DiskJoin: Large-scale Vector Similarity Join with SSD

Yanqi Chen
University of Massachusetts Amherst
yanqichen@cs.umass.edu

Xiao Yan
Centre for Perceptual and Interactive Intelligence
yanxiaosunny@gmail.com

Alexandra Meliou
University of Massachusetts Amherst
ameli@cs.umass.edu

Eric Lo
Chinese University of Hong Kong
ericlo@cse.cuhk.edu.hk

Metric	DiskANN [48]	DiskJoin [this paper]
Total time (sec)	130, 107	**1,185**
Disk time (sec)	93, 417	**198**
Disk traffic (GB)	115, 854	**265**

Figure 1: Profiling results for a baseline solution—using the state-of-the-art SSD-based vector index DiskANN [48] to perform vector similarity join—and our DiskJoin on the Bi-gANN100M dataset at 90% recall. Both methods use a memory size that is 10% of the dataset size.

Abstract

Similarity join—a widely used operation in data science—finds all pairs of items that have distance smaller than a threshold. Prior work has explored distributed computation methods to scale similarity join to large data volumes but these methods require a cluster deployment, and efficiency suffers from expensive inter-machine communication. On the other hand, disk-based solutions are more cost-effective by using a single machine and storing the large dataset on high-performance external storage, such as NVMe SSDs, but in these methods the disk I/O time is a serious bottleneck. In this paper, we propose DiskJoin, the first disk-based similarity join algorithm that can process billion-scale vector datasets efficiently on a single machine. DiskJoin improves disk I/O by tailoring the data access patterns to avoid repetitive accesses and read amplification. It also uses main memory as a dynamic cache and carefully manages cache eviction to improve cache hit rate and reduce disk retrieval time. For further acceleration, we adopt a probabilistic pruning technique that can effectively prune a large number of vector pairs from computation. Our evaluation on real-world, large-scale datasets shows that DiskJoin significantly outperforms alternatives, achieving speedups from 50× to 1000×.

Keywords

Vector, Similarity join, Disk-based processing

1 Introduction

In machine learning workflows, it is common practice to use models to map objects with complex semantics (e.g., text, images, videos,

Authors' Contact Information: Yanqi Chen, University of Massachusetts Amherst, , yanqichen@cs.umass.edu; Xiao Yan, Centre for Perceptual and Interactive Intelligence, , yanxiaosunny@gmail.com; Alexandra Meliou, University of Massachusetts Amherst, , ameli@cs.umass.edu; Eric Lo, Chinese University of Hong Kong, , ericlo@cse.cuhk.edu.hk.

proteins) to vector embeddings and manage these objects via embeddings [35, 42, 51]. For instance, convolution neural networks (CNNs) map images to embeddings [35], transformer-based large language models (LLMs) map text segments to embeddings [42], and recommendation models map users and products to embeddings [51]. On these vector embeddings, similarity—e.g., measured via Euclidean distance or cosine similarity—is a crucial notion. For example, close similarity of two vector embeddings is used to indicate that two images are similar, that two text segments have similar meanings, or that a user may like a product.

Vector similarity join. Similarity self-join (SSJ), which seeks all pairs of similar data points, is a well-established problem that has been extensively studied in the literature. SSJ for vector data lies at the core of many applications like near-duplicate detection [1, 17, 27], outlier detection [10], and more. We give three practical examples and then provide a formal problem definition.

Video deduplication [16]: A video website maintains a large number of videos, but many are similar to each other [52, 58]. For example, the same clip may be uploaded by different users or encoded in different formats; highly-overlapping segments may be cropped from one video. Identifying and removing these duplicates can save resources [59] and avoid repetitive content recommendations. Similar videos (as well as other formats like images or text) can be identified by performing SSJ over their vector embeddings.

Training data deduplication [1]: Dataset deduplication is a crucial step in machine learning pipelines, especially when training large language models (LLMs), where the training datasets often contain billions of examples. In particular, semantic deduplication—which identifies pairs of data that are semantically similar but not exactly identical—has been shown to significantly reduce dataset size, while maintaining, or even improving, model performance. A practical approach to enabling semantic deduplication at scale is to apply similarity self-join over vector embeddings of the data.

Outlier detection [11]: An object is considered an outlier if it significantly deviates from other objects. With the embedding vectors, the deviation can be quantified using the number of neighbors that are within a distance threshold from a vector [39]. If a vector has only a few similar neighbors, we can conclude that its corresponding object is an outlier, which requires special attention (e.g., a fraudulent transaction or abnormal training sample).

Definition 1 (Similarity self-join (SSJ) for vector dataset). *Given a vector dataset* $X = \{x_1, x_2, \cdots, x_N\}$, *where each* $x_n \in X$ ($1 \leq n \leq N$) *is a d-dimensional vector, and a threshold* $\epsilon > 0$, *find all vector pairs* $(x_n, x_{n'})$ *with* $\|x_n - x_{n'}\| \leq \epsilon$ *and* $1 \leq n \neq n' \leq N$.

In this paper (including Definition 1), we use the L-2 norm (Euclidean distance) as the similarity measure, but other similarity measures are also possible. The similarity-based self-join (SSJ) problem may be extended to cross-join if we consider similar vector pairs (x_n, y_m) from two vector datasets X and Y. We assume that the vector dimension d is high (e.g., > 100), which is the common case for embedding models. Note that solving the SSJ problem exactly (i.e., finding all qualifying vector pairs with $\|x_n - x_{n'}\| \leq \epsilon$) is prohibitively expensive. This is because traditional distance-based pruning techniques (e.g., using triangle inequality) are not effective in high dimensions [29], and thus the problem almost degrades to a linear scan over the dataset X to find all ϵ-neighbors of each vector x_n. As such, we solve SSJ *approximately* to trade accuracy for efficiency. We can gain efficiency by reducing the number of candidate vector pairs that are checked for ϵ-neighbors; this reduction may miss some vector pairs that should have been returned, thus hurting recall. We use the standard definition for recall: If $\mathcal{R}$ is the set of all vector pairs whose distance is less than ϵ, then the recall of an approximate result set $\mathcal{R}'$ of vector pairs is: $r = \frac{|\mathcal{R} \cap \mathcal{R}'|}{|\mathcal{R}|}$.

Existing literature (see Section 2 for a detailed discussion), treats SSJ as an *end-to-end task over a static dataset* executed exclusively by a dedicated server or system, with the goal of minimizing the overall execution time. For example, RSHJ [61] conducts approximate similarity join using LSH-based probing, where the entire join pipeline—from hashing to candidate generation to filtering—is optimized for overall performance. Similarly, distributed similarity join solutions like MAPSS [56], ClusterJoin [18], and C2Net [38] partition and process data in a way that tightly integrates filtering and verification stages, focusing on reducing total job runtime.

Our work is aligned with this line of research: we also treat SSJ as a unified, end-to-end task over a static dataset. Thus, our objective is to gain efficiency (measured by end-to-end execution time), while maintaining high recall. We do not need to separately consider precision, as incorrect candidate pairs with distances larger than ϵ will be simply removed by checking their distances. Therefore, an approximate solution to SSJ would always achieve perfect precision.

Scaling to large data sizes. In many applications, SSJ often involves large datasets. For example, a public multi-modal dataset can provide billions of images and their text descriptions [46], a video website such as YouTube can also host billions of videos [4], and an e-commerce platform can have hundreds of millions of users [30]. Moreover, each individual vector may also be large due to the high dimensionality d. As a result, vector datasets can take up TBs of storage and do not fit in the memory of a server. A way to scale up

SSJ is to use distributed methods that involve multiple machines. However, distributed solutions are expensive to deploy and usually suffer from costly inter-machine communication [18, 56].

SSD-based alternative and challenges. SSDs can offer a practical and cost-effective alternative to distributed SSJ: they are cheap (1TB currently costs ~$60), capacious (10+ TBs are common), and reasonably fast (reaching 1.4M random IOPS and 7 GB/s bandwidth). An alternative storage option is persistent memory (PM), which is byte-addressable, persistent, and offers SSD-like capacity at a lower cost than DRAM. However, PM is significantly more expensive ($12/GB) while delivering comparable read bandwidth (9 GB/s vs 7.4 GB/s) but lower write bandwidth (2.8 GB/s vs 6.9 GB/s). Although PM offers much lower latency (300ns vs 40μs), this advantage has minimal impact on end-to-end applications like similarity join. Thus, we study the case where the vector dataset is stored on SSD, while the memory is only a small fraction (e.g., $< 20\%$) of the data.

A baseline disk-based SSJ approach is to use disk-oriented vector indexes (e.g., DiskANN [48]), which target top-k nearest neighbor queries: we first build an index on the data and then use each vector as a query to search the index. We gradually increase k until the distances of the identified neighbors exceed ϵ. Figure 1 shows that this approach is slow and disk access dominates the overall time, due to the large volume of disk traffic. Our analysis suggests that the poor performance boils down to two reasons: *read amplification* and *repetitive access*. In particular, disk-oriented indexes access an individual vector each time but a vector is usually smaller than the 4KB disk page size (e.g., a 128-dimension float vector takes up 512B), which is the minimum granularity of disk read.[1] Thus, disk bandwidth is wasted on reading irrelevant data. Critically, this approach treats the vectors as independent queries and misses the opportunities to share and reuse the data fetched from disk. For example, vectors x_a and x_b may both need to check if x_c is their ϵ-neighbor. If x_b is processed right after x_a, x_c will reside in memory and can be reused, leading to smaller disk traffic and shorter runtime.

Our solution: DiskJoin. Motivated by this profiling analysis, we design a novel disk-based SSJ method, DiskJoin, that effectively reduces disk access time and, in turn, significantly improves the efficiency of SSJ processing. DiskJoin achieves this with two key designs: *access batching* and *task orchestration*.

Access batching addresses read amplification by reading sets of vectors (buckets), instead of one vector at a time. At a high level, DiskJoin organizes vectors in the dataset into buckets; each bucket is represented by a center vector and holds the vectors that are similar to the center vector. Using the bucket centers, DiskJoin then constructs a *bucket graph*, where each bucket is a node, and an edge connects two buckets if vectors in one bucket may have ϵ-neighbors in the other bucket. Our method evaluates SSJs by processing one bucket at a time, retrieving all neighbors of that bucket in the bucket graph. By reading data at bucket granularity, instead of individual vectors, DiskJoin largely eliminates read amplification, as each bucket contains multiple vectors and typically has size larger than the page size. It is also conducive to sharing disk access, as vectors in

---

[1]While some more recent embeddings have high dimensionality (e.g., 1536 for OpenAI's text embedding), dimensionality reduction techniques like the Matryoshka Representation Learning (MRL) [37] and vector quantization methods [2, 23] can be used to compress the vector size to well below a page size without loss of accuracy.

the same bucket are similar, and thus are likely to have ϵ-neighbors in common. We make further system contributions through careful implementation to ensure that memory consumption stays below budget when assigning the vectors to buckets, and that construction of the bucket graph is fast and incurs limited overhead.

