	166931	0.99344	5532.6	0.00023%
Higgs	0.98659	8258.2	0.99031	4553.2	1	38724	0.99014	2044.6	0.98659	10948	0.99031	6339.2	1	38778	0.99067	2020.3	0.00109%

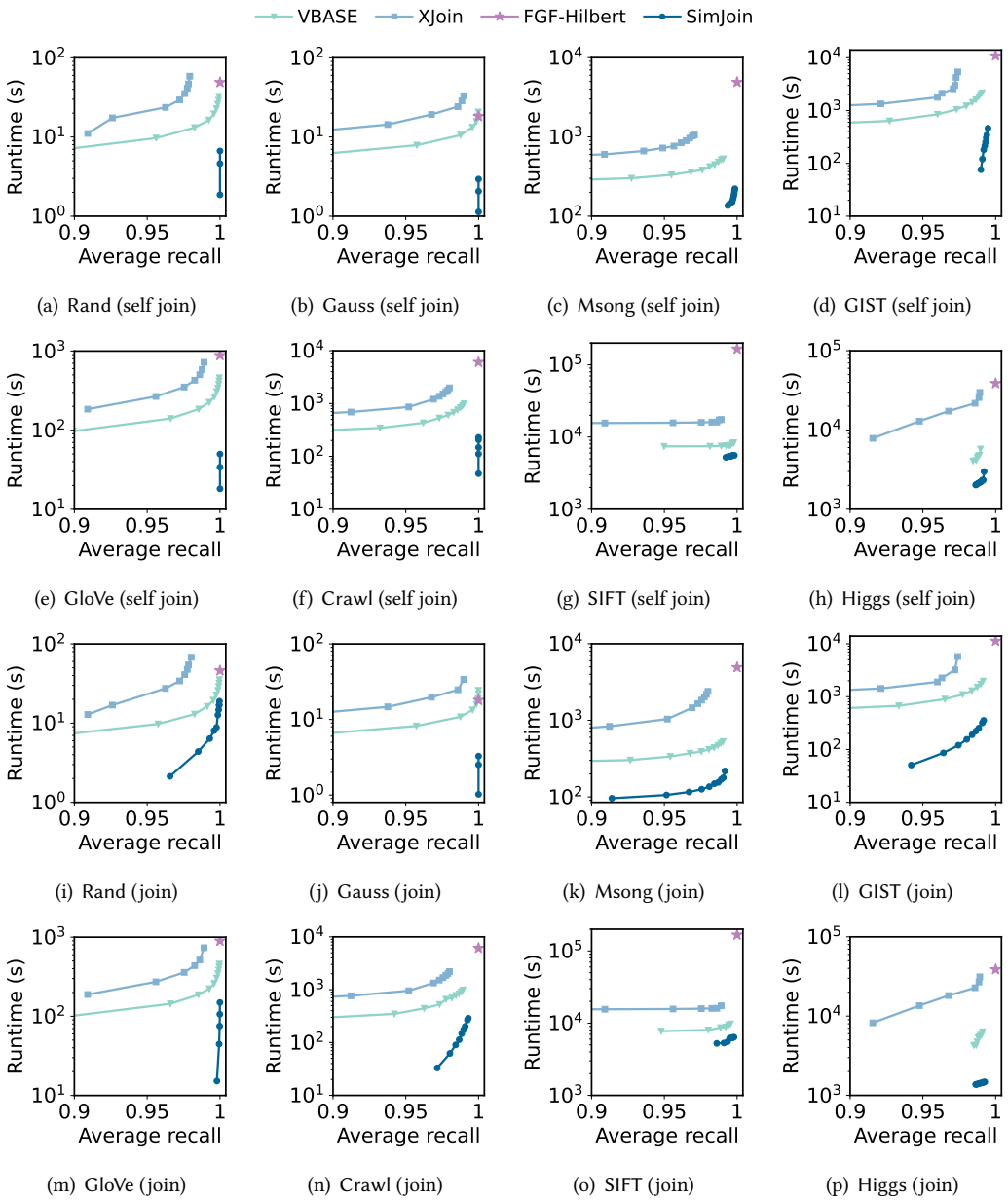
audio (Msong [6]), text (GloVe [35], Crawl [2]) and high-energy physics (Higgs [5]). The summary can be found in Table 2, with the number of dimensions (dim.) and the number of data points (# of points). To evaluate the algorithms, the datasets are randomly divided into two parts of equal size. Self-similarity joins are evaluated on one of the split datasets, while similarity joins are assessed on both datasets.

