App2Exa: Accelerating Exact kNN Search via Dynamic Cache-Guided Approximation

Ke Li [1] , **Leong Hou U**[2] , **Shuo Shang**[1*]
[1]University of Electronic Science and Technology of China, Chengdu, China
[2]University of Macau, Macau, China
like_like@std.uestc.edu.cn, ryanlhu@um.edu.mo, jedi.shang@gmail.com

Abstract

The k-nearest neighbor (kNN) query is a cornerstone of similarity-based applications across various domains. While prior work has enhanced kNN search efficiency, it typically focuses on approximate methods for high-dimensional data or exact methods for low-dimensional data, often assuming static query and data distributions. This creates a significant gap in accelerating exact kNN search for low-to-medium dimensional data with dynamic query distributions. To fill this gap, we propose App2Exa, a cache-guided framework that integrates approximate and exact kNN search. App2Exa utilizes a dynamically maintained cache graph index to retrieve approximate results, which subsequently guide exact search using a VP-Tree with a best-first strategy. A benefit-driven caching mechanism further optimizes performance by prioritizing vectors based on frequency, recency, and computational cost. Experimental results demonstrate that App2Exa significantly boosts efficiency, providing a robust and scalable solution for evolving query patterns and enabling exact kNN search to support higher dimensionality more effectively.

1 Introduction

The k-nearest-neighbor (kNN) query is widely employed in similarity-based tasks across various domains, such as image retrieval [Peng *et al.*, 2022], web search [Bae *et al.*, 2009], and recommendation [Halder *et al.*, 2024]. Let $\mathcal{V}$ represent a finite set of vectors in a d-dimensional space, and $dist(\cdot)$ denote a distance function between two vectors. Given a query vector v_q and a positive integer k, the kNN query identifies k vectors $v \in \mathcal{V}$ such that $dist(v, v_q)$ is smaller than that of all other vectors in $\mathcal{V}$.

The efficiency of kNN search can be greatly enhanced through indexing techniques. For exact kNN search, tree-based indexes such as KD-trees [Bentley, 1975] and VP-trees [Yianilos, 1993] are widely used. However, as vector dimensionality increases, the efficiency of exact search declines due to the "curse of dimensionality" [Beyer *et al.*, 1999].

Dimension	Principles	Algorithms
medium-high	graph-based	KGraph, HNSW, MRNG
↑	hash-based	FAISS(+PQ codes), FALCONN
↑	cache-guided	**App2Exa***
low-medium	tree-based	ball-tree*, KD-Tree*, VP-Tree*
↑ Dimension		* Ensure exact result

Table 1: Overview of kNN algorithms

In high-dimensional vectors space, graph-based approaches like kGraph [Dong *et al.*, 2011] and HNSW [Malkov and Yashunin, 2020] have emerged as popular alternatives. Table 1 provides a comparison of representative indexing methods alongside the approach proposed in this work. While extensive research has focused on improving the efficiency of kNN search, efforts to support higher dimensionality while ensuring exact results remain limited. Moreover, most existing methods assume static query and data distributions, offering minimal consideration for scenarios involving periodic or dynamic updates. In many real-world applications [Li *et al.*, 2023; Li *et al.*, 2022], query distributions evolve over time (e.g., varying hot queries) while the database of object vectors remains static. Certain object vectors experience periods of high-frequency queries due to evolving user needs or external factors. For instance, medical articles may receive frequent queries during health crises (e.g., the COVID-19 pandemic or flu season), user interests in travel destinations may fluctuate seasonally, and students or educators focus on specific topics during exams or trending research periods. Advancements in embedding techniques now enable such content to be efficiently transformed into feature vectors for similarity-based computations. In these dynamic scenarios, obtaining exact kNN results becomes critical for ensuring accurate and reliable outcomes.

Caching is commonly employed to store the results of frequently occurring queries, which enables "hot" queries to be answered directly, thereby reducing the amount of computation and improving query latency [Zulfa *et al.*, 2020]. Fig. 1 illustrates the speed-up achieved by our cache-guided approach on the SIFT dataset used in our experiments. By leveraging approximate kNN results with a cache update factor ϵ (cf. Alg. 3), our method delivers a substantial performance boost, potentially allowing exact kNN search to handle higher dimensionality more efficiently. While caching is

*Corresponding author.

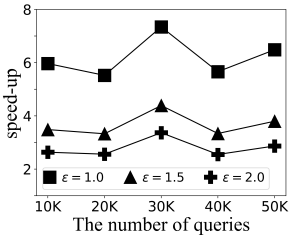


Figure 1: The speed-up performance of App2Exa

widely used in web search scenarios, there are important differences between general web search and vector kNN search in terms of query matching and cost variation. In web search, results from one query are considered a match to another if they share the same value (e.g., identical text), as exact matching is required. In contrast, for embedding vectors, exact matches are nearly impossible. In kNN queries, results are considered a match if their similarity ratio exceeds a predefined threshold. When a cache miss occurs, additional computation is required to obtain the query result, and some results can be significantly more costly to compute than others. To optimize query performance, it is crucial to develop a cost (or benefit) model that estimates the evaluation cost of a query. However, to the best of our knowledge, no prior work has addressed cost estimation for kNN queries within the context of any index structure.

To address these gaps, we propose a cache-guided approach called App2Exa, which bridges approximate and exact kNN search to improve efficiency (cf. Table 1). Initially, approximate kNN results are retrieved from dynamically maintained cached object vectors, supported by a tailored cache graph index. A benefit model for caching vectors is proposed, along with a benefit-driven cache (BDC) replacement algorithm. These results are then reused to accelerate exact kNN search using a VP-Tree index with a best-first strategy. Empirical results on real datasets show that App2Exa consistently outperforms the VP-Tree approach by at least $3x$ when using the BDC policy. Our method is particularly effective in environments where certain object vectors experience high-frequency retrieval during specific periods, while exact kNN results are still required. The main contributions of this paper can be summarized as follows.