Task orchestration further reduces disk access through more effective sharing of disk access and better-informed cache eviction policy. It achieves this by choosing a processing order for buckets, to leverage that buckets processed consecutively can share the buckets loaded to memory (i.e., temporal locality). Identifying the optimal bucket ordering is NP-hard (Section 4.1). However, we leverage a popular algorithm for classical graph ordering (which assigns IDs to graph nodes, such that nodes with adjacent IDs share common neighbors), as an approximation heuristic, and process buckets in the order of their new IDs. We note that for a given bucket processing order, the bucket graph provides complete knowledge of the required bucket accesses. We choose Belady's algorithm [6] to manage the memory cache eviction as it is proven to maximize the cache hit rate given perfect knowledge of future data access.

Similarity join vs vector similarity search. We note that similarity self-join has different requirements from vector similarity search (VSS) [21, 31, 40]. VSS is an *online task* with stringent latency requirements for each query (e.g., 10ms). VSS cannot collect many queries within the latency budget period, and thus the opportunities for the joint optimizations of multiple queries are limited. In contrast, SSJ is an *offline batch processing task* that focuses on end-to-end time. As such, DiskJoin batches the processing of vectors that check the same ϵ-neighbors and obtains the data access pattern beforehand for orchestration. These designs make DiskJoin fundamentally different from disk-based indices for VSS [48, 55].

Some recent research explores *filtered* vector similarity search by integrating attribute filtering into the index. For instance, Filtered-DiskANN [25] builds a graph whose edges respect both geometric proximity and shared attribute labels. SeRF [66] overlays range-specific subgraphs to support filtering over value ranges. NHQ [54] constructs a composite graph index to enable joint pruning on vector and attribute similarity. We are, however, not aware of any research on filtered vector similarity join. While our work focuses on the standard setting of vector SJ—without attribute filtering—we briefly discuss how our approach can be extended to support attribute filtering in Section 3.

Summary of evaluation and results. We evaluate DiskJoin on three real-world datasets that contain up to a billion vectors and limit the memory size to a small portion of the dataset size (e.g., 10%). The results show that compared with searching a disk-oriented vector index, DiskJoin is 2–3 orders of magnitude faster. As suggested by the results in Figure 1, this is because DiskJoin significantly reduces disk traffic and hence disk access time. Moreover, Figure 1 shows that disk access no longer dominates the running time for DiskJoin. Our experiment suggests that the designs of DiskJoin are effective at improving cache hit rate and reducing disk access. Specifically, Belady's algorithm and bucket ordering improve the cache hit rate by about 20% and 50%, respectively.

Contributions and outline. We make the following contributions.

- We organize a review of the related work on similarity joins, highlighting the existing bottlenecks of huge disk traffic and long disk access time. [Section 2]
- We provide the intuition and overview for our novel DiskJoin method for SSJ. DiskJoin reduces disk access drastically, through clever implementation of bucket-wise processing of SSJs. Through the key contributions of access batching and task orchestration, DiskJoin allows for more sharing of disk data access and richer information for more effective cache management. [Section 3]
- We model the problem of task orchestration as an edge-covering problem of the bucket dependency graph and show theoretical results on its hardness. We proceed to provide practical heuristics that leverage prior work on cache management and graph ordering, in a careful, memory-conscious implementation that performs task orchestration efficiently. [Section 4]
- We detail the optimizations that support the effectiveness of our bucketization model, allowing for clustering the vectors with limited memory and pruning unnecessary vector similarity computations. [Section 5]
- We experimentally evaluate our methods over three real-world datasets of up to 1.4B vectors, and show that DiskJoin offers clear, consistent, and up to 3-orders-of-magnitude improvement over the prior state of the art. [Section 6]

2 Related Work

Similarity join (both self-join and cross-join) is a classical problem that has been extensively studied for data types including strings [12, 43, 44, 53, 60], sets [3, 8, 12, 53], and vectors [7, 9, 15, 20, 22, 33, 38, 57]. The general solution is a two-step *filter-verify* procedure, where the *filter* step generates a set of candidate pairs, and the *verify* step checks the candidate pairs via similarity computation. While the *verify* step is straightforward, the *filter* step is crucial for accuracy and efficiency by excluding dissimilar pairs and keeping similar pairs. Many filtering techniques have been proposed, including those targeting generic metric spaces [9, 13, 15, 18, 20, 56] as well as techniques tailored to specific data types [3, 22, 60].

Similarity join for strings and sets. Prefix filtering is a dominant technique for set and string similarity joins. Consider sorted sets r and s, and denote their length-$(|r| - t + 1)$ prefixes as $P(r, |r| - t + 1)$ and $P(s, |r| - t + 1)$; if r and s have more than t common elements, it holds that $P(r, |r| - t + 1) \cap P(s, |r| - t + 1) \neq \emptyset$. Using this property, SSJoin [12] generates candidate pairs by considering objects whose prefixes have common elements, and can handle both sets and strings. AdaptJoin [53] improves prefix filtering by selecting the first $|r| - t + l$ elements as the prefix and filtering out object pairs whose prefixes share fewer than l common elements; l is selected to balance the costs of filtering and verification. GPJoin [8] groups identical prefixes for sets and executes filtering in batches. MGJoin [44] reduces the number of candidates by using multiple prefixes with different lengths for each string during filtering.

Similarity join for low-dimensional vectors. Space-partitioning indices can prune dissimilar vectors in low dimensions (e.g., $d < 10$). For example, tree-based variants, such as R-tree [9], M-tree [15] and eD-tree [20], partition the space hierarchically. Search for ϵ-neighbors recurses from the root and uses the triangle inequality to prune tree branches that are unlikely to contain qualified neighbors.

Some algorithms use the ϵ-grid (e.g., EGO [7], EGO-star [33], SuperEGO [32]), which splits the space into ϵ-sized grids and sorts the vectors by the lexicographical order of their grid coordinates. Then, the vectors are grouped into parts, and two parts are checked if their ϵ-margin bounding boxes intersect. Space-partitioning indices do not work for high-dimensional vectors (e.g., $d > 100$) produced by machine learning models, as the pruning power of the triangle inequality and bounding boxes diminishes in high dimensions [29].

Similarity join for high-dimensional vectors. RSHJ [61] conducts similarity cross-join for two vector datasets X and Y and uses locality-sensitive hashing (LSH) [28], a popular technique for vector similarity search in high dimensions. It assumes that X and Y reside in memory and produces an approximation (i.e., identifies most, instead of all similar pairs) like we do. LSH uses tailored hash functions to map vectors to buckets ensuring that similar vectors are more likely to be mapped to the same bucket. RSHJ builds hash tables for both datasets and generates candidate pairs by querying the hash table of Y using vectors in X. To reduce lookup cost, it only uses a representative subset of X for lookups. However, state-of-the-art vector indices, such as proximity graphs [40, 41, 48] and inverted file (IVF) [31], perform much better than LSH for similarity search.

Distributed similarity join. Some prior work scales up similarity join with distributed frameworks like MapReduce [19]. MAPSS [56] first selects a set of centroids and then partitions the dataset according to the centroids. The mappers use the triangle inequality to filter the candidate objects for each partition, and each reducer verifies the candidate pairs for a partition. ClusterJoin [18] improves MAPSS by pruning the candidate pairs with a bisector-based filter in general metric spaces, as well as a set of filters specific to different distance functions. It achieves load balancing by re-partitioning the datasets over the machines. Chen et al. [13] further enhance candidate filtering using pivot filtering and plane sweeping techniques and propose two partitioning methods based on space-filling curve and KD-tree to avoid re-partitioning. More recently, C2Net [38] applies an LSH variant, C2LSH [22], to conduct similarity-based vector cross-join by hashing both datasets to buckets. The mappers enumerate the vector pairs mapped to the same bucket, and the reducers aggregate the pairs to calculate their collision counts. A vector pair is regarded as a candidate if their collision count exceeds a threshold. These distributed solutions generally suffer from a high inter-machine communication cost because similar object pairs need to be sent to the same machine for verification.

To the best of our knowledge, DiskJoin is the first to conduct similarity join for high-dimensional vectors on disk. Compared with existing vector similarity join work (e.g., EGO, RSHJ, C2Net) that use tree, grid, or LSH, DiskJoin leverages state-of-the-art vector indices (i.e., proximity graph and IVF), which are more efficient for high-dimensional vectors. Compared with distributed solutions, DiskJoin is easier to deploy (as it does not require a cluster) and does not suffer from high inter-machine communication cost. DiskJoin also follows the general *filter-verify* procedure for similarity join but the main challenge is to reduce the disk access cost, which has not been explored by existing work.

Disk-based Similarity Search. Prior work has explored disk-based vector similarity search systems to support billion-scale vector retrieval with low memory footprints and high throughput. GRIP [63]

pioneered memory-SSD hybrid similarity search by introducing a multi-store IVF-based framework, where the in-memory index quickly routes queries to nearby clusters and completes validation by accessing the SSD store. SPANN [14] adopts similar design but optimizes the IVF index construction to better suit the disk scenario. DiskANN [48] proposes a graph-based approach, using compressed vectors for in-memory graph traversal and refining results with full-precision vectors stored on SSDs. Other systems, such as SmartANNS [50] and ES4D [36], also explore the opportunity for near-data processing, which offloads the query processing to SSDs equipped with computational capabilities.

Batch Processing. DiskJoin leverages batch processing, which is a widely-used concept in database research. SharedDB [24] first introduced the design of batching queries in RDBMSs to share data access and computation. This concept was later extended to vector similarity search in [45]. Query batching is especially common in GPU-based similarity search systems—such as GGNN [26], SONG [65], and BANG [34]—to fully exploit the massive parallelism of GPUs. Other systems, including FreshDiskANN [47] and IP-DiskANN [62], also employ batch processing for index updates to amortize operational overhead.

3 DiskJoin Overview

In this section, we provide an overview of DiskJoin's workflow, depicted in Figure 2. DiskJoin takes as input the vector dataset X, the memory budget C, the distance threshold ϵ for similar vector pairs, and a target recall λ. Under the assumption that X resides on local disk, DiskJoin will identify and write a set of similar vector pairs R' to disk, such that the expected recall of R' is λ and memory usage is bounded by C. The core of DiskJoin is the bucket-wise processing of SSJ. It first groups X into buckets, such that each bucket contains similar vectors. Then, it identifies candidate bucket pairs that may contain neighboring vectors. Finally, for each candidate bucket pair, it performs pairwise vector comparisons across the buckets to produce the final results. For this approach to be effective, the algorithm needs to be smart about the bucketization, the identification of likely dependencies across buckets, and the processing order of buckets. This provides three benefits: (1) it avoids read amplification, as the vectors are read from disk at bucket granularity; (2) data access is shared among the vectors in the same bucket because they are similar and hence are likely to have high overlap in their ϵ-neighbors; (3) understanding the dependencies across buckets (i.e., which buckets are likely to contain ϵ-neighbors) allows for advanced memory cache management. We provide here an overview of these key components in DiskJoin, and describe them in detail in subsequent sections.