Algorithms: We conduct comparisons between our approaches, named as SimJoin, and three state-of-the-art methods, including approximate similarity join methods (VBase [51] and XJoin [46]), and exact similarity join method (FGF-Hilbert [36]). (1) **VBase** [51] is the state-of-the-art in approximate similarity join approaches. It executes the join algorithm by treating each data point in one dataset as an individual range query and implements an efficient range filter by integrating it with an index scan operator. (2) **XJoin** [46] is the latest approach for approximate similarity join. It employs a learning-based technique to predict whether a data point possesses a sufficient number of join results. Specifically, for $X \bowtie_\epsilon Y$, where X is the larger dataset and Y is the smaller dataset to join. First, XJoin is trained to predict if a query vector $y_i \in Y$ contains sufficient neighbors within the ϵ range in X. Second, the query vectors identified as having adequate neighbors are formed as a new dataset Y'. Finally, the LSH-based join algorithm is performed between X and Y'. As demonstrated in [46], XJoin outperforms various LSH-based approaches. (3) **FGF-Hilbert** [36] is the state-of-the-art method of exact similarity join, relying on the Epsilon Grid Order technique and enhancing runtime efficiency by improving data locality, thereby rendering the algorithms cache-oblivious. (4) **SimJoin**: our join algorithm contains Algorithm 2 for two distinct datasets and a self-join algorithm. In the self-join algorithm, each data point inherently includes itself in the results due to zero distance. Hence, for every data point in X, we can initiate the sliding of the join window starting with itself. In other words, for each data point $x_i \in X$, we can invoke Algorithm 3 via JoinSlide $(G, x_i, x_i, \{x_i\}, \epsilon, w)$, where G is proximity graph of X. For VBase and our approach, we use the same proximity graph, NSG [14], for comparisons in each dataset.

In the comparisons of join time, we exclude the index construction time for VBase, XJoin and our approach as the index constructed is independent of the ϵ value selected. The index can be constructed offline. We include the index construction time for FGF-Hilbert, because FGF-Hilbert requires preprocessing and cannot reuse the same index constructed for different ϵ values.

We evaluate the performance of the similarity join algorithms using a single thread, and all results are averaged over 5 runs.

Performance metrics: We focus on two metrics for evaluating the join results: average recall



(a) Rand (self join) (b) Gauss (self join) (c) Msong (self join) (d) GIST (self join)

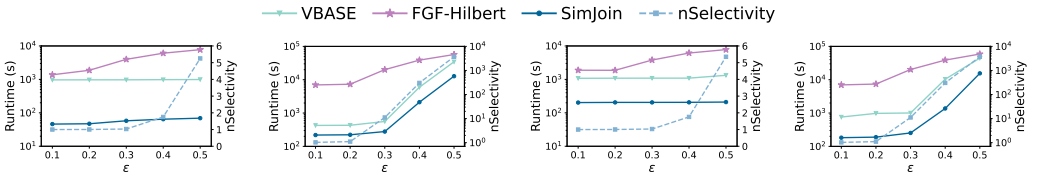
(e) GloVe (self join) (f) Crawl (self join) (g) SIFT (self join) (h) Higgs (self join)

(i) Rand (join) (j) Gauss (join) (k) Msong (join) (l) GIST (join)

(m) GloVe (join) (n) Crawl (join) (o) SIFT (join) (p) Higgs (join)

Fig. 6. ϵ-join comparisons with current three join algorithms (Runtime v.s. Recall) when $\epsilon = 0.4$ (Exp. 1)

and average precision, as detailed in Section 2. Given that the join results produced by the four algorithms compared in our experiments are all based on exact distance comparisons, their average precision are all 1. Therefore, our comparisons focus solely on the average recall of these algorithms.
Experimental environments: The experiments are conducted on a CentOS Linux server with Intel(R) Xeon(R) Silver 4215 CPU @ 2.50GHz with 128GB memory. All algorithms are implemented in C++11. The code is compiled with g++ 11.4.1 under O3 optimization.