- We introduce a benefit-driven caching mechanism that prioritizes vectors based on query frequency, recency, and computational cost, coupled with an incremental cache graph index to enhance retrieval efficiency.
- We develop a best-first strategy tailored for the VP-Tree, which cooperates with a guided distance to prune unqualified objects at an early stage.
- We propose App2Exa, a novel cache-guided approach that bridges approximate and exact kNN search to enhance efficiency.
- We conduct extensive experiments[1], demonstrating that our proposal significantly accelerates kNN search, with the benefit-driven cache performing as an optimal caching policy.

[1]The implementation is available at https://github.com/likemelike/App2ExaCache.git.

2 Related Work

2.1 The k Nearest Neighbor Search Problem

The kNN search problem is a fundamental component of similarity-based query tasks and has garnered significant attention from the data science community. Various approaches have been proposed to address this problem, including tree-based [Ram and Sinha, 2019], hash-based [Zhao *et al.*, 2023; Zheng *et al.*, 2022], and graph-based methods [Malkov and Yashunin, 2020]. As vector dimensionality increases, exact kNN search becomes increasingly challenging for two main reasons. First, the curse of dimensionality causes data points to become more uniformly distributed, making it harder to distinguish true neighbors, thereby reducing the meaningfulness of similarity search [Beyer *et al.*, 1999; Babenko and Lempitsky, 2016]. Second, traditional indexing methods, such as R-trees [Guttman, 1984] and KD-trees [Bentley, 1975], perform poorly in high dimensions. These methods divide the data space into orthotopes but struggle to effectively separate objects in high-dimensional spaces. Pivot-based indexes, such as the VP-Tree [Yianilos, 1993], are better suited for exact similarity search in metric spaces. The VP-Tree hierarchically partitions data based on a vantage point at each node and leverages the triangle inequality to prune irrelevant regions during search. The MVP-Tree [Bozkaya and Özsoyoglu, 1999] extends the VP-Tree by generalizing it to an m-ary structure. It organizes objects based on their distance to the vantage point, partitioning them into m groups of equal size. During search, the mean distance of each group is used to efficiently prune unqualified sub-trees, improving search performance in high-dimensional spaces. A recent study [Lampropoulos *et al.*, 2023] considers short-lived object vectors with limited queries before obsolescence. However, no existing research addresses varied query distributions for exact kNN in vector space. In contrast, LSH-based indexes [Liu *et al.*, 2021] are not well-suited for scenarios requiring exact search. More recently, graph-based solutions such as HNSW and MRNG [Fu *et al.*, 2019] have been proposed to ensure connectivity and enable efficient routing in poly-logarithmic time. However, these methods lack error guarantees on search results, making them unsuitable for directly answering exact kNN queries.

2.2 Cache Replacement Policies

Caching is widely used in web search engines to retrieve the top-k relevant documents (e.g., web pages) that match a text query. Maintaining a cache can be costly and may not be worthwhile if queries are evenly distributed. However, it is particularly effective when a small number of queries dominate in frequency and remain popular for a period of time [Frieder *et al.*, 2024]. Existing caching strategies can be classified into dynamic caching [Markatos, 2001; Long and Suel, 2006; Gan and Suel, 2009; Cambazoglu *et al.*, 2010] and static caching [Baeza-Yates *et al.*, 2007]. Dynamic caching stores results of recently accessed queries and adapts

Notation	Definition
$\mathcal{V}$	A collection of vectors
v_q	A query vector
v	A vector in $\mathcal{V}$
k	The query result quantity
$\mathcal{C}$	The vector cache
B	The budget of a vector cache
$\mathcal{R}/\mathcal{R}'$	The exact / approximate kNN results

Table 2: The summary of notations

quickly to changing query patterns. For example, the Least-Recently-Used (LRU) method [Jelenkovic and Radovanovic, 2004] replaces the least recently used result with the current query result upon a cache miss, while the First-In-First-Out (FIFO) replacement policy evicts the oldest cached result regardless of access patterns. Techniques like Least-Frequently-Used (LFU) replacement [Lee *et al.*, 2001] analyze past query logs to identify and cache queries that dominate in frequency. Although these approaches adapt well to dynamic query distributions, they may incur frequent update overhead for kNN vector search. Static caching [Ozcan *et al.*, 2012], on the other hand, focuses on storing results for the most popular queries based on historical data, which are updated periodically (e.g., daily).

3 Preliminaries

Let $\mathcal{V}$ be a finite vector set in a d-dimensional vector space $\mathbb{R}^d$, and $dist$ be a distance metric. For two vectors $v, v' \in \mathcal{V}$, the distance metric is typically the Euclidean distance, i.e., $dist(v, v') = \sqrt{\sum_{i=1}^{d}(v[i] - v'[i])^2}$. Table 2 summarizes the notations frequently used throughout the paper.

k-nearest-neighbor (kNN) query. Given a query vector $v_q \in \mathcal{V}$ and a positive integer k, a k-nearest-neighbor query problem returns a subset $\mathcal{R} \subseteq \mathcal{V}$, such that $|\mathcal{R}| = k$ and $\forall v \in \mathcal{R}, \forall v' \in \mathcal{V} \setminus \mathcal{R}, dist(v, v_q) \leq dist(v', v_q)$.

k-approximate-nearest-neighbor (kANN) query. Given a query vector $v_q \in \mathcal{V}$ and a positive integer k, the k-approximate-nearest-neighbor (kANN) query returns a set $\mathcal{R}'$ that is approximated to $\mathcal{R}$, which contains a proportion of ground-truth answers in $\mathcal{R}$.

In our setting, we assume that the query distribution evolves periodically, with certain object vectors being frequently retrieved as kNN results. The vector cache, as defined below, can be placed either at a proxy or the server. It optimizes the computation of approximate kNN searches and subsequently reduces the response time for exact queries.