Vector bucketization. DiskJoin partitions X into buckets, with each bucket b represented by a center vector c_b and containing similar vectors. Traditional clustering methods, such as k-means, are not suitable for this task, due to high memory requirements—often, they need the entire dataset to fit in memory, but, in practical settings, the memory size C is much smaller than the dataset size. DiskJoin implements an efficient bucketization procedure (detailed in Section 5.1). At a high level, it starts with a random selection of vectors to serve as the bucket centers, and then assigns each vector
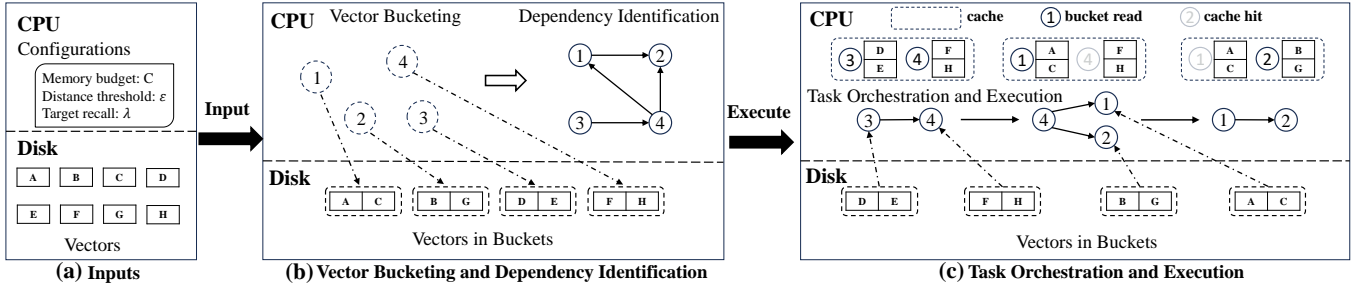
Figure 2: Workflow of DiskJoin. Numbers (e.g., 1 and 2) indicate buckets, and letters (e.g., A and B) indicate vectors. (a) The inputs are the vector dataset and task configurations; (b) the vectors are grouped into buckets, and a bucket graph is constructed, where an edges means that the vectors in one bucket needs to check another bucket for neighbors; (c) task orchestration decides a good processing order for the edges in the bucket graph to reduce cache miss, where the cache can hold 2 buckets.

to the nearest center. Each bucket also records a radius r_b, which is the maximum distance between its vectors and the center.

DiskJoin also adjusts the disk layout of the dataset by storing the vectors of each bucket consecutively (Figure 2b). This allows for fetching each bucket via sequential disk reads. A naive implementation of reorganizing the disk layout that uses random reads to retrieve the vectors for each bucket and then writes them back to disk would cause read amplification. We instead implement a more efficient, sequential-disk-access method that does not violate the memory budget (Section 5.1).

Dependency identification. DiskJoin constructs a *bucket graph* that indicates, for each bucket b, which other buckets contain vectors that need to be checked for potential ϵ-neighbors. In particular, after vector bucketization, we have the center and radius of the buckets in memory (i.e., c_b and r_b for bucket b). Using the triangle inequality, bucket b_j may contain ϵ-neighbors of bucket b_i if:

$$\|c_{b_i} - c_{b_j}\| - r_{b_i} - r_{b_j} \leq \epsilon. \tag{1}$$

Since the triangle inequality has poor pruning power in high-dimension spaces, each bucket will have many candidate neighboring buckets. Inspired by [64], we adopt the probabilistic bucket pruning rule (Section 5.2), which meets the target recall λ while pruning many more candidate buckets. We denote the bucket graph as $G(V, E)$, where each bucket $b \in V$ is a node, and a directed edge $e_{ij} \in E$ if bucket b_i and b_j are a candidate bucket pair and $i < j$. We do not record edges with $i > j$ because Euclidean distance is symmetric, i.e., by computing the distances of the vectors in b_i to the vectors in b_j, we also obtain the distances from b_j to b_i.

Task orchestration. The edges of the bucket graph identify all pairs of buckets that need to be processed together to identify ϵ-neighbors. A *task* refers to the processing of an edge (i.e., bucket pair). SSJ completes when all edges are processed. The order of processing the edges is critical for cache hit rate, which in turn determines IO cost and impacts efficiency, as a bucket needs to be loaded from disk upon cache miss. For example, if edge e_{jh} is processed after e_{ij}, it will not incur a cache miss for bucket b_j, as b_j will already reside in memory. We define the problem of identifying the bucket load sequence to process all edges with minimum cache misses and show that it is NP-hard (Section 4.1). We then propose a heuristic approximation that decomposes it into two sub-problems: optimal cache management and task ordering. For a given task order, the bucket graph provides complete knowledge of future data

access; thus, we use Belady's algorithm [6] for cache management (Section 4.2), which has been proven to yield the minimum total cache misses when data access is known. To solve task ordering, we leverage *graph reordering* methods. The intuition is that buckets that have a large overlap in their neighboring buckets should be processed consecutively for temporal cache locality (Section 4.3).

Task execution. Task orchestration generates the processing order of the edges and the buckets to load to and evict from the cache memory at every step. During execution, our method evaluates the distances of vector pairs from two buckets each time via in-memory computation to check the ϵ-neighbors for the vectors in each bucket, and follows the task orchestration to load and evict buckets. The resulting ϵ-similar vector pairs are written to disk.

Extending to cross-join. DiskJoin can be extended to process the cross-join for two vector datasets with the following changes. (1) Vector bucketing is conducted separately for the two datasets, generating their own buckets. (2) The bucket graph becomes a bipartite graph, with the buckets of one dataset requiring the buckets of the other dataset. (3) During task orchestration, we use graph reordering to decide the processing order of the buckets from the larger dataset, the memory cache to hold buckets from the smaller dataset, and Belady's algorithm to manage cache eviction. The other way, i.e., reordering the smaller dataset and caching the larger dataset, is less efficient. This is because the reordered dataset only needs to be streamed to the memory once while the cached dataset will experience bucket loading and eviction. Our experiments show that the disk traffic can be several times the dataset size.

Extending to attribute filtering. DiskJoin can be extended to support attribute filtering with a simple modification: before performing pairwise vector comparisons between candidate bucket pairs, apply the attribute filter to each vector to generate a bitmap, and skip similarity computations for vectors that do not pass the filter. If the attribute filter satisfies the range-preserving property—that is, the filtered results form a contiguous block in the lexicographical order of attribute values—further optimization is possible. Specifically, during vector bucketization, vectors within each bucket can be sorted by the attributes' lexicographical order. As a result, during task execution, only the vectors that satisfy the filter need to be loaded from disk, using a single sequential read.

4 Task Orchestration

In this section, we describe how DiskJoin determines the processing order of bucket pairs and how it handles cache management during execution. At this stage, we assume a known bucket graph; we describe the bucketization and graph construction in Section 5. We start by defining the problem of covering all graph edges with minimum cache misses, and show that it is NP-hard. We then propose an approximation heuristic that decomposes it into the sub-problems of cache management (Section 4.2) and task ordering (Section 4.3).

4.1 Edge Covering with Cache Constraints

We assume a bucket graph $G = (V, E)$, where each node $v \in V$ is a bucket, and an edge $(u, v) \in E$ indicates that buckets u and v may contain vectors that are ϵ-close. To compute the distance of vectors in the bucket pair (u, v), both u and v have to be in memory. We assume that all buckets are of the same size, and, with slight abuse of notation, we use C to denote the number of buckets that fit in memory. To perform SSJ, we need to process all edges of the bucket graph, and the goal is to minimize the disk access cost, which corresponds to minimizing the number of bucket load operations. We define the *minimum edge cover with cache* (MECC) problem as follows.

Definition 2 (MECC: Minimum Edge Cover with Cache). *Assume a given undirected graph $G = (V, E)$ and an initially-empty cache with capacity C. The cache supports two operations: load(v), which loads node v to the cache, and evict(v), which evicts node v from the cache following a given policy P when the cache is full and needs to load a node. An edge of $(u, v) \in G$ is said to be covered if both u and v are present in the cache at the same time. The Minimum Edge Cover with Cache problem seeks to find the sequence of load operations with the minimum length that covers all edges of G.*

We note that edge direction in bucket graphs simply implies that each pair of buckets needs to be processed once, but directionality is not relevant to the covering problem. Thus, we consider G to be undirected in this discussion. Without loss of generality, we assume that G is connected—disconnected components can simply be handled independently of each other. The length of the node load sequence corresponds to the number of buckets loaded from disk to memory, and thus MECC minimizes the disk access cost. The minimum possible length of the load sequence is $|V|$, as each node needs to be loaded at least once. Figure 3 shows an example of MECC, where the graph has 4 nodes, the cache capacity is 2, and the length of the minimum load sequence is 7. The MECC in Figure 3 is easy to solve because two subsequently-checked edges share at most 1 bucket. However, MECC becomes difficult in general cases.

Theorem 1. *The MECC problem is NP-hard.*

PROOF. We show that the decision version of MECC can be reduced from independent set (IS), which, given a graph $G = (V, E)$ and an integer T, asks whether there exists a set $V' \subseteq V$ such that $|V'| = T$ and $\nexists u, v \in V'$ such that $(u, v) \in E$. Given an instance of IS with connected graph $G(V, E)$ and integer T, we construct an instance of MECC with the same graph $G(V, E)$ and a cache of size $C = |V| - T + 1$ with last-in-first-out (LIFO) as the eviction policy P.

We first show that if G has an independent set of size T, the solution of MECC is a load sequence of length $|V|$. Let $V_T \subseteq V$ be
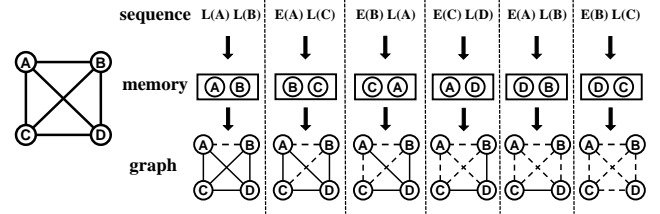
Figure 3: An example of the minimum edge covering with cache (MECC) problem, the graph is in the left plot, the cache size is 2, and $L(\cdot)$ and $E(\cdot)$ mean load and evict a node, respectively. The dotted lines show the edges covered in each step.
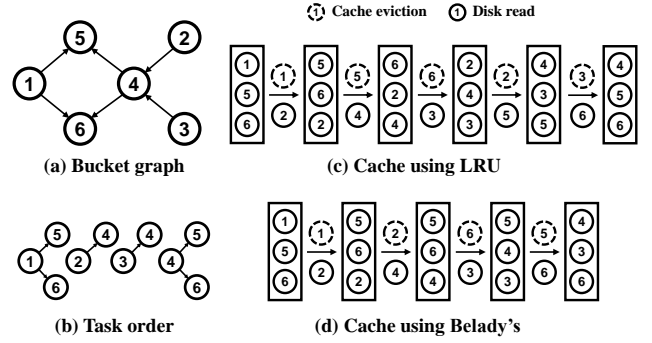
Figure 4: An example for Belady's algorithm. The edge processing order is in (b), the cache size is 3, LRU loads 8 buckets while Belady's algorithm loads 7 buckets.

an independent set of size T in G. We create a load sequence of length $|V|$ that loads the nodes $V \backslash V_T$ first into the cache, followed by the nodes in V_T. When executing this load sequence, nodes $V \backslash V_T$ will take up $|V| - T$ slots in the cache, leaving one free slot. Each subsequent node in V_T, is loaded into that slot, and, after being processed, gets evicted (based on LIFO) to load the next node. Since the nodes in V_T form an independent set, there are no edges between them. All edges in E are either among nodes in $V \backslash V_T$, which are together in the cache, and thus are covered, or between a node in $u \in V_T$ and nodes in $V \backslash V_T$, which are covered when u is loaded. Thus, this sequence covered all edges in G.