(a) Crawl (self join) (b) Higgs (self join) (c) Crawl (join) (d) Higgs (join)

Fig. 7. Runtime at 0.99 average recall when vary ϵ (Exp. 3)

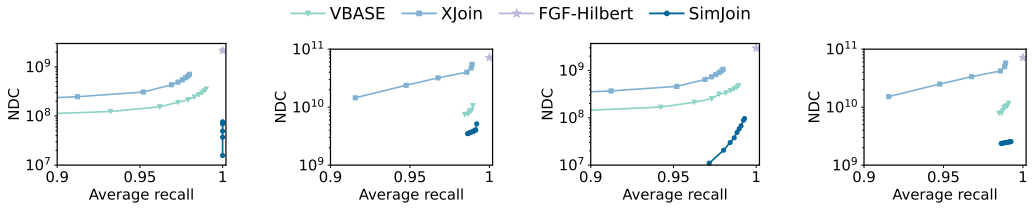
Exp. 1. Effects of the parameters in join algorithms: In this evaluation, we keep the dataset and ϵ fixed at 0.4, and then adjust various parameters in different algorithms. In our join algorithms, we modify the parameter w. In VBASE, their search algorithms are based on beam search, so we adjust the beam width in their algorithms. For XJoin, we vary the value of τ in their algorithm, a parameter used to identify ground-truth negative training samples, where a higher τ value can enhance efficiency at the loss of quality. FGF-Hilbert does not have adjustable parameters and is represented as a single point in the figures. The results are shown in Fig. 6, where lower and further to right curves indicate better performance.

The results reveal that our algorithms outperform the three approaches significantly, achieving higher average recall with much-reduced runtime. Moreover, the initial points of the curves for our algorithms are notably to the right compared to the other approximate similarity join algorithms, showcasing that our algorithms can efficiently identify nearly all join results without heavy time cost on searches. The results further indicate that our algorithms show reduced dependency on parameters, as evidenced by the narrower range of average recall values of our algorithms' curves.

Exp. 2. Overall performance of ϵ-similarity join: As shown in Table 3, our join algorithms, denoted as SimJoin, including the ϵ-self similarity join algorithm and the ϵ similarity join algorithm, is compared to three state-of-the-art approaches when we set ϵ equals 0.4. Compared to VBASE, our approach demonstrates up to 1 order of magnitude speedups in runtime with superior quality. Specifically, SimJoin achieves speedups of 8.8x, 11.7x, 3.9x, 15.2x, 12.4x, 17.8x, 1.5x, 2.2x in self-similarity join, and 2.6x, 13.1x, 3.0x, 5.0x, 14.7x, 4.2x, 1.7x, 3.1x in similarity join on the Rand, Gauss, Msong, GIST, GloVe, Crawl, SIFT, Higgs datasets respectively. Compared to the state-of-the-art method in exact similarity join, our algorithm achieves up to speedups of over 2 orders of magnitude in runtime while maintaining an average loss of less than 1% in join results. We further present the ratio of the runtime of the join window order selection to the total runtime of SimJoin on ϵ-similarity join, denoted as "pre-ratio" in Table 3, to provide the time breakdown of our algorithm. The time of our join window order selection for many datasets, such as Rand in the first row, is notably less than 1ms. To ensure precise reporting of such times, we conducted the join window selection algorithm 100 times, and reported the average time. The results indicate that the order selection phase constitutes a negligible portion of the total time of our SimJoin.

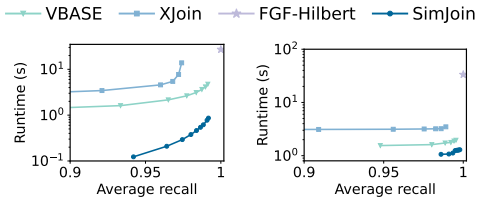
Exp. 3. Vary the value of ϵ: In this experiment, we vary the value of ϵ to compare the runtime of different algorithms when achieving an average recall of 0.99. The results are shown in Fig. 7, which does not include XJoin, as it cannot achieve 0.99 recall in either the Crawl or Higgs datasets. We use nSelectivity to denote the average number of join results for each data point in a dataset, defined as the value of $|X|$ multiplied by the selectivity of the join operation, i.e., the nSelectivity of the join operation $X \bowtie Y$ is given by $|X| \cdot \frac{|X \bowtie Y|}{|X \times Y|}$.