Vector cache. Given a cache budget B, a vector cache $\mathcal{C}$ is a subset of $\mathcal{V}$ such that $|\mathcal{C}| \leq B$, where $|\mathcal{C}|$ denotes the total number of vectors in the cache.

4 App2Exa: A Cache-Guided Approach

In this section, we introduce App2Exa, a cache-guided approach designed to accelerate the exact computation of kNN queries in the context of evolving query distributions. Fig. 2 illustrates the overall framework of our proposal, which consists of three main components: cache-guided approximate

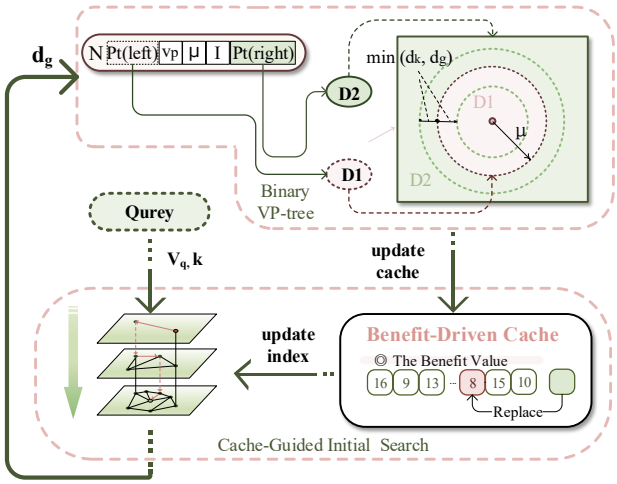


Figure 2: The components in App2Exa framework

kNN search, exact kNN search, and a dynamically updated, benefit-driven caching mechanism. Given a query v_q and a positive integer k, the high-level process of our approach is as follows. First, we retrieve approximate kNN results for the query vector v_q, denoted as $\mathcal{R}'$, from the dynamically maintained cache of vectors. To ensure efficient search, we develop an incremental cache graph index and apply a benefit-driven caching strategy to optimize the results. The maximum distance between any vector in $\mathcal{R}'$ and v_q, denoted as d_g, is then determined. Next, we perform the exact kNN search using an improved VP-Tree. The guide value d_g is used to effectively prune the VP-Tree search space, and a best-first strategy is employed to further enhance efficiency. Finally, we obtain the exact kNN results, denoted as $\mathcal{R}$. The maximum distance between any vector in $\mathcal{R}$ and v_q is denoted as d_k. Based on the approximate ratio between $\mathcal{R}$ and $\mathcal{R}'$, the cache and the cache graph index is dynamically updated. Specifically, if the approximate ratio meets or exceeds a given threshold ϵ, we update the cache using a benefit-driven replacement policy.

4.1 Cache-Guided Approximate kNN Search

Incremental Cache Graph Index

We investigate the cache index structure, focusing on maximizing speed-up and efficiently supporting dynamic updates while ensuring fast result retrieval. To achieve this, we propose an incremental cache graph index, denoted as $\mathcal{G}(\mathcal{C}, l_{max}, \mathcal{M}(l, v), m)$. Here, $\mathcal{C}$ represents the vector cache and l_{max} denotes the maximum layer. The function $\mathcal{M}(l, v)$ maps each vector v to its connections at a specific layer l, with a maximum of m connections. As illustrated in the bottom-left of Fig. 2, the structure forms a multi-layered graph, where each layer corresponds to a subset of nodes with progressively decreasing density and connectivity as the layers ascend. Our proposal aligns with established practices from prior studies [Malkov and Yashunin, 2020], adopting a multi-layered structure, approximate kNN search, and graph connections.

Algorithm 1 Insert(v)

Input: A vector to be added, v;
Output: The updated graph index, $\mathcal{G}$;

1: Let l_{random} be the assigned level;
2: $\mathcal{R}' \leftarrow PriorityQueue()$;
3: **for** $l = l_{random}; l \geq 0; l--$ **do**
4: $\quad \mathcal{R}' \leftarrow find\_AkNN(v, m, l, \mathcal{R}')$;
5: $\quad connect(v, \mathcal{R}', l)$; ▷ Record connections of v
6: **return** $\mathcal{G}$;

Algorithm 2 Delete(v, v')

Input: A vector to be removed, v;
A vector to repair connections, v';
Output: The updated graph index, $\mathcal{G}$;

1: $\mathcal{C}.remove(v)$;
2: **for** $l = 0 : l_{max} - 1$ **do**
3: $\quad$ **if** $\mathcal{M}(l, v) \neq \emptyset$ **then**
4: $\quad\quad \mathcal{M}(l, \cdot).remove(v)$;
5: $\quad\quad$ **for** $v_n \in v.connections$ **do**
6: $\quad\quad\quad$ **if** $\mathcal{M}(l, v_n) \neq \emptyset$ **then**
7: $\quad\quad\quad\quad \mathcal{M}(l, v_n).remove(v)$;
8: $\quad\quad\quad\quad \mathcal{M}(l, v_n).add(v')$;
9: **return** $\mathcal{G}$;

Innovatively, we introduce Algorithms 1 and 2 to enable dynamic insertion and deletion of vectors, respectively.

Insert algorithm details. Algorithm 1 demonstrates the insertion process, where a vector v is added by traversing the graph from a random maximum layer (Line 1) downward to identify the closest entry point. At each layer, v is connected to its m-nearest-neighbors, and these connections are recorded (Lines 3–5). The $find\_AkNN(\cdot)$ and $connect(\cdot)$ functions are consistent with approaches outlined in prior work [Malkov and Yashunin, 2020].