Conversely, we also show that if the MECC instance can be solved with exactly $|V|$ node loads (so every node is loaded once) and a cache size of $|V| - T + 1$ using LIFO, the set of evicted nodes form an independent set of size T. This is because any two evicted nodes are never present in the cache at the same time, and if any edges existed between them they would not have been covered.

Since IS is NP-hard, and IS reduces to MECC in polynomial time, it follows that MECC is also NP-hard. □

We note that this hardness result is specific to self-joins. Cross-joins are likely simpler, since their bucket graph is bipartite. Self-joins are generally harder, and our algorithms can also be used in the evaluation of cross-joins, as discussed in Section 3. However, it is possible that cross-joins offer additional opportunities for improvements; we leave this direction to future work.

Algorithm 1 Cache Management via Belady's Algorithm

Input: Bucket read sequence S, bucket number M, cache size C
1: Bucket queue $Q \leftarrow$ Max_Heap(C)
2: Bucket position $P \leftarrow$ Array(M, List())
3: Bucket access count $c \leftarrow$ Array(M, 0)
4: **for** $i \in \{1 \cdots |S|\}$ **do**   ▷ *Store access time for each bucket*
5: ⌊ $P[S[i]]$.append(i)
6: **for** $i \in \{1 \cdots |S|\}$ **do**   ▷ *Enumerate bucket access sequence*
7: ⌊ $b \leftarrow S[i]$, $c[b] \leftarrow c[b] + 1$
8: ⌊ **if** Cache.find(b) = true **then**   ▷ *Cache hit*
9: ⌊ ⌊ Q.update(b, $P[b][c[b]]$)
10: ⌊ **else**   ▷ *Cache miss*
11: ⌊ ⌊ **if** Q.size = C **then**
12: ⌊ ⌊ ⌊ $b' \leftarrow Q$.pop(), Cache.evict(b')
13: ⌊ ⌊ Cache.load(b), Q.push(b, $P[b][c[b]]$)

As MECC is NP-hard, we solve it approximately by decomposing it into two sub-problems.

- *Given a processing order of the edges, we examine the cache eviction policy that minimizes the number of bucket loads.* We show that this can be solved optimally using Belady's algorithm for cache management in Section 4.2.

- *We examine how to derive a good processing order for the edges to achieve a small number of bucket loads.* We solve this heuristically, using a lightweight graph reordering algorithm in Section 4.3.

4.2 Cache Management via Belady's Algorithm

Assume that the processing order for the edges in $G(V, E)$ has been decided as $\{e_{uv}, \cdots, e_{u'v'}\}$, we can deduce the access sequence of the buckets as $S = \{u, v, \cdots, u', v'\}$ by extracting the buckets connected by each edge. S has a length of $2|E|$ and specifies when each bucket will be used for computation. The cache can keep C buckets in memory. Given the bucket access sequence, our goal is to minimize the number of bucket loads with cache management.

The cache load decision is straightforward, i.e., when checking edge e_{uv} and a required bucket (e.g., u) is not in the cache, the bucket should be fetched from disk. However, when u needs to be loaded and the cache is full, we need to choose which bucket to evict from the cache. There are popular strategies such as first-in-first-out (FIFO), least-frequently-used (LFU), and least-recently-used (LRU), with the latter often being the choice method in many practical settings. However, these policies make eviction decisions based on *prior* data access. What is special in our setting is that the bucket access sequence S specifies all data access, including the future. Thus, we use Belady's algorithm for cache management, which can leverage this information to minimize the number of cache misses [6].

Belady's algorithm looks into future bucket access and evicts the bucket in the cache that will be accessed again at the latest point in the future. This minimizes the number of cache misses because, compared with the bucket chosen for eviction, the other buckets will be accessed sooner; thus, evicting any of these buckets would yield at least the same number of cache misses as evicting the chosen bucket. Figure 4 compares LRU with Belady's algorithm with an example. LRU (c) evicts bucket b_5, which needs to be loaded again later to process another edge. In contrast, Belady's algorithm (d) evicts bucket b_2 because it will never be used again, and this keeps b_5 in the cache and saves one future bucket load.

Algorithm 1 shows how DiskJoin implements Belady's algorithm. We use a max-heap Q to manage the cache-resident buckets, and each element of the heap is a key-value pair, where the key is the bucket id and the value is the index of the bucket's *subsequent* access in the bucket access sequence. Q can hold C elements, with C being the capacity of the cache. DiskJoin also uses a two-dimensional array P to record when each bucket will be accessed, and each row, $P[b]$, records the indices in the bucket access sequence for bucket b. A counter array c records how many times each bucket has been accessed until now. Algorithm 1 fills in the index array P by enumerating the bucket access sequence S (lines 4–5). Then, the algorithm handles access one bucket at a time (lines 6–13). If the required bucket b is in the cache (i.e., Cache.find(b) = true), we simply update the index of the bucket in the max-heap Q (line 9). If the bucket is not in the cache and the cache has free slots, we load the bucket and add its entry to Q (line 13). If the cache is full, we choose the bucket b' with the largest next access index and remove it from both Q and the cache (line 12). Algorithm 1 scans S over two passes, and the worst-case complexity is $O(|E| \log C)$, since $|S| = 2|E|$ and the size of the heap Q is C.

4.3 Task Ordering via Graph Reordering

Now we revisit the issue of determining the processing order of the edges in $G(V, E)$. Given a bucket node $v \in V$, let $\mathcal{N}(v)$ be the set of v's out-neighbors, i.e., $\mathcal{N}(v) = \{u : (v, u) \in E\}$. There are $|\mathcal{N}(v)|$ edges between v and $\mathcal{N}(v)$, and they all share bucket v. The first intuition applied in our heuristic is to process all of v's outgoing edges in succession. This ensures that at least one bucket for each task will not incur a cache miss. When the cache size is $C \geq 2$, this reduces the worst-case cache miss count from $2|E|$ to $|V| + |E|$. As $|E|$ is usually much larger than $|V|$, the reduction is close to 50%.

Now, task ordering boils down to determining the processing order of the nodes in G. While following the order of the node IDs presents a straightforward solution, it may still incur a high number of cache misses. For instance, Figure 5(b) shows that following the ID order needs to load 8 nodes to the cache. In comparison, Figure 5(c) shows that by processing node 4 before nodes 2 and 3, the number of loaded nodes can be reduced to 5. This is because nodes 1 and 4 share common neighbors $\{5, 6\}$, and processing node 4 immediately after node 1 can reuse $\{5, 6\}$ in the cache.

The example in Figure 5 suggests that we should process nodes u and v subsequently if the overlap between their neighbors, $\mathcal{N}(u)$ and $\mathcal{N}(v)$, is high. This allows for the reuse of nodes already loaded to the cache. More generally, denote the average out-degree of the nodes in G as d_{avg}, a cache with capacity C can hold the neighbors of C/d_{avg} nodes. We should decide an ordering for the nodes in G such that the nodes in a sliding window of size $w = C/d_{avg}$ can share the most neighbors. Thus, we aim to maximize the following score.

$$F(P) = \sum_{0 < P(v) - P(u) \leq w} |\mathcal{N}(u) \cap \mathcal{N}(v)| = \sum_{i=1}^{n} \sum_{j=\max\{1, i-w\}}^{i-1} |\mathcal{N}(u) \cap \mathcal{N}(v)|, \quad (2)$$

where n is the number of nodes and $P : [1, n] \rightarrow [1, n]$ is an one-to-one function that maps a node from its new id to original id. We will process the nodes following their new ids.

The formulation above resembles Gorder [57], a classical graph reordering algorithm that aims to improve the cache locality of
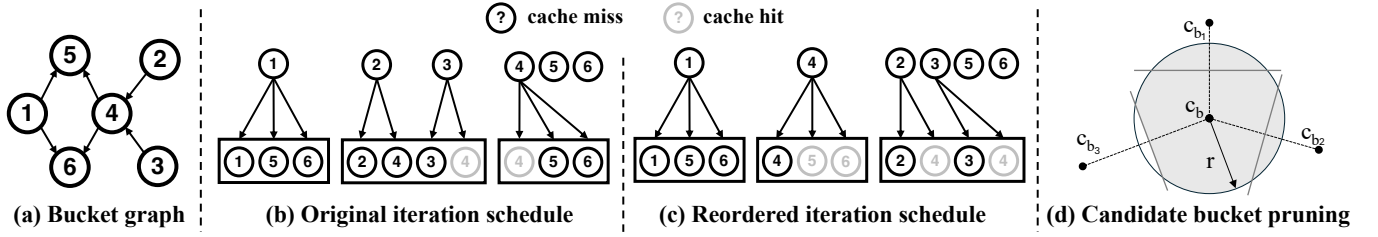
Figure 5: Given the bucket graph (a), the original task ordering (b) results in 8 cache misses in a cache that holds 3 nodes (rectangle box). The reordered schedule (c) reduces the cache misses to 6. In the illustration of candidate bucket pruning (d), bucket b_3 is pruned, and as a result, the neighbors in the white arc are missed.

Algorithm 2 Task Ordering with Graph Reordering

Input: Bucket graph $G(V, E)$, sliding window size $w = \frac{C}{d_{avg}}$
Output: An ordering P for the nodes in G
1: select a node v as the start node, $P[1] \leftarrow v$
2: $V_R \leftarrow V(G) \setminus \{v\}, i \leftarrow 2$
3: **while** $i \leq n$ **do**   ▷ *Greedily insert remaining nodes to P*
4:     $v_{max} \leftarrow \max_{v \in V_R} \sum_{j=\max\{1,i-w\}}^{i-1} \left| \mathcal{N}(P[j]) \cap \mathcal{N}(v) \right|$
5:     $P[i] \leftarrow v_{max}, i \leftarrow i + 1$
6:     $V_R \leftarrow V_R \setminus \{v_{max}\}$
7: **return** P

---

in-memory graph processing. We use the greedy heuristic in Algorithm 2 to conduct the graph reordering. In particular, to initialize, we select the node with the largest number of out-neighbors as the first node (line 1). Then, in each step, we select the nodes from the remaining node set V_R that has the maximum overlap in its neighbors with the previous w nodes in the sliding window (lines 2–6). Calculating the overlap and selecting the maximum for every iteration has a prohibitive time complexity: $O(w \cdot d_{max} \cdot n^2)$ where d_{max} is the maximum out-degree of G. To resolve this, our implementation adopts a priority queue to keep track of the following score k_v for each node $v \in V$

$$ k_v = \sum_{j=\max\{1, i-w\}}^{i-1} \left| \mathcal{N}(P[j]) \cap \mathcal{N}(v) \right| $$

which is the number of overlapping neighbors between v and nodes in the current sliding window. We incrementally update only the k_v of nodes v that are influenced by v_o and v_i—the nodes going out of and into the window, respectively, while the window is sliding. The algorithm's time complexity is $O(\sum_{u \in V} (d^+(u)^2))$, where $d^+(u)$ is the out-degree of u. Our evaluation shows that our optimized implementation incurs little overhead, as $d^+(u)$ is usually not large.