The results in the figures indicate that the runtime of our algorithms increases linearly with the value of nSelectivity, demonstrating good scalability as the number of join results increases.



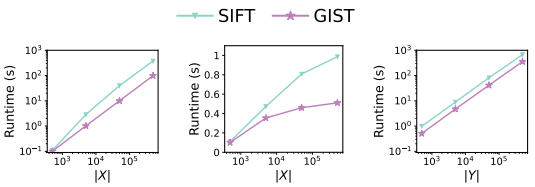
(a) Crawl (self join) (b) Higgs (self join) (c) Crawl (join) (d) Higgs (join)

Fig. 8. NDC v.s. Recall when $\epsilon = 0.4$ (Exp. 4)



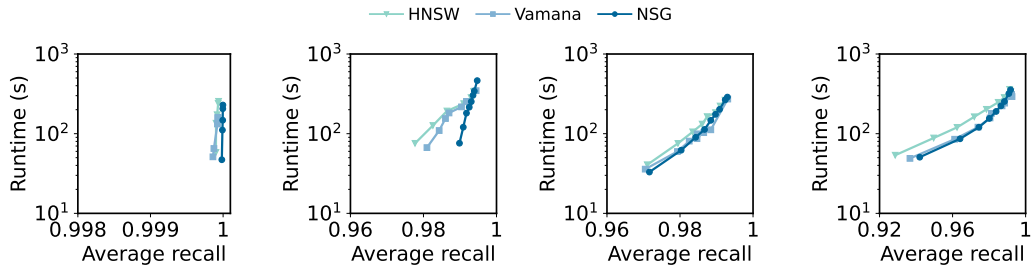
(a) GIST (b) SIFT

Fig. 9. base and query data join (Exp. 5)



(a) self join (b) join at $|Y| = 500$ (c) join at $|X| = 500,000$

Fig. 10. Varying dataset size (0.99 average recall) (Exp. 6)



(a) Crawl (self join) (b) GIST (self join) (c) Crawl (join) (d) GIST (join)

Fig. 11. ϵ-join by different proximity graphs (Exp. 7)

Besides, our join algorithms consistently remain much shorter than that of VBASE and FGF-Hilbert across different values of ϵ.

Exp. 4. Number of distance computations during join: In this part, we conduct experiments to compare our approach with others based on a new metric, the number of distance computations (NDC), using Crawl and Higgs datasets. As shown in Fig. 8, our approach has a notable enhancement in terms of NDC over the existing state-of-the-art approaches, consistent with the results of Exp. 2.

Exp. 5. Join between base and query data: In this part, we use the base data and query data as two separate datasets for join. We evaluate our approach against three other approaches using SIFT and GIST datasets sourced from the ANN benchmark[1]. The results, shown in Fig. 9, demonstrate that, akin to the findings in Exp. 2, our approach significantly outperforms the existing methods.

Exp. 6. Vary the size of datasets for join: To evaluate the scalability of our proposed SimJoin method, we conduct experiments on SIFT and GIST datasets by (i) varying the size of dataset ($|X|$)

[1]https://ann-benchmarks.com/

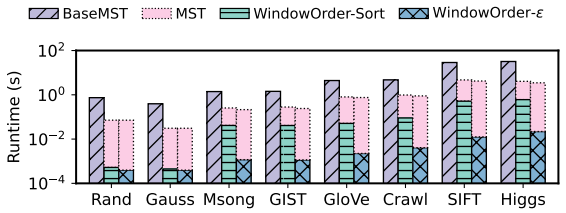


Fig. 12. Runtime of join window order algorithms (Exp. 8)

from 500 to 500, 000 in self join, (ii) varying the size of base data ($|X|$) from 500 to 500, 000 while keeping the size of the query data ($|Y|$) as 500, and (iii) varying the size of the query data ($|Y|$) from 500 to 500, 000 while keeping the size of the base data ($|X|$) constant as 500, 000. The results of runtime at 0.99 average recall are shown in Fig. 10. Notably, the runtime shows a linear increase or a slower rate of growth in comparison to the expansion of the dataset size, which validates that our method exhibits strong scalability when either the base data or query data size expands.