Delete algorithm details. Algorithm 2 presents the pseudo code for the delete operation. Unlike prior studies, the inputs include both the vector to be removed and a vector used to repair connections. First, the vector v is removed from the cache $\mathcal{C}$ (Line 1). Subsequently, all connections involving v are deleted (Lines 4, 7), and these edges are repaired by establishing new connections with v' (Line 8).

Benefit-Driven Cache Replacement

In web search scenarios, if a new query matches cached historical results, the result is immediately returned to the user. Therefore, the objective of cache designs (e.g., LRU, LFU, FIFO) is to maximize the cache hit ratio. When the cache reaches budget, LRU evicts the least recently used result, LFU removes the least frequently used result, and FIFO evicts the oldest result to make room. However, in the context of kNN search, maximizing the cache hit ratio does not necessarily minimize the overall computation cost, as each cache miss incurs a distinct and often significant computational overhead. Existing cache replacement policies fail to effectively quantify the benefit of a completed kNN query. Furthermore, the importance of a vector may inherently depend on its characteristics and should decay over time.

Motivated by these limitations, we propose a benefit-driven approach to identify which queried vectors should be prioritized for caching.

Benefit model. A benefit model is proposed to evaluate the caching potential of vectors for future queries, which considers three key factors: (i) query frequency (F), (ii) processing cost (E), which refers to the distance computation count, and (iii) recency of the object vector (R). The normalization factors $max(F)$, $max(E)$, and $max(T)$ are computed for global normalization across the dataset. The benefit value of a vector v is computed using Equation 1, where $F'(v)$, $R'(v)$, and $E'(v)$ represent the normalized query frequency, recency (i.e., inverted time decay), and computation cost, respectively. Parameters α, β, and γ are adjustable weights that reflect the relative importance of each metric, subject to the constraint $\alpha + \beta + \gamma = 1$. The values of $F'(v)$, $R'(v)$, and $E'(v)$ are determined using Equation 2, where $T(v)$ denotes the time elapsed since the last access of v.

$$benefit(v) = \alpha \times F'(v) + \beta \times E'(v) + \gamma \times R'(v) \quad (1)$$

$$F'(v) = \frac{F(v)}{\max(F)}, E'(v) = \frac{E(v)}{\max(E)}, R'(v) = 1 - \frac{T(v)}{\max(T)}, \quad (2)$$

Compared to the replacement methods discussed earlier, our benefit model offers distinct advantages. While frequency prioritizes frequently accessed queries and recency emphasizes recently accessed ones, computation cost ensures that queries with high computational overhead are prioritized to minimize recomputation. By integrating these factors, the model effectively ranks cached vectors, balancing the trade-offs between frequency, recency, and computational cost.

Benefit-Driven cache algorithm. Algorithm 3 outlines the pseudo-code for the benefit-driven cache replacement algorithm. A cache update is triggered if the cache size exceeds the budget B and the ratio d_g/d_k is greater than or equal to the specified update factor ϵ, where d_g and d_k represent the maximum distances of the approximate results $\mathcal{R}'$ and exact results $\mathcal{R}$ to the query, respectively. When the condition $d_g/d_k \geq \epsilon$ is satisfied, the cache is updated by adding vector v to $\mathcal{C}$ and inserting v into the graph $\mathcal{G}$ (Line 4). If a cache update occurs, the cached vector v with the minimal benefit is identified (Line 6). This vector is then removed from the cache, and all its connections in $\mathcal{G}$ are deleted (Line 7).

4.2 Exact Search with Distance Guidance

For exact search, the distance d_g (cf. Section 4.1) is utilized to prune unqualified objects early on. Specifically, we use the VP-Tree as the underlying index structure for exact kNN search due to its superior efficiency and minimal memory requirements compared to ball-trees [Rakotondrasoa *et al.*, 2023; Kumar *et al.*, 2008]. Notably, our approach is compatible with other tree-based solutions as well. As illustrated in Fig. 2, each node in the VP-Tree is represented as $\mathcal{N} = \{pt(left), v_p, \mu, I, pt(right)\}$. Here, v_p denotes the vantage point, I represents the vector items stored in a leaf node, and $pt()$ serves as a pointer to child nodes. For non-leaf nodes, I remains empty. Starting at the root node $\mathcal{N}_r$,

Algorithm 3 BDC($\mathcal{R}, \mathcal{R}', \mathcal{G}, B, \epsilon$)

Input: The exact / approximate kNN results, $\mathcal{R}$ and $\mathcal{R}'$;
The cache graph index, $\mathcal{G}$;
Parameter: The cache budget, B;
The cache update factor, ϵ;
Output: The updated cache graph index, $\mathcal{G}$;

1: $d_g = \mathcal{R}'.top(); d_k = \mathcal{R}.top();$
2: **if** $d_g/d_k \geq \epsilon$ **then**
3: **for** $v' \in \mathcal{R}$ **do**
4: $\mathcal{G}.Insert(v');$
5: **if** $|\mathcal{C}| > B$ **then**
6: $v \leftarrow$ a cached vector with the minimal benefit;
7: $\mathcal{G}.Delete(v, v');$
8: **return** $\mathcal{G}$;

which indexes all data, each node $\mathcal{N}$ selects a vantage object v_p from the data indexed at that node. The data are then split into two partitions: the medium distance μ to v_p is calculated for all points under $\mathcal{N}$. Objects with a distance to v_p less than μ are assigned to the left subtree, while those with a distance greater than or equal to μ are placed in the right subtree. To evaluate a kNN query, the VP-Tree is traversed recursively following the same partitioning principle.