Putting the two components together. Task orchestration first runs graph reordering to determine the processing order of the nodes, which induces a processing order of the edges. During task execution, we follow the order to process the edges and use Algorithm 1 for cache management.

5 Bucketing Optimizations

The core of DiskJoin relies on bucket-wise processing of SSJs. In this section, we discuss how the partitioning of the vector dataset into buckets can be done effectively and efficiently, and detail our pruning strategy when building the bucket graph.

5.1 Efficient Vector Bucketization

During vector bucketization, DiskJoin groups the vectors of the dataset $\mathcal{X}$ into buckets that contain similar vectors. Standard clustering approaches cannot be straightforwardly adapted for our setting. For example, K-means would require a full disk dataset scan per iteration for assigning points to clusters and updating the centroids. This overhead across multiple iterations is impractical in our memory-constrained setting. Here, we propose an efficient 3-step vector bucketization process that respects strict memory constraints (e.g., 5% of the dataset size).

First, we select a set $\mathcal{X}'$ of randomly-sampled vectors as the bucket centers. We first generate the ids of the vectors to sample and then stream the entire dataset over memory to collect the sampled vectors. If the vectors in $\mathcal{X}$ are already permuted, we may simply use the first $|\mathcal{X}'|$ vectors to save disk traffic. The number of centers and, hence, buckets is configured to be large (e.g., around 1‰ of the dataset size); this leads to small buckets and fine-grained space partitioning, which reduces the number of vectors each bucket needs to check for ϵ-neighbors and, hence, disk traffic. A random sample is sufficiently representative of the dataset because the sample size is large (e.g., a 1‰ sample of a billion-scale dataset still contains 1M vectors). This step safely meets the memory constraint since it only requires streaming the dataset and preserving 1‰ of it in memory.

Then, we read a block of vectors from the disk each time to avoid read amplification, and for each vector x in the block, we search for its nearest center c in $\mathcal{X}'$ for bucket assignment. A brute-force scan over $\mathcal{X}'$ for this purpose is expensive as $\mathcal{X}'$ is also a large vector dataset. As such, we construct an HNSW [40] on $\mathcal{X}'$ in memory, which is one state-of-the-art index for vector similarity search (VSS), and use x as a query to search the HNSW to identify its nearest center c. In particular, HNSW is a proximity graph index, where vectors are the graph nodes and edges connect similar vectors. A VSS query is processed by a traversal on the proximity graph that moves toward the neighbors with small distances to the query. HNSW can be built with complexity $O(|\mathcal{X}'| \log |\mathcal{X}'|)$, and process a VSS query with $O(\log |\mathcal{X}'|)$, which is much faster than linear scan. The memory footprint of the HNSW index—including the vectors in $\mathcal{X}'$ and the adjacency lists—is only about 2‰ of the dataset size. Therefore, it comfortably fits within reasonable memory constraints.

We do not write x immediately to b_c (i.e., the bucket corresponding to center c) as each vector is usually smaller than the 4KB disk page size and this causes write amplification. Instead, we keep an in-memory buffer for each bucket, append its vectors to the buffer, and flush the buffer to disk when it is full.

Algorithm 3 Probabilistic Pruning for Candidate Buckets

Input: Target recall λ, candidate bucket list L for bucket b
Output: Pruned candidate bucket list L'

1: $L' \leftarrow L$, sum $\leftarrow 0$
2: **for** $b_i \in L$ in descending order of distance to b **do**
3:      sum $\leftarrow$ sum $+ \arccos(\min\{x_i, 1\})$
4:      **if** sum $\geq 1 - \lambda$ **then**   ▷ *Stop when error exceeds* $1 - \lambda$
5:          break
6:      $L' \leftarrow L' \backslash \{b_i\}$
7: **return** L'

Our vector bucketing procedure is time-efficient because it only involves sequential disk accesses and requires minimal number of disk dataset scans—one for centroids sampling, one for bucket assignment and one for writing buckets back to disk. It is also memory-efficient since the minimum memory requirement is only 2‰ of the dataset size, which can fit within any reasonable memory constraint. When it finishes, the vectors of $\mathcal{X}$ are stored as buckets on disk. We note that the disk organization and the HNSW index on bucket centers are independent of the distance threshold ϵ and recall target λ. They can be built once and reused across queries with varying distance thresholds and recall requirements. The HNSW index for the centers is also used to build the dependency graph $G(V, E)$ for the buckets. In particular, for each bucket b with center c, we use c as a query to search the HNSW and identify a set $\mathcal{B}_c = \{b_1, b_2, \cdots, b_L\}$ of nearest centers (i.e., buckets). Then, we use a probabilistic pruning rule (discussed next) to determine the buckets that should be checked by b for ϵ-neighbors.

5.2 Probabilistic Pruning for Candidate Buckets

Using the triangle inequality in Eq. (1) to decide the candidate buckets for each bucket b will produce many candidate buckets. This will result in high disk access and computation costs as each bucket reads and checks many buckets. Inspired by [64], we use a probabilistic candidate bucket pruning method to reduce the number of candidate buckets while meeting the recall target λ.

Figure 5(d) illustrates the idea with a toy example. We are considering bucket b and have identified all the candidate buckets for verification, i.e., $\{b_1, b_2, b_3\}$, whose centers are plotted as black points. We also plot a ball with center c_b and radius r, which represents the ϵ-neighborhood of bucket b. For each candidate bucket (e.g., b_1), we plot its boundary region with the ϵ-neighborhood of bucket b, which is a mid-vertical hyperplane as we assign each vector to its nearest center. We prune the candidate buckets in descending order of their distances to c_b. As shown in the example, if bucket b_3 is pruned, any ϵ-neighbors in the white arc will be missed. We assume that the vectors are uniformly distributed in the hypersphere $B(c_b, r)$, and estimate the percentage of missed neighbors based on relative volume. We can keep pruning candidate buckets until the estimate of missed neighbors reaches $1 - \lambda$.

More generally, assume that a bucket b has l candidate buckets, and we prune the j furthest buckets from b. The percentage of missed neighbors can be expressed as

$$\beta(j) = \frac{V(\bigcup_{i=l-j}^{l} A(i))}{V(B(c_b, r))} \leq \frac{\sum_{i=l-j}^{l} V(A(i))}{V(B(c_b, r))} \quad (3)$$

Name	Vectors	Dimensions	Size (GB)
Deep100M	100,000,000	96	36
BigANN100M	100,000,000	128	48
SPACEV1B	1,402,202,072	100	523

Figure 6: Summary statistics of the experiment datasets
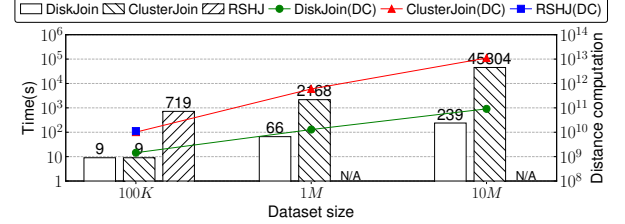
Figure 7: Despite being a disk-based method, DiskJoin is order of magnitudes faster than Clusterjoin and RSHJ—the latter failing to execute in larger data sizes. This is because DiskJoin performs orders-of-magnitude fewer distance computations.

where $V(\cdot)$ is the volume of a high-dimension shape, $A(i)$ is the arc between bucket b and bucket b_i. It has been shown by [64] that $\beta(j)$ can be bounded by

$$\beta(j) \leq \pi^{-\frac{1}{2}} \frac{\Gamma(\frac{d-1}{2})}{\Gamma(\frac{d}{2})} \sum_{i=l-j}^{l} \arccos(\min\{x_i, 1\}) = \mu \sum_{i=l-j}^{l} \arccos(\min\{x_i, 1\})$$

where $\mu = \pi^{-\frac{1}{2}} \frac{\Gamma(\frac{d-1}{2})}{\Gamma(\frac{d}{2})}$ is a constant related to the dimension d, and $\Gamma \cdot$ is the gamma function. $x_i = \frac{db_i}{r}$, where db_i is the distance from point c_b to the line the boundary of b, b_i. When $x_i > 1$, b and b_i have no intersection, and thus does not contribute to the sum.

Algorithm 3 shows how to use Eq. (3) to prune the candidate buckets. Given a desired recall λ, for each bucket b, we sort its candidate buckets in decreasing order of their distances to b, and take the sum of $arccos(x)$ starting from the most distant bucket until the sum reaches the error bound $1 - \lambda$. We stop and prune the buckets encountered so far.