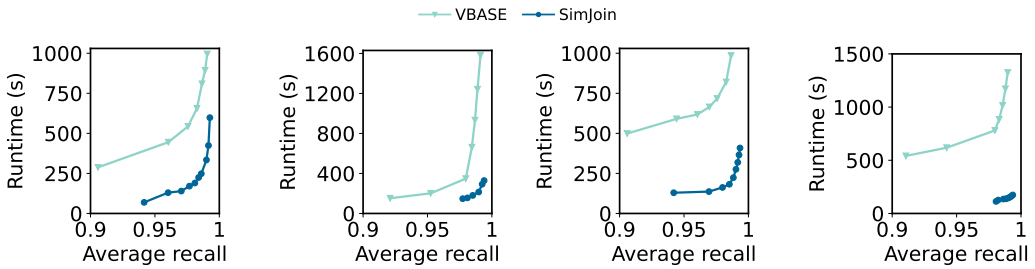
Exp. 7. Comparisons using different proximity graphs: Existing proximity graph indexes for k-ANN searches can seamlessly integrate with our join algorithm. Here, we compare three state-of-the-art proximity graph index methods for ϵ-similarity joins on Crawl and GIST datasets when $\epsilon = 0.4$. As shown in Fig. 11, HNSW does not perform well due to higher edge distances, which impacts adjacency quality according to the analysis in Theorem 5.1. The higher edge distances in HNSW are because it is built via incremental insertion, which means candidate neighbors are only selected among previously inserted nodes. Vamana and NSG exhibit similar performance, since they differ mainly in pruning strategies that affect a small edge subset in the graph.

Exp. 8. Comparisons among join window order determination algorithms: We compare our two advanced join window order determination algorithms: MST + WindowOrder-Sort (Algorithms 4+5) and WindowOrder-ϵ (Algorithms 4+5, with line 2 in Algorithm 5 replaced by an approximate $\max_T$-range query), against the baseline algorithm BaseMST. The details of all these algorithms are provided in Section 6.2.

The runtime of the three algorithms is shown in Fig. 12. We can find that it is essential to accelerate the running time of the baseline approach, as it consumes a significant portion of the total runtime. For example, in GloVe dataset, the runtime of the baseline approach for determining the join window order is 4.4s, which accounts for over 20% of the total join processing time. For the two advanced algorithms, MST step can be completed offline before the join operation. Even with this preprocessing, the advanced algorithms outperform the baseline by more than 1 order of magnitude in runtime. Moreover, WindowOrder-Sort and WindowOrder-ϵ are faster than the baseline by over 2 and 3 orders of magnitude, respectively. The join costs for BaseMST and MST + WindowOrder-Sort are the same, while the difference in join costs between MST + WindowOrder-ϵ and the others are less than 0.01% across all datasets.

Exp. 9. Comparisons in k-similarity join: We evaluate our k-similarity join algorithm by comparing it to VBase. Although VBase does not directly support k-similarity join, we utilized their k-nearest neighbor search to facilitate the join by treating each data point in the smaller dataset as a query point, following a similar method to their ϵ-similarity join. The results on Crawl and Higgs datasets are depicted in Fig. 13, showcasing that the runtime of our algorithm is significantly lower than VBase's while achieving similar or even better average recall.

Exp. 10. Index maintenance comparisons: We follow the experimental setup of FreshDiskANN [40] to evaluate our algorithms for index maintenance on Crawl and Higgs datasets. For insertion, we



(a) Crawl (self join) (b) Higgs (self join) (c) Crawl (join) (d) Higgs (join)

Fig. 13. k-similarity join comparisons when $k = 10$ (Exp. 9)

Table 4. NSG index maintenance comparisons (Exp. 10)