Algorithm details. Algorithm 4 provides the pseudo code for kNN search using the VP-Tree with a best-first strategy (BF-kNN). The inputs to the algorithm include the starting node of the VP-Tree $\mathcal{N}$, a query vector v_q, and a cache-guided distance d_g. A priority queue, denoted as $\mathcal{R}$, is used to store the exact kNN results in descending order of their distances. The kNN search follows a recursive traversal process that begins at the root of the tree and progressively updates the result set $\mathcal{R}$. Upon reaching a bucket (i.e., a leaf node), the elements within the bucket are searched sequentially (Lines 3–11). The cache-guided distance d_g is utilized to prune unqualified candidates early in the search process (Line 9). In a metric space with the distance $dist(v, v_q)$, the triangle inequality is employed to reduce the search space. Specifically, only the left subtree is visited if $dist(v, v_q) < \mathcal{N}.\mu - \min(d_k, d_g)$. Similarly, only the right subtree is explored if $dist(v, v_q) > \mathcal{N}.\mu + \min(d_k, d_g)$. In cases where $\mathcal{N}.\mu - \min(d_k, d_g) \leq dist(v, v_q) \leq \mathcal{N}.\mu + \min(d_k, d_g)$, both subtrees must be visited. The high-level concept of our cache-guided best-first strategy in the VP-Tree is as follows: (i) Cache-guided: Instead of relying solely on the maximum distance among the current kNN results, we incorporate $\min(d_k, d_g)$ to guide the search, enabling early-stage pruning of the search space. (ii) Best-first: If $dist(v, v_q) < \mathcal{N}.\mu$, the left subtree is prioritized for traversal; otherwise, the right subtree is explored first (Lines 14–23).

4.3 Overall Solution: Cache-Guided Search

The overall cache-guided search algorithm (CGS) is presented in Algorithm 5. Initially, the cache graph index $\mathcal{G}$ is empty (Line 1). The algorithm starts with a greedy search from a random entry point in the top layer of the cache graph index, progressively moving closer to the query point through each layer until the bottom layer is reached. At this stage, the top k approximate nearest neighbors $\mathcal{R}'$ and the guiding distance d_g are identified (Lines 2-5). In the exact search phase, the guide distance d_g is used to effectively prune the VP-Tree search space (Line 6). Based on the ratio between $\mathcal{R}$ and $\mathcal{R}'$, the cache is dynamically updated (Line 7). Finally, the exact kNN results, $\mathcal{R}$, are returned (Line 8).

Algorithm 4 BF-kNN($v_q, \mathcal{N}, \mathcal{R}, d_g, k$)

Input: A query vector, v_q;
A node of the VP-Tree, $\mathcal{N}$;
The exact kNN results, $\mathcal{R}$;
A cache-guided distance, d_g;
Parameter: The query result quantity, k;
Output: The exact kNN results, $\mathcal{R}$;

1: **if** $\mathcal{N} = null$ **then**
2: **return**;
3: **if** $\mathcal{N}.I \neq \emptyset$ ▷ Leaf Node **then**
4: **for** $v \in \mathcal{N}.I$ **do**
5: **if** $|\mathcal{R}| < k$ **then**
6: $\mathcal{R}.push([v, dist(v, v_q)]$;
7: **else**
8: $d_k \leftarrow \mathcal{R}.top()$;
9: **if** $dist(v, v_q) < \min(d_k, d_g)$ **then**
10: $\mathcal{R}.poll()$;
11: $\mathcal{R}.push([v, dist(v, v_q)])$;
12: $v \leftarrow \mathcal{N}.v_p$;
13: The same as Lines 5–11;
14: **if** $dist(v, v_q) < \mathcal{N}.\mu$ ▷ Best-First **then**
15: **if** $dist(v, v_q) - \mathcal{N}.\mu \leq \min(d_k, d_g)$ **then**
16: BF-kNN($v_q, \mathcal{N}.left, \mathcal{R}, d_g, k$);
17: **if** $\mathcal{N}.\mu - dist(v, v_q) \leq \min(d_k, d_g)$ **then**
18: BF-kNN($v_q, \mathcal{N}.right, \mathcal{R}, d_g, k$);
19: **else**
20: **if** $\mathcal{N}.\mu - dist(v, v_q) \leq \min(d_k, d_g)$ **then**
21: BF-kNN($v_q, \mathcal{N}.right, \mathcal{R}, d_g, k$);
22: **if** $dist(v, v_q) - \mathcal{N}.\mu \leq \min(d_k, d_g)$ **then**
23: BF-kNN($v_q, \mathcal{N}.left, \mathcal{R}, d_g, k$);
24: **return** $\mathcal{R}$;

5 Experiments

5.1 Experimental Setup

Data preparation. We conducted experiments on two real-world datasets to evaluate the performance of our proposal: SIFT [2] and AOL. The SIFT dataset comprises SIFT image descriptors, which are widely used in feature matching applications. The AOL dataset consists of approximately 20 million web queries collected from 650,000 users over three months [Pass *et al.*, 2006]. The data is organized by anonymous user IDs and sequentially arranged. For the AOL dataset, we use the word2vec technique described in [Řehůřek and Sojka, 2010] to convert each query text into 10-dimensional vectors by default. To simulate evolved query distributions in SIFT, we use KMeans clustering [Kanungo *et al.*, 2002] to reorder the queries. We treat the embedding and clustering process as a black-box mechanism, and optimizing the vector embeddings themselves is beyond the scope of our study. Both the cache graph index and the VP-Tree are kept

[2] http://corpus-texmex.irisa.fr

Algorithm 5 CGS$(v_q, \mathcal{N}_r, \mathcal{G}, k, B, \epsilon)$

Input: A query vector, v_q;
The root node of the VP-Tree, $\mathcal{N}_r$;
The cache graph index, $\mathcal{G}$;
Parameter: The query result quantity, k;
The cache budget, B;
The cache update factor, ϵ;
Output: The exact kNN results, $\mathcal{R}$;