6 Experimental Evaluation

In this section, we present an extensive evaluation of DiskJoin, using three large vector datasets and three alternative baseline methods based on prior art. Our evaluation aims to answer the following questions:

- *How efficient is DiskJoin compared to related prior art, when the memory is small with respect to the dataset?*
- *How do the task configurations (i.e., target recall, distance threshold, and memory budget) affect the performance of DiskJoin?*
- *How effective are our designs (i.e., Belady's algorithm, task reordering, and pruning) in improving the efficiency of DiskJoin?*

6.1 Experimental Settings

Datasets. We use 3 datasets of varied characteristics. Deep100M (a subset of Deep1B [5]) is a vector dataset of image embeddings produced by the last fully connected layer of the GoogLeNet model [49];
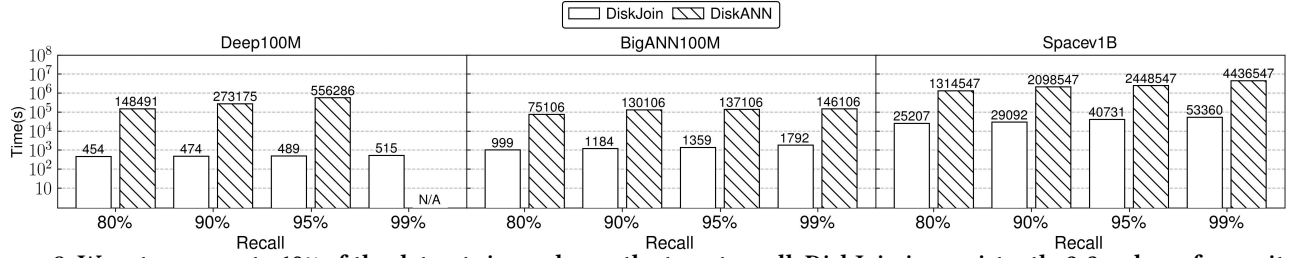
Figure 8: We set memory to 10% of the dataset size and vary the target recall. DiskJoin is consistently 2-3 orders of magnitude faster than DiskANN. DiskANN cannot achieve 99% recall for Deep100M due to the approximation errors in vector compression.
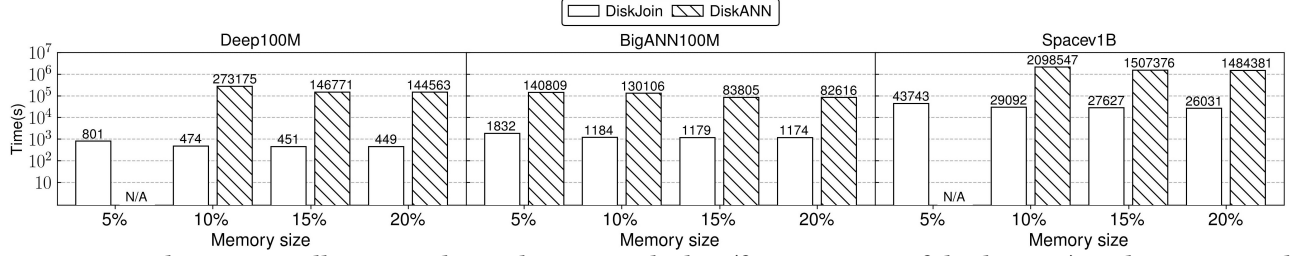
Figure 9: We set the target recall to 90% and vary the memory budget (from 5% to 20% of the data size). DiskJoin is 2-3 orders of magnitude faster than DiskANN. In low memory settings, DiskAnn may fail to achieve the target recall due to low vector compression accuracy.

it contains 100M vectors with 96 dimensions. BigANN100M contains 100M vectors that are scale-invariant feature transform (SIFT) descriptors of images, with 128 dimensions. SPACEV1B consists of more than 1.4 billion vectors of 100 dimensions, where each vector is a document embedding produced by the Microsoft SpaceV Superior model. The dataset statistics are summarized in Figure 6.

Baselines. To the best of our knowledge, no existing systems target disk-based vector similarity join. Therefore, we use the following prior art for related problem settings as baselines.

- DiskANN [48] is a popular state-of-the-art index for disk-based vector similarity search. As discussed in Section 1, we first build a DiskANN index and then use each vector in the dataset as a query to search the index for ϵ-neighbors. DiskANN is designed to search the top-k similar neighbors for a query, and we adjust the value of k to control the recall for vector similarity join. Other disk-based vector indexes (e.g., Starling [55]) may outperform DiskANN for *some* datasets. However, the speedups they report compared to DiskANN are modest. Given that DiskJoin demonstrates much more significant speedups over DiskANN (50x) for similarity join, such disk-based vector index baseline alternatives do not alter our conclusions.

- ClusterJoin [18] is a distributed similarity join solution. We implement a single-node in-memory version for fair comparison.

- RSHJ [61] is an in-memory vector similarity join solution using locality-sensitive hashing (LSH).

Platforms. We conduct the experiments on two AWS servers, one server is equipped with a 48-core Intel Xeon Platinum 8375C CPU @ 2.90GHz, 96GB RAM, and 2 × 1.3TB NVMe SSDs, while the other server has a 72-core Intel Xeon CPU E5-2686 v4 @ 2.30GHz, 512GB RAM, and 8 × 1.8TB NVMe SSDs. We use the first server for all

experiments on the Deep100M and BigANN100M datasets, and the second server for all experiments on the SPACEV1B dataset.

Implementation. We implement DiskJoin as a separate engine, independent from any RDBMSs. Future integration into RDBMS is certainly possible, and our algorithms would easily translate in this context. To prevent interference of OS page caching, we enabled the O_DIRECT flag in all file manipulation operations.

Evaluation protocol. We are interested in the efficiency of vector similarity join and thus use the end-to-end execution time—which includes any necessary index construction time (e.g., for DiskJoin and DiskANN)—as the main performance metric. We restrict the memory usage to be a small portion (e.g., 5%-20%) of the vector dataset size for two purposes: (1) it ensures that the compared methods use the same amount of memory, enabling a fair comparison and (2) it allows us to emulate scenarios where vector datasets are so large that they exceed available memory, which is common in practice. By default, we use a memory that is 10% of the dataset size, configure the target recall as $\lambda = 0.9$, and set the distance threshold ϵ for similar vector pairs such that each vector has 100 similar vectors on average.

6.2 Comparison with ClusterJoin and RSHJ

We first evaluate the performance of DiskJoin against the other similarity join baselines. Figure 7 reports the execution time and number of distance computations (DC) for DiskJoin, ClusterJoin, and RSHJ on BigANN dataset subsets of varying sizes: 100K, 1M and 10M. As ClusterJoin is an exact algorithm, we configure DiskJoin to achieve a recall of 99.5% for a fair comparison. The recall of RSHJ, despite our best efforts to tune its parameters, reaches only 97.6% and cannot be improved further. The memory budget of DiskJoin is constrained to 10% of the dataset size. RSHJ fails to run on the 1M and 10M datasets due to its prohibitive $O(n^2)$ memory consumption. Despite being a disk-based approach, DiskJoin significantly outperforms

the in-memory ClusterJoin and RSHJ. RSHJ performs particularly poorly due to a sub-procedure with $O(n^3)$ time complexity. Beyond that, the performance gap is primarily attributed to the substantial difference in the number of candidate pairs for distance computation in the verification phase. Notably, the number of distance computations in DiskJoin grows linearly with dataset size, whereas ClusterJoin exhibits near-quadratic growth. Considering the high overhead of network communication and disk I/O in a MapReduce environment, the distributed execution of CluserJoin is unlikely to outperform DiskJoin, while incurring higher resource costs.

> *Key takeaway:* DiskJoin is significantly faster than in-memory state-of-the-art methods for similarity join, because it reduces distance computation by orders of magnitude.

6.3 Performance under Varied Configurations

In this section, we evaluate the performance of DiskJoin with respect to all its parameters. We compare with DiskANN, which is the only disk-based method in our baselines.

Target recall. Figure 8 reports the execution time of DiskJoin and DiskANN when changing the target recall. Note that the y-axis is in log scale. Since DiskANN usually can not finish execution within one day, we estimate its execution time by sampling a subset of the vectors as queries. In particular, we randomly sample 1‰ of the vectors, and this yields 100K-1M vectors for the datasets, which make the sample sets sufficiently large for accurate time estimation. The execution time of DiskANN is omitted at 99% recall on Deep100M because DiskANN uses Product Quantization (PQ) [31] to compress vectors and stores the compressed dataset in memory. Under our default 10% memory size constraint, DiskANN can not reach a recall of 99% for Deep100M due to the approximation errors introduced by vector compression.

This experiment shows that DiskJoin significantly outperforms DiskANN for all datasets and target recalls. In particular, the speedup of DiskJoin over DiskANN is 52× at the minimum and 1137× at the maximum (i.e., on Deep100M at a recall of 95%). As we will show later with detailed profiling, this is because DiskANN suffers from long disk I/O time and computation time while DiskJoin's optimizations for disk I/O and candidate pruning effectively reduce both IO and computation costs. Both DiskANN and DiskJoin run longer to achieve higher target recalls. This is because DiskJoin needs to check more bucket pairs, and DiskANN needs to search more neighbors for each query. However, the execution time of DiskANN increases faster than that of DiskJoin. For example, to improve the recall from 80% to 99% for Spacev1B, execution time increases 4× for DiskANN but only about 2× for DiskJoin.

Memory constraint. Figure 9 reports the execution time of DiskJoin and DiskANN under varying memory constraints (from 5% to 20% of the data size), for fixed target recall (90%). For DiskJoin, the memory constraint determines the number of buckets that can be cached in memory. For DiskANN, two factors affect its memory consumption: the compression ratio of the PQ for approximate vectors and the portion of the proximity graph index that is cached in memory. We first select the PQ compression ratio that is closest to the memory constraint, and cache the proximity graph in the remaining memory. This is because DiskANN uses the approximate
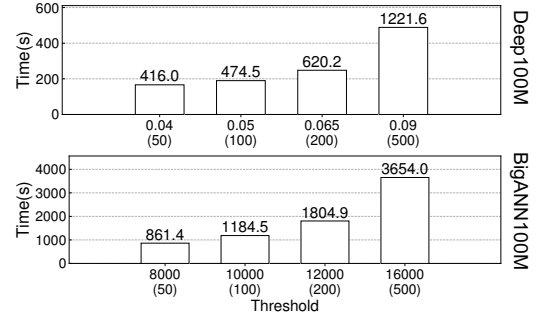
Figure 10: Execution time of DiskJoin increases as we increase the distance threshold ϵ, but growth remains sublinear for settings up to 500 similar neighbors, on average.
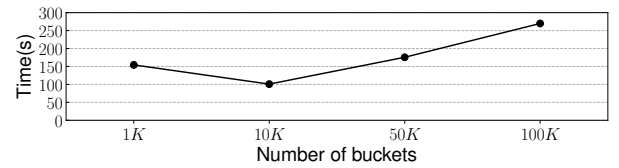

Figure 11: The number of buckets affects the performance of DiskJoin over a subset of BigANN100M with 10M vectors. The best time is achieved at around 1‰ of the data size.

vectors for proximity graph traversal, and thus the compressed vectors need to be accurate for good recall. DiskANN fails to reach 90% recall on Deep100M and Spacev1B when the memory constraint is 5% of the dataset size due to vector compression errors.

We observe that DiskJoin is consistently 2-3 orders-of-magnitude faster than DiskANN. We observe diminishing performance gains when enlarging the memory constraint beyond 10%, i.e., the speedup of increasing memory size from 10% to 20% is much smaller than increasing the memory size from 5% to 10%. As we will show in later experiments, this is because at a memory size of 10% dataset size, DiskJoin's execution time is dominated by computation rather than IO. Thus, further increasing the memory to reduce IO has a limited effect on overall execution time. This indicates a significant benefit of DiskJoin: it only needs a small amount of memory (with respect to the dataset) to avoid making IO the bottleneck. For DiskANN, the performance improvement is primarily attributed to the enhanced accuracy of the compressed vectors. For example, on Deep100M, increasing the memory from 10% to 15% of the dataset size results in a speedup of 1.86× due to more accurate compressed vectors. However, increasing the memory from 15% to 20% has nearly no reduction in execution time, as the vectors are already accurate, and the additional memory is used to cache the proximity graph index.