Dataset	global rebuilding		FreshDiskANN		k-similarity join	
	peak memory	time (s)	peak memory	time (s)	peak memory	time (s)
Rand	430.72 MB	129.624	274.64 MB	37.3372	249.95 MB	33.3368
Gauss	419.21 MB	131.991	166.51 MB	29.9972	167.75 MB	29.2273
Msong	3.9465 GB	2632.23	1.6452 GB	343.617	1.6432 GB	327.254
GIST	9.0220 GB	6632.01	5.5203 GB	960.500	5.5196 GB	906.157
GloVe	4.2359 GB	958.089	1.1092 GB	875.349	1.0849 GB	864.672
Crawl	7.3334 GB	1963.67	2.4671 GB	674.069	2.4202 GB	585.061
SIFT	17.257 GB	3269.74	8.1037 GB	1989.38	8.0456 GB	1825.94
Higgs	22.394 GB	2961.66	4.7927 GB	769.701	4.7922 GB	744.140

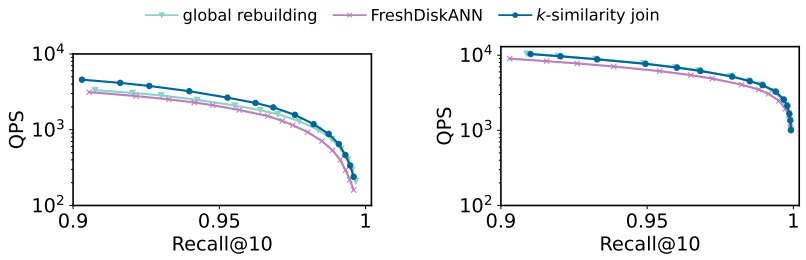
divide the original dataset into two parts with equal size. One part is used for the initial index construction, and the other part is used for insertion; for deletion, we randomly select points for removal following FreshDiskANN. Specifically, we randomly remove 10% of data points from one part of the dataset and then insert the same number of data points from the other part. Our algorithm is compared to (i) an approach based on global rebuilding, where the index is rebuilt globally after the complete deletion and insertion process; and (ii) FreshDiskANN [40], the state-of-the-art graph-based index maintenance approach. Our algorithm and FreshDiskANN utilize only one thread for supporting maintenance, while global rebuilding employs all available threads in the servers for processing. The comparisons are based on the NSG graph index, where the reported times include the index rebuilding time for global rebuilding and the total time taken for index merging during data point insertion and batch updates for data point deletion in FreshDiskANN and our algorithms.

Regarding the global rebuilding, the results, as depicted in Table 4, show that our maintenance algorithms achieve significantly higher efficiency and smaller peak memory. And Fig. 14 shows that our final graph index is even better than that of global rebuilding, attributed to the higher quality of the k-nearest neighbors of each data point in our algorithms compared to NN-descent.

For FreshDiskANN, although it has comparable memory usage and processing time to ours, in terms of the quality of the index maintained, our approach outperforms FreshDiskANN, as shown in Fig. 14, because FreshDiskANN focuses on delta changes within the index to be maintained.

9 Related Work

Many works focus on exact high-dimensional similarity joins, including ϵ-similarity join [7, 9, 25, 36, 39] and k-NN join [8, 48]. However, the so-called high dimension may be different from



(a) Crawl (b) Higgs

Fig. 14. ANNS performance on final NSG (Exp. 10)

time to time. Most of these works [8, 25, 39, 48] consider dimensions around 40 as high, whereas current vector databases often feature significantly more dimensions, e.g., the number of dimensions in GIST is 960. Among them, the state-of-the-art methods for exact high-dimensional similarity join [22, 23, 29, 31, 36] involve two steps: filtering and refinement. In the filtering step, candidate pairs of vectors that might be part of the final join results are selected through indexing or sorting. In the refinement step, the exact distance between candidate pairs of two vectors is computed. Filtering using index structures, such as R-trees [9], to support the join predicate can be costly in index construction. Therefore, these approaches opt to sort the dataset based on a function that supports the join predicate, for example, Epsilon Grid Order [7].