1: **Init:** $\mathcal{G} \leftarrow \emptyset$;
2: $\mathcal{R}' \leftarrow PriorityQueue(), \mathcal{R} \leftarrow PriorityQueue()$;
3: **for** $l = l_{max} - 1; l \geq 0; l - -$ **do**
4: $\quad \mathcal{R}' \leftarrow \mathcal{G}.find\_AkNN(v_q, k, l, \mathcal{R}')$;
5: $d_g \leftarrow \mathcal{R}'.top()$;
6: $\mathcal{R} \leftarrow$ BF-kNN$(v_q, \mathcal{N}_r, \mathcal{R}, d_g, k)$;
7: $\mathcal{C} \leftarrow$BDC$(\mathcal{R}, \mathcal{R}', \mathcal{G}, B, \epsilon)$;
8: **return** $\mathcal{R}$;

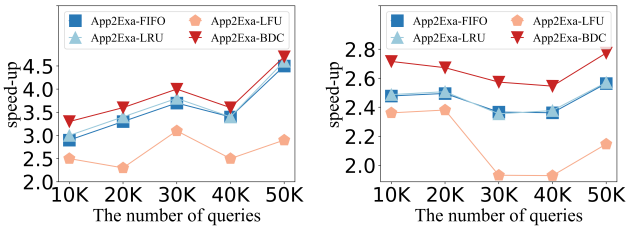
	SIFT	AOL
$\lvert\mathcal{Q}\rvert$	10K, 20K, 30K, 40K, 50K 30K (default)	10K, 20K, 30K, 40K, 50K 30K (default)
$\lvert\mathcal{V}\rvert$	100K–500K 300K (default)	100K–500K 300K (default)
k	5, 10, 15, 20, 25 10 (default)	5, 10, 15, 20, 25 10 (default)
B	0.5% – 2.5%×$\lvert\mathcal{V}\rvert$ 1%×$\lvert\mathcal{Q}\rvert$ (default)	0.5%–2.5%×$\lvert\mathcal{V}\rvert$ 1%×$\lvert\mathcal{Q}\rvert$ (default)
ϵ	1.0, 1.5, 2.0, 2.5, 3.0 2.0 (default)	1.0, 1.5, 2.0, 2.5, 3.0 2.0 (default)

Table 3: Evaluation parameter settings

in memory. By default, we randomly select 30,000 vectors as query vectors $\mathcal{Q}$ and 300,000 vectors as object vectors $\mathcal{V}$.

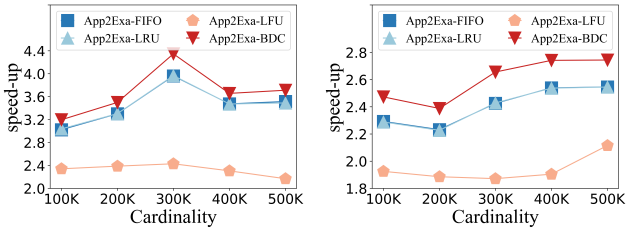
Compared algorithms. We evaluate the performance of the following proposed methods. For approximate kNN search, we employ the cache graph index with four caching strategies: the Least-Recently-Used (LRU) strategy, First-In-First-Out (FIFO) strategy, the Least-Frequently-Used (LFU) strategy, and the Benefit-Driven Cache (BDC) strategy, denoted as App2Exa-LRU, App2Exa-FIFO, App2Exa-LFU, and App2Exa-BDC, respectively. To validate the best-first strategy on the VP-Tree for exact search, we compare the performance of App2Exa-BDC with and without the best-first strategy. The latter is referred to as App2Exa-BDC#. Efficiency is evaluated by measuring the speed-up, computed as $\frac{t_0}{t}$, where t_0 is the query time of the VP-Tree without caching, and t represents the query time when specific caching strategy. For instance, if an App2Exa based solution takes 1 second to process a query while the original VP-Tree without caching takes 10 seconds, the resulting speed-up is 10.

Parameter settings. The evaluation parameter settings are listed in Table 3. For the adjustable parameters α, β and γ in Equation 1, the default values are set to be equal (e.g., 1/3). All the methods are implemented in Java and evaluated on Ubuntu equipped with 2 Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz and 128GB memory, utilizing a single thread for execution. Unless otherwise specified, the reported experimental results represent the averages of 20 independent trials.



(a) SIFT (b) AOL

Figure 3: Effect of the number of queries



(a) SIFT (b) AOL

Figure 4: Effect of the cardinality

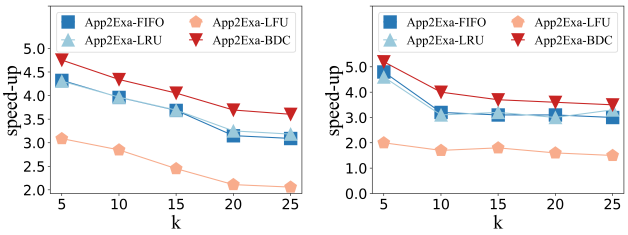
Note that when we vary one parameter, the others are set to default values.

5.2 Performance Evaluation

Effect of the number of queries. First, we evaluate the speed-up performance of the proposed methods by varying the number of queries from 10,000 to 50,000. Fig. 3 illustrates the speed-up performance across different settings. Using our App2Exa techniques, the exact VP-Tree search achieves a speed-up of at least $2x$ for SIFT dataset, and $1.5x$ for AOL dataset. It is observed that App2Exa-BDC consistently achieves the best performance among all methods for all settings, which accelerates the exact search on VP-Tree by at least $3x$ for SIFT, and $2.5x$ for AOL. The similar performance of FIFO and LRU stems from the fact that both rely solely on recency for cache replacement, and thus behave similarly in our scenarios. These experimental results demonstrate the superiority of our proposal.