Distance threshold. In the next experiment, we vary the distance threshold ϵ, so that each vector has 50, 100, 200 and 500 similar neighbors, on average. Figure 10 shows that the execution time of DiskJoin increases more quickly for BigANN100M than Deep100M. This is in line with the results in Figure 8, i.e., when increasing the target recall, DiskJoin's execution time grows slower for Deep100M than BigANN100M, suggesting that finding neighbors is more difficult in BigANN100M. Overall, growth remains sublinear for settings up to 500 similar neighbors, on average.
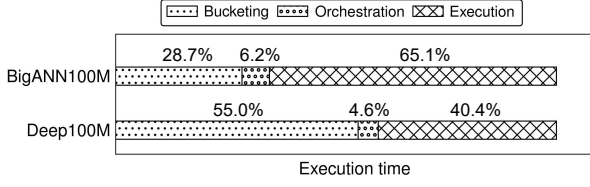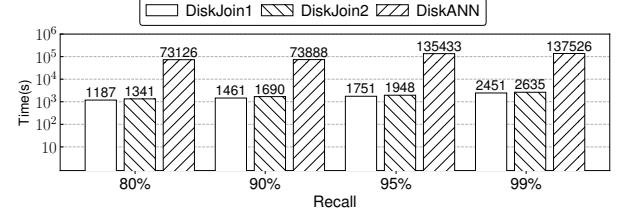
Figure 12: Execution time breakdown of DiskJoin into three parts: bucketing, orchestration and execution.

Figure 13: Execution time of cross-join on two Deep datasets of size 50M and 100M. DiskJoin1 denotes reordering the larger dataset and caching the smaller dataset, while DiskJoin2 means the opposite.

Fragmentation level	No. of extents	Execution time (s)
None	536	108.1
1024KB	1579	110.7
128KB	6530	111.4
16KB	265565	123.6

Figure 14: Performance of DiskJoin is robust under moderate fragmentation conditions, with some degradation only when the number of extents becomes extremely large (at 16KB).

Number of buckets. Figure 11 shows the execution time on a 10M size subset of BigANN100M, when varying the number of buckets used by DiskJoin as 0.1‰, 1‰, 5‰, and 1% of the dataset size. DiskJoin achieves best performance when the number of buckets is around 1‰ of the dataset size. With fewer buckets, the space partitioning is less fine-grained and thus produces more candidate pairs. Moreover, the number of buckets that can be cached decreases, leading to a higher cache miss rate and increased execution time. Conversely, when the number of buckets is large (e.g., 1%), the average bucket size drops to around 100 vectors. Considering that bucket sizes are not evenly distributed, many buckets may occupy only a few pages or even less than a page, resulting in significant read amplification and degraded I/O efficiency.

Execution time breakdown. Figure 12 reports the running time of the three main phases for DiskJoin: bucketing, orchestration, and execution. Bucketing maps the vectors to buckets and stores the buckets on disk; orchestration includes dependency graph construction, task reordering, and cache management; execution loads the buckets from disk and performs computations. The results show that the orchestration part, which encompasses our main optimizations, has a small overhead (around 5% of the total execution time). For Deep100M, the time spent on bucketing exceeds that of execution. This is because most vector pairs are filtered out, leaving a small number of candidate pairs for verification.

Randomness sensitivity. The execution time and recall depend on the quality of vector bucketization, which involves random center selection. To evaluate the robustness of this randomness, we repeat the experiment on the 10M subset of BigANN100M fifty times. The results show that performance is robust: the mean recall is 0.903 with a standard deviation of 0.005, and the mean execution time is 276.02 seconds with a standard deviation of 12.62 seconds. This low variance indicates that the performance is largely insensitive to the specific random centers chosen.

Stress-testing DiskJoin. DiskJoin assumes ideal sequential disk layout. In practice, however, files may split into non-contiguous pieces (extents) as the file system ages, a phenomenon known as file system fragmentation. In this experiment, we stress-test DiskJoin under such degraded disk conditions. We create three 100GB file systems, fill them with small files of varying sizes (16KB, 128KB and 1024KB), and then delete a random subset of files to simulate fragmentation. Figure 14 shows the execution time of DiskJoin and the number of extents in the disk index file on the 10M size subset of BigANN100M across different fragmentation level. As fragmentation increases, the disk index is broken into more extents, increasing

execution time. However, the performance impact remains minimal at moderate fragmentation level (1024KB and 128KB) and only becomes noticeable—still under a 10% slowdown—under severe fragmentation. This is due to the fact that SSDs do not physically seek, so there is no difference between non-sequential access and sequential access. The performance of DiskJoin only degrades when the file system is so fragmented that many extents of the disk index have size less than 4KB, causing read amplification.

Cross-join. We evaluate DiskJoin's performance in cross-joins using two disjoint subsets of the Deep1B dataset [5], of sizes 50M and 100M, respectively. The distance threshold and memory constraint are set to their default values. As discussed in Section 3, DiskJoin may conduct cross-join in two ways: (1) reordering the larger dataset and caching the smaller dataset or (2) the reverse. These are denoted as DiskJoin1 and DiskJoin2 in Figure 13, respectively. Both DiskJoin1 and DiskJoin2 outperform DiskANN consistently, across different target recalls, and DiskJoin1 runs slightly faster than DiskJoin2. This is expected, as DiskJoin1 has shorter disk read time compared with DiskJoin 2 (Section 3).

> *Key takeaway:* The execution time of DiskJoin is consistently 2-3 orders of magnitude lower than DiskANN, and remains robust across varied configuration settings.

6.4 Performance Analysis

In this section, we profile the execution statistics of DiskANN and DiskJoin to understand their performance. For all experiments, we use the default configurations (Section 6.1).

Figure 15 breaks down the execution time of DiskANN and DiskJoin into IO time and computation time. Note that the lengths of the bars do not indicate the execution time of the solutions, instead, we report the execution time next to the bars. The results show that disk IO takes up about 70% of DiskANN's execution time on both datasets, while DiskJoin only spends 21.4% and 16.7%
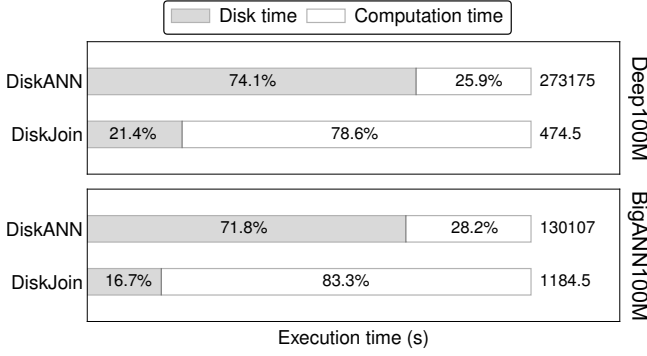
Figure 15: Breaking down the execution time for DiskJoin and DiskANN. Note that the lengths of the bar do not correspond to total time.

		Total (GB)	Useful (GB)	Amp.
Deep	DiskJoin	181	180.3	1.0039
	DiskANN	139,485	19,427	7.18
BigANN	DiskJoin	265	264.3	1.0026
	DiskANN	115,854	19,118	6.06

Figure 16: The disk traffic of DiskJoin and DiskANN. *Amp.* is the ratio of total disk traffic over the useful disk traffic.

of the execution time on disk IO respectively. Figure 16 further explains this result, reporting the useful disk traffic against the total disk traffic for each approach (and their ratio, which is the read amplification). We see that DiskANN needs to read far more data from the disk than DiskJoin. In particular, the average disk read volume per vector is 1.4MB and 1.2MB for DiskANN while DiskJoin only reads 1.9KB and 2.8KB. The large difference between DiskJoin's and DiskANN's disk traffic (i.e., over 400x) is due to three reasons. (1) DiskANN suffers from severe read amplification due to the random access pattern of the proximity graph index. As a result, only a small fraction of the data read from disk is actually used by DiskANN. In contrast, DiskJoin enjoys nearly no disk amplification thanks to its bucket-based block disk access. (2) DiskJoin optimizes for disk I/O through its dynamic memory cache and task reordering. (3) DiskJoin's pruning technique successfully prunes most of the candidate pairs, further reducing the disk I/O time and computation time, which we will show later.

Overall, these results indicate that, with our optimizations, disk IO is no longer the bottleneck for vector similarity joins. Instead, the limiting factor becomes computation and the execution time of DiskJoin can be further accelerated by using GPUs for computation.

> *Key takeaway:* Remarkably, DiskJoin achieves nearly 0 read amplification, eliminating disk IO as the bottleneck.

6.5 Effectiveness of DiskJoin Designs

In this section, we analyze the impact of our three optimizations: cache management using Belady's algorithm, task reordering, and candidate pruning. Cache management and task reordering preserve the graph structure and thus do not reduce recall, while candidate pruning may affect recall, as it modifies the dependency graph. However, despite its heuristic nature, pruning performs very
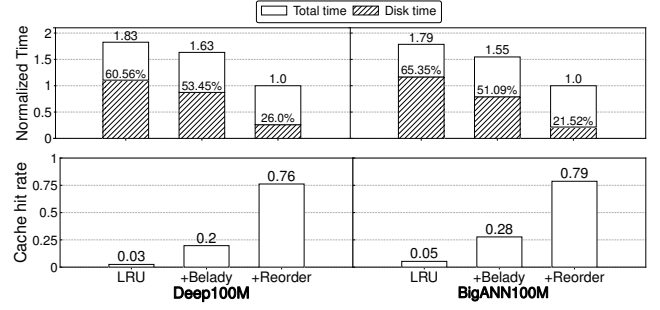
Figure 17: Normalized execution time and cache hit rate when enabling the task orchestration optimizations.

well empirically, demonstrating significant gains in efficiency with only minimal impact on recall (approximately a 0.1% reduction in our experiments). Below, we present an experimental analysis of how these optimizations influence efficiency.

First, we consider our task orchestration optimizations. In Figure 17, the method annotated as LRU disables both cache optimizations, instead using the node id to determine task order and LRU as the cache replacement policy; +Belady uses Belady's algorithm instead of LRU; finally, +Reorder augments +Belady with task reordering. For ease of comparison, we normalize the execution time results by the execution time of +Reorder, and report the percentage of disk IO time within total execution time.

Both optimizations are effective at reducing the execution time of DiskJoin, as they improve the cache hit rate when accessing the buckets. When the two optimizations are disabled (LRU), disk IO takes up more than half of the execution time, while disk IO is no longer the bottleneck when the optimizations are used (+Reorder). Task reordering is particularly effective in improving the cache hit rate: When both optimizations are used, the cache hit rate is above 75% for both datasets. This is remarkable because the memory size is only 10% of the dataset size. The cache hit rate of LRU is below 10% because similarity join does not have popular buckets, and a bucket that was recently accessed may not be accessed in the future, which constitutes an adversarial case for LRU.

Finally, we examine the effect of candidate bucket pruning, which directly influences the average degree of the bucket dependency graph—and consequently, the computational and disk I/O costs. Figure 18 reports the number of candidate vector pairs[2] and execution time under varying distance thresholds. The results show that pruning leads to a substantial reduction in both the number of candidate vector pairs and execution time across all settings. For example, on the Deep100M dataset with a threshold of 0.05, pruning reduces the number of candidate pairs from 5.59×10^{12} to 1.81×10^{11}, resulting in a drop in execution time from 5188.1 seconds to just 474.5 seconds. The trend holds for larger thresholds as well: at 0.09, pruning still reduces the execution time by nearly 9× despite the increased candidate volume.