In the realm of approximate high-dimensional similarity search, existing approaches treat each data point in a dataset as an individual approximate ϵ-range query. These queries are then supported by a locality-sensitive hashing (LSH) based index [4, 20, 27, 37, 46, 49, 50] or a proximity graph index [44, 51]. At a high level, LSH-based approaches support approximate similarity joins by following three steps: (i) first projecting each vector into a hash value; (ii) next executing an equi-join on the pairs that collide under the same hash value to establish candidate join results; (iii) finally, exact distance computations are conducted to derive the final join results. Therefore, the effectiveness of these approaches heavily relies on the hash functions in step (i), i.e., mapping similar vectors to the same hash value, which is also the key factor that highly impacts the performance in LSH-based methods for approximate nearest neighbor search. However, different to recent graph [34, 47] and partition [10, 16, 18] based approaches, LSH-based approaches do not take data distribution into consideration, which often leads to a significant performance decline in real-world datasets [45], where data is unevenly distributed data.

10 Conclusion

In this paper, we introduce a novel join algorithm designed to support approximate similarity joins in high-dimensional spaces. Our approach leverages the intrinsic properties of the join operation, utilizing results from processed data points to expedite the entire join process. Our extensive experiments demonstrate that our proposed join algorithm outperforms all existing state-of-the-art methods for approximate similarity joins by up to one order of magnitude with superior quality of join results. Our future work will focus on developing a distributed algorithm based on our current approach to offer a more scalable approximate similarity join method.

Acknowledgements

This work was supported by the Research Grants Council of Hong Kong, China, No.14205520, and the National Natural Science Foundation of China, No.62002274.

References

[1] 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/.

[2] 2023. Common Crawl. https://commoncrawl.org/.

[3] W. Ackermann. 1928. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Ann.* 99 (1928), 118–133.

[4] Martin Aumüller and Matteo Ceccarello. 2022. Implementing Distributed Similarity Joins using Locality Sensitive Hashing. In *EDBT 2022*. OpenProceedings.org, 1:78–1:90.

[5] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. 2014. Searching for exotic particles in high-energy physics with deep learning. *Nature communications* 5, 1 (2014), 4308.

[6] Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. In *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011*. University of Miami, 591–596.

[7] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. 2001. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. ACM, 379–388.

[8] Christian Böhm and Florian Krebs. 2004. The *k*-Nearest Neighbour Join: Turbo Charging the KDD Process. *Knowl. Inf. Syst.* 6, 6 (2004), 728–749.

[9] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient Processing of Spatial Joins Using R-Trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*. ACM Press, 237–246.

[10] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *NeurIPS 2021*. 5199–5212.

[11] Sheng Chen, Akshay Soni, Aasish Pappu, and Yashar Mehdad. 2017. DocTag2Vec: An Embedding Based Multi-label Learning Approach for Document Tagging. In *Rep4NLP@ACL 2017*. Association for Computational Linguistics, 111–120.

[12] B. Delaunay. 1934. Sur la sphère vide. *Bull. Acad. Sci. URSS* 1934, 6 (1934), 793–800.

[13] Rex A Dwyer. 1989. Higher-dimensional Voronoi diagrams in linear expected time. In *Proceedings of the fifth annual symposium on Computational geometry*. 326–333.

[14] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

[15] Harold N. Gabow and Robert Endre Tarjan. 1983. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*. ACM, 246–251.

[16] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3 (2024), 167.

[17] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 855–864.

[18] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020 (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 3887–3896.

[19] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. IEEE Computer Society, 5713–5722.

[20] Xiao Hu, Ke Yi, and Yufei Tao. 2019. Output-Optimal Massively Parallel Algorithms for Similarity Joins. *ACM Trans. Database Syst.* 44, 2 (2019), 6:1–6:36.

[21] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing*. ACM, 604–613.

[22] Edwin H. Jacox and Hanan Samet. 2008. Metric space similarity joins. *ACM Trans. Database Syst.* 33, 2 (2008), 7:1–7:38.

[23] Dmitri V. Kalashnikov. 2013. Super-EGO: fast multi-dimensional similarity join. *VLDB J.* 22, 4 (2013), 561–585.

[24] Dmitri V. Kalashnikov and Sunil Prabhakar. 2007. Fast similarity join for multi-dimensional data. *Inf. Syst.* 32, 1 (2007), 160–177.

[25] Nick Koudas and Kenneth C. Sevcik. 2000. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. *IEEE Trans. Knowl. Data Eng.* 12, 1 (2000), 3–18.