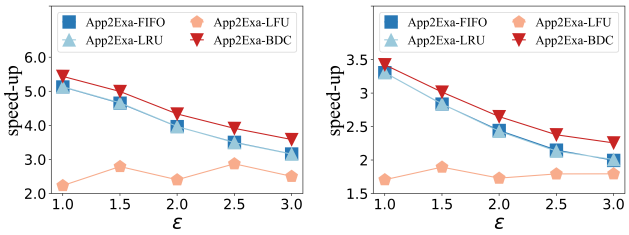
Effect of the database size. Fig. 4 presents the speed-up performance of the proposed methods when varying the data cardinality from 100,000 to 500,000. In SIFT dataset, all methods achieve the best performance when the cardinality is 300,000. We can observe that the App2Exa approach with our BDC cache replacement policy, consistently outperforms all other methods, significantly accelerating the exact computation by a factor of $3.2x - 4.4x$ in SIFT dataset, demonstrating the capability of our proposal in handling large data sets.

Effect of the k. We examine the impact of the k — the results are shown in Fig. 5. Intuitively, a larger k results in more nearest neighbors, in which more vectors are required to be computed. Additionally, as k increases, the distance between the query and its k-th nearest neighbor becomes larger.



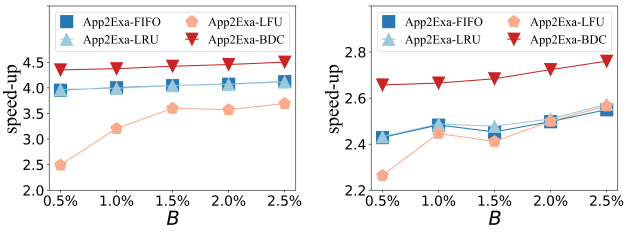
(a) SIFT (b) AOL

Figure 5: Effect of k



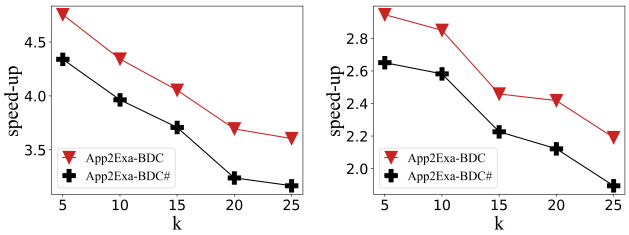
(a) SIFT (b) AOL

Figure 7: Effect of ϵ



(a) SIFT (b) AOL

Figure 6: Effect of B



(a) SIFT (b) AOL

Figure 8: Evaluation on the best-first strategy

This, in turn, causes the VP-Tree to have a wider initial search range for exact k-NN search, thereby requiring more computational effort. All methods achieves their best performance at $k = 5$, with the speed-up stabilizing for $k > 20$. As k increases, App2Exa-BDC still provides significant speed-up by at least $3.5x$ for SIFT dataset, and $3x$ for AOL dataset, demonstrating its scalability across various application scenarios with different k values.

Effect of the cache budget. Fig. 6 shows the speed-up performance as a function of the cache budget B. Specifically, B ranges from 0.5% to 2.5% of the size of object vectors (i.e., database). An increasing trend of the speed-up ratio is observed across all methods as B increases. Notably, due to the effects of data distribution, the speed-up in the AOL dataset increase at a slower rate compared to SIFT. Among the evaluated methods, App2Exa-BDC, which incorporates an effective cache replacement policy in approximate search phase and employs the best-first strategy in the exact search phase using the VP-Tree, consistently outperforms others across all datasets. Specifically, App2Exa-BDC achieves a speed-up exceeding $4x$ for SIFT and $2.6x$ for AOL, highlighting the effectiveness of our proposed approach.

Effect of the cache update factor. Fig. 7 shows the speed-up performance when varying the cache update factor ϵ (cf. Alg. 3). A larger value of ϵ results in fewer cache replacement. As ϵ increases, the methods with LRU, BDC, and FIFO replacement policies exhibit a declining trend in speed-up performance. In contrast, this trend is less pronounced for App2Exa-LFU, due to LFU's tendency to retain frequently accessed vectors without considering time decay. This behavior leads to outdated but previously popular vectors remaining in the cache. Among all methods, our benefit-driven cache (BDC) achieves the highest speed-up performance, which demonstrates the superior effectiveness of our caching replacement policy in adapting to dynamic query distributions.

Evaluation on the best-first strategy. Fig. 8 shows the speed-up performance of best-first strategy that employed in the VP-Tree index. A performance gap is observed between the App2Exa-BDC and App2Exa-BDC# across all datasets. Specifically, the best-first strategy used in the VP-Tree can further shorten the query latency by 8%-12% in SIFT, and 9%-13% in AOL, demonstrating the effectiveness of this strategy in enhancing query efficiency.

6 Conclusion

We proposed App2Exa, a cache-guided approach that bridges approximate and exact kNN searches to enhance efficiency. App2Exa combines an incremental cache graph for fast approximate retrieval with an enhanced VP-Tree for exact search, addressing the challenge of dynamic query distributions in low-to-medium dimensional spaces. Its benefit-driven caching mechanism ensures optimal cache utilization by prioritizing vectors based on query frequency, recency, and computational cost. Experiments on real-world datasets highlight the scalability and robustness of App2Exa, achieving up to a $3x$ speed-up over traditional exact kNN methods.

Acknowledgments

This paper was supported by the National Key R&D Program of China 2024YFE0111800, NSFC 62032001, and the Science and Technology Development Fund Macau SAR (0003/2023/RIC, 0052/2023/RIA1, 0031/2022/A, 001/2024/SKL for SKL-IOTSC).

References

[Babenko and Lempitsky, 2016] Artem Babenko and Victor S. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *CVPR*, pages 2055–2063. IEEE Computer Society, 2016.

[Bae *et al.*, 2009] Wan D. Bae, Shayma Alkobaisi, Seon Ho Kim, Sada Narayanappa, and Cyrus Shahabi. Web data retrieval: solving spatial range queries using k-nearest neighbor searches. *GeoInformatica*, 13(4):483–514, 2009.

[Baeza-Yates *et al.*, 2007] Ricardo A. Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *SIGIR*, pages 183–190. ACM, 2007.