> *Key takeaway:* Our ablation study shows that all of DiskJoin's optimizations are useful in reducing disk access and runtime, with minimal impact on recall.

---

[2]The number of candidate vector pairs is proportional to the number of candidate buckets—both reflect the computational and I/O costs; we report the more fine-grained statistic here.

Dataset	Threshold (# neighbors)	No. of candidates		Execution time (s)	
		w/o pruning	w/ pruning	w/o pruning	w/ pruning
Deep100M	0.04 (50)	4.19×10^{12}	1.12×10^{11}	4,204	416
	0.05 (100)	5.59×10^{12}	1.81×10^{11}	5,188	475
	0.065 (200)	7.31×10^{12}	3.93×10^{11}	7,671	620
	0.09 (500)	9.66×10^{12}	1.14×10^{12}	10,696	1,222
BigANN100M	8000 (50)	5.89×10^{12}	4.60×10^{11}	8,235	861
	10000 (100)	7.76×10^{12}	7.78×10^{11}	9,930	1,185
	12000 (200)	9.52×10^{12}	1.26×10^{12}	14,793	1,805
	16000 (500)	1.28×10^{13}	2.57×10^{12}	20,813	3,654

Figure 18: Impact of candidate bucket pruning on candidate count and execution time. Pruning yields significant performance gains across all thresholds.

7 Conclusion

In this paper, we study the problem of similarity-based self-join over high-dimensional, large vector datasets. We focus on disk-based solutions and observe that disk access dominates the execution time of disk-based vector similarity join due to read amplification and repetitive data access. We propose DiskJoin, which relies on smart bucket-wise processing of SSJs to reduce disk IO. DiskJoin carefully manages the in-memory data cache to improve cache hit rate, which is achieved by ordering the tasks for temporal locality and using a cache eviction policy that fully utilizes data access information. Through extensive experiments, we show that DiskJoin achieves consistently 2-3 orders-of-magnitude improvement in execution time and all but eliminates read amplification. Remarkably, DiskJoin removes disk access as a bottleneck in vector similarity self-join.

References

[1] Amro Abbas, Kushal Tirumala, Dániel Simig, Surya Ganguli, and Ari S Morcos. 2023. Semdedup: Data-efficient learning at web-scale through semantic deduplication. *arXiv preprint arXiv:2303.09540* (2023).

[2] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. 2023. Similarity Search in the Blink of an Eye with Compressed Indices. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3433–3446.

[3] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*. 918–929.

[4] Jane Arthurs, Sophia Drakopoulou, and Alessandro Gandini. 2018. Researching youtube. *Convergence* 24, 1 (2018), 3–15.

[5] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[6] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Syst. J.* 5, 2 (1966), 78–101.

[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. *ACM SIGMOD Record* 30, 2 (2001), 379–388.

[8] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-textual similarity joins. *Proceedings of the VLDB Endowment* 6, 1 (2012), 1–12.

[9] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient processing of spatial joins using R-trees. *ACM SIGMOD Record* 22, 2 (1993), 237–246.

[10] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009), 1–58.

[11] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys* 41, 3 (2009), 1–58.

[12] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A primitive operator for similarity joins in data cleaning. In *22nd International Conference on Data Engineering*. 5–5.

[13] Gang Chen, Keyu Yang, Lu Chen, Yunjun Gao, Baihua Zheng, and Chun Chen. 2016. Metric similarity joins using MapReduce. *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (2016), 656–669.

[14] 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. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[15] Paolo Ciaccia, Marco Patella, Pavel Zezula, et al. 1997. M-tree: An efficient access method for similarity search in metric spaces. In *Vldb*, Vol. 97. 426–435.

[16] Henrique B da Silva, Zenilton KG do Patrocínio, Guillaume Gravier, Laurent Amsaleg, Arnaldo de A Araújo, and Silvio Jamil F Guimaraes. 2016. Near-duplicate video detection based on an approximate similarity self-join strategy. In *14th International Workshop on Content-Based Multimedia Indexing*. 1–6.

[17] Henrique B. da Silva, Zenilton K. G. do Patrocínio, Guillaume Gravier, Laurent Amsaleg, Arnaldo de A. Araújo, and Silvio Jamil F. Guimarães. 2016. Near-duplicate video detection based on an approximate similarity self-join strategy. In *CBMI*. 1–6.

[18] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. Clusterjoin: A similarity joins framework using map-reduce. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1059–1070.

[19] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[20] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. 2003. Similarity join in metric spaces using ed-index. In *Database and Expert Systems Applications: 14th International Conference*. 484–493.

[21] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. [n.d.]. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment* 12, 5 ([n. d.]).

[22] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of 2012 International Conference on Management of Data*. 541–552.

[23] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2024. Practical and Asymptotically Optimal Quantization of High-Dimensional Vectors in Euclidean Space for Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2409.09913* (2024).

[24] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: killing one thousand queries with one stone. *Proc. VLDB Endow.* 5, 6 (Feb. 2012), 526–537. https://doi.org/10.14778/2168651.2168654

[25] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. 2023. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*. 3406–3416.

[26] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik PA Lensch. 2022. Ggnn: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data* 9, 1 (2022), 267–279.

[27] Bikash Gyawali, Lucas Anastasiou, and Petr Knoth. 2020. Deduplication of Scholarly Documents using Locality Sensitive Hashing and Word Embeddings. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*. Marseille, France, 901–910.

[28] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 9, 1 (2015), 1–12.

[29] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.

[30] Vipin Jain, BINDOO Malviya, and SATYENDRA Arya. 2021. An overview of electronic commerce (e-Commerce). *The journal of contemporary issues in business and government* 27, 3 (2021), 665–670.

[31] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

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

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

[34] Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, Jyothi Vedurada, et al. 2024. Bang: Billion-scale approximate nearest neighbor search using a single gpu. *arXiv preprint arXiv:2401.11324* (2024).

[35] Douwe Kiela and Léon Bottou. 2014. Learning image embeddings using convolutional neural networks for improved multi-modal semantics. In *Proceedings of the 2014 Conference on empirical methods in natural language processing*. 36–45.

[36] Juhwan Kim, Jongseon Seo, Jonghyeok Park, Sang-Won Lee, Hongchan Roh, and Hyungmin Cho. 2022. ES4D: Accelerating Exact Similarity Search for High-Dimensional Vectors via Vector Slicing and In-SSD Computation. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. 298–306. https://doi.org/10.1109/ICCD56317.2022.00051

[37] Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, et al. 2022. Matryoshka representation learning. *Advances in Neural Information Processing Systems* 35 (2022), 30233–30249.

[38] Hangyu Li, Sarana Nutanong, Hong Xu, Foryu Ha, et al. 2018. C2Net: A network-efficient approach to collision counting LSH similarity join. *IEEE Transactions on Knowledge and Data Engineering* 31, 3 (2018), 423–436.

[39] Chen Luo and Anshumali Shrivastava. 2018. Arrays of (locality-sensitive) count estimators (ace) anomaly detection on the edge. In *Proceedings of the 2018 World Wide Web Conference*. 1439–1448.

[40] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[41] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. 2019. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition* 96 (2019), 106970.

[42] Usman Naseem, Imran Razzak, Katarzyna Musial, and Muhammad Imran. 2020. Transformer based deep intelligent contextual embedding for twitter sentiment analysis. *Future Generation Computer Systems* 113 (2020), 58–69.

[43] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceedings of 2011 International Conference on Management of data*. 1033–1044.

[44] Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du, Yueguo Chen, and Anthony KH Tung. 2012. Efficient and scalable processing of string similarity join. *IEEE Transactions on Knowledge and Data Engineering* 25, 10 (2012), 2217–2230.

[45] Viktor Sanca and Anastasia Ailamaki. 2024. Efficient Data Access Paths for Mixed Vector-Relational Search. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. 1–9.

[46] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, et al. 2022. Laion-5b: An open large-scale dataset for training next generation image-text models. *Advances in Neural Information Processing Systems* 35 (2022), 25278–25294.

[47] 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. *arXiv preprint arXiv:2105.09613* (2021).

[48] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*. 13748–13758.

[49] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[50] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2024. Scalable billion-point approximate nearest neighbor search using {SmartSSDs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1135–1150.

[51] Flavian Vasile, Elena Smirnova, and Alexis Conneau. 2016. Meta-prod2vec: Product embeddings using side-information for recommendation. In *Proceedings of the 10th ACM conference on recommender systems*. 225–232.

[52] Hanli Wang, Fengkuangtian Zhu, Bo Xiao, Lei Wang, and Yu-Gang Jiang. 2015. GPU-based MapReduce for large-scale near-duplicate video retrieval. *Multimedia Tools and Applications* 74 (2015), 10515–10534.

[53] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering? An adaptive framework for similarity join and search. In *Proceedings of 2012 International Conference on Management of Data*. 85–96.

[54] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2023. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. *Advances in Neural Information Processing Systems* 36 (2023), 15738–15751.

[55] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[56] Ye Wang, Ahmed Metwally, and Srinivasan Parthasarathy. 2013. Scalable all-pairs similarity search in metric spaces. In *19th International Conference on Knowledge Discovery and Data Mining*. 829–837.

[57] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.

[58] Xiao Wu, Alexander G Hauptmann, and Chong-Wah Ngo. 2007. Practical elimination of near-duplicates from web video search. In *Proceedings of the 15th ACM international conference on Multimedia*. 218–227.

[59] Xiao Wu, Chong-Wah Ngo, Alexander G Hauptmann, and Hung-Khoon Tan. 2009. Real-time near-duplicate elimination for web video search with content and context. *IEEE Transactions on Multimedia* 11, 2 (2009), 196–207.

[60] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the VLDB Endowment* 1, 1 (2008), 933–944.

[61] Chenyun Yu, Sarana Nutanong, Hangyu Li, Cong Wang, and Xingliang Yuan. 2016. A generic method for accelerating LSH-based similarity join processing. *IEEE Transactions on Knowledge and Data Engineering* 29, 4 (2016), 712–726.

[62] Song Yu, Shengyuan Lin, Shufeng Gong, Yongqing Xie, Ruicheng Liu, Yijie Zhou, Ji Sun, Yanfeng Zhang, Guoliang Li, and Ge Yu. 2025. A Topology-Aware Localized Update Strategy for Graph-Based ANN Index. *arXiv preprint arXiv:2503.00402* (2025).

[63] Minjia Zhang and Yuxiong He. 2019. Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 1673–1682.

[64] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. 2023. Fast, Approximate Vector Queries on Very Large Unstructured Datasets. In *20th Symposium on Networked Systems Design and Implementation*. 995–1011.

[65] Weijie Zhao, Shulong Tan, and Ping Li. 2020. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1033–1044.

[66] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: segment graph for range-filtering approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.