[26] D. T. Lee and Bruce J. Schachter. 1980. Two algorithms for constructing a Delaunay triangulation. *Int. J. Parallel Program.* 9, 3 (1980), 219–242.

[27] Hangyu Li, Sarana Nutanong, Hong Xu, Chenyun Yu, and Foryu Ha. 2019. C2Net: A Network-Efficient Approach to Collision Counting LSH Similarity Join. *IEEE Trans. Knowl. Data Eng.* 31, 3 (2019), 423–436.

[28] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488.

[29] Youzhong Ma, Shijie Jia, and Yongxin Zhang. 2017. A novel approach for high-dimensional vector similarity join query. *Concurr. Comput. Pract. Exp.* 29, 5 (2017).
[30] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.
[31] Ahmed Metwally and Christos Faloutsos. 2012. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *Proc. VLDB Endow.* 5, 8 (2012), 704–715.
[32] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *27th Annual Conference on Neural Information Processing Systems 2013*. 3111–3119.
[33] Nasser M Nasrabadi and Robert A King. 1988. Image coding using vector quantization: A review. *IEEE Transactions on communications* 36, 8 (1988), 957–971.
[34] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proc. ACM Manag. Data* 1, 1 (2023), 54:1–54:27.
[35] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, 1532–1543.
[36] Martin Perdacher, Claudia Plant, and Christian Böhm. 2019. Cache-oblivious High-performance Similarity Join. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019*. ACM, 87–104.
[37] Sébastien Rivault, Mostafa Bamha, Sébastien Limet, and Sophie Robert. 2022. A Scalable Similarity Join Algorithm Based on MapReduce and LSH. *Int. J. Parallel Program.* 50, 3-4 (2022), 360–380.
[38] Raimund Seidel. 1995. The Upper Bound Theorem for Polytopes: an Easy Proof of Its Asymptotic Version. *Comput. Geom.* 5 (1995), 115–116.
[39] Kyuseok Shim, Ramakrishnan Srikant, and Rakesh Agrawal. 1997. High-Dimensional Similarity Joins. In *ICDE*. IEEE Computer Society, 301–311.
[40] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *CoRR* abs/2105.09613 (2021).
[41] SOAX. 2024. *How many hours of video are uploaded to YouTube each minute?* https://soax.com/research/how-many-hours-of-video-are-uploaded-to-youtube-every-minute
[42] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming Similarity Search over one Billion Tweets using Parallel Locality-Sensitive Hashing. *Proc. VLDB Endow.* 6, 14 (2013), 1930–1941.
[43] Yukihiro Tagami. 2017. AnnexML: Approximate Nearest Neighbor Search for Extreme Multi-label Classification. In *SIGKDD*. ACM, 455–464.
[44] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD '21*. ACM, 2614–2627.
[45] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (2021), 1964–1978.
[46] Yifan Wang, Vyom Pathak, and Daisy Zhe Wang. 2024. Xling: A Learned Filter Framework for Accelerating High-Dimensional Approximate Similarity Join. *CoRR* abs/2402.13397 (2024).
[47] Shuo Yang, Jiadong Xie, Yingfan Liu, Jeffrey Xu Yu, Xiyue Gao, Qianru Wang, Yanguo Peng, and Jiangtao Cui. 2024. Revisiting the Index Construction of Proximity Graph-Based Approximate Nearest Neighbor Search. *CoRR* abs/2410.01231 (2024).
[48] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. 2007. Efficient index-based KNN join processing for high-dimensional data. *Inf. Softw. Technol.* 49, 4 (2007), 332–344.
[49] Chenyun Yu, Sarana Nutanong, Hangyu Li, Cong Wang, and Xingliang Yuan. 2017. A Generic Method for Accelerating LSH-Based Similarity Join Processing. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 712–726.
[50] Xingliang Yuan, Xinyu Wang, Cong Wang, Chenyun Yu, and Sarana Nutanong. 2017. Privacy-Preserving Similarity Joins Over Encrypted Data. *IEEE Trans. Inf. Forensics Secur.* 12, 11 (2017), 2763–2775.
[51] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023*. USENIX Association, 377–395.

Received October 2024; revised January 2025; accepted February 2025