[Bentley, 1975] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[Beyer *et al.*, 1999] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Database Theory - ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer, 1999.

[Bozkaya and Özsoyoglu, 1999] Tolga Bozkaya and Z. Meral Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Syst.*, 24(3):361–404, 1999.

[Cambazoglu *et al.*, 2010] Berkant Barla Cambazoglu, Flavio Paiva Junqueira, Vassilis Plachouras, Scott A. Banachowski, Baoqiu Cui, Swee Lim, and Bill Bridge. A refreshing perspective of search engine caching. In *WWW*, pages 181–190. ACM, 2010.

[Dong *et al.*, 2011] Wei Dong, Moses Charikar, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, pages 577–586. ACM, 2011.

[Frieder *et al.*, 2024] Ophir Frieder, Ida Mele, Cristina Ioana Muntean, Franco Maria Nardini, Raffaele Perego, and Nicola Tonellotto. Caching historical embeddings in conversational search. *ACM Trans. Web*, 18(4):42:1–42:19, 2024.

[Fu *et al.*, 2019] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, 2019.

[Gan and Suel, 2009] Qingqing Gan and Torsten Suel. Improved techniques for result caching in web search engines. In *WWW*, pages 431–440. ACM, 2009.

[Guttman, 1984] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM Press, 1984.

[Halder *et al.*, 2024] Rajib Kumar Halder, Mohammed Nasir Uddin, Ashraf Uddin, Sunil Aryal, and Ansam Khraisat. Enhancing k-nearest neighbor algorithm: a comprehensive review and performance analysis of modifications. *J. Big Data*, 11(1):113, 2024.

[Jelenkovic and Radovanovic, 2004] Predrag R. Jelenkovic and Ana Radovanovic. Least-recently-used caching with dependent requests. *Theor. Comput. Sci.*, 326(1-3):293–327, 2004.

[Kanungo *et al.*, 2002] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, 2002.

[Kumar *et al.*, 2008] Neeraj Kumar, Li Zhang, and Shree K. Nayar. What is a good nearest neighbors algorithm for finding similar patches in images? In *ECCV*, volume 5303 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2008.

[Lampropoulos *et al.*, 2023] Konstantinos Lampropoulos, Fatemeh Zardbani, Nikos Mamoulis, and Panagiotis Karras. Adaptive indexing in high-dimensional metric spaces. *Proc. VLDB Endow.*, 16(10):2525–2537, 2023.

[Lee *et al.*, 2001] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Computers*, 50(12):1352–1361, 2001.

[Li *et al.*, 2022] Ke Li, Lisi Chen, Shuo Shang, Haiyan Wang, Yang Liu, Panos Kalnis, and Bin Yao. Towards controlling the transmission of diseases: Continuous exposure discovery over massive-scale moving objects. In Luc De Raedt, editor, *IJCAI*, pages 3891–3897. ijcai.org, 2022.

[Li *et al.*, 2023] Ke Li, Hongyu Wang, Ziwen Chen, and Lisi Chen. Relaxed group pattern detection over massive-scale trajectories. *Future Gener. Comput. Syst.*, 144:131–139, 2023.

[Liu *et al.*, 2021] Wanqi Liu, Hanchen Wang, Ying Zhang, Wei Wang, Lu Qin, and Xuemin Lin. EI-LSH: an early-termination driven I/O efficient incremental c-approximate nearest neighbor search. *VLDB J.*, 30(2):215–235, 2021.

[Long and Suel, 2006] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. *World Wide Web*, 9(4):369–395, 2006.

[Malkov and Yashunin, 2020] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

[Markatos, 2001] Evangelos P. Markatos. On caching search engine query results. *Comput. Commun.*, 24(2):137–143, 2001.

[Ozcan *et al.*, 2012] Rifat Ozcan, Ismail Sengör Altingövde, Berkant Barla Cambazoglu, Flavio Paiva Junqueira, and Özgür Ulusoy. A five-level static cache architecture for web search engines. *Inf. Process. Manag.*, 48(5):828–840, 2012.

[Pass *et al.*, 2006] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems*, volume 152 of *ACM International Conference Proceeding Series*, page 1. ACM, 2006.

[Peng *et al.*, 2022] Yun Peng, Byron Choi, Tsz Nam Chan, and Jianliang Xu. LAN: learning-based approximate k-nearest neighbor search in graph databases. In *ICDE*, pages 2508–2521. IEEE, 2022.

[Rakotondrasoa *et al.*, 2023] Hanitriniala Malalatiana Rakotondrasoa, Martin Bucher, and Ilya Sinayskiy. Quantitative comparison of nearest neighbor search algorithms. *CoRR*, abs/2307.05235, 2023.

[Ram and Sinha, 2019] Parikshit Ram and Kaushik Sinha. Revisiting kd-tree for nearest neighbor search. In *SIGKDD*, pages 1378–1388. ACM, 2019.

[Řehůřek and Sojka, 2010] Radim Řehůřek and Petr Sojka. Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50. ELRA, 2010.

[Yianilos, 1993] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM/SIGACT-SIAM*, pages 311–321. ACM/SIAM, 1993.

[Zhao *et al.*, 2023] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proc. VLDB Endow.*, 16(8):1979–1991, 2023.

[Zheng *et al.*, 2022] Bolong Zheng, Xi Zhao, Lianggui Weng, Quoc Viet Hung Nguyen, Hang Liu, and Christian S. Jensen. PM-LSH: a fast and accurate in-memory framework for high-dimensional approximate NN and closest pair search. *VLDB J.*, 31(6):1339–1363, 2022.

[Zulfa *et al.*, 2020] Mulki Indana Zulfa, Rudy Hartanto, and Adhistya Erna Permanasari. Caching strategy for web application - a systematic literature review. *Int. J. Web Inf. Syst.*, 16(5):545–569, 2020.