Batching for Locality: Cache-Efficient Dynamic kNN Joins over High-Dimensional Data

Guangjian Zhang
University of New South Wales
guangjian.zhang@student.unsw.edu.au

Zhengyi Yang
University of New South Wales
zhengyi.yang@unsw.edu.au

Xiaoyang Wang
University of New South Wales
xiaoyang.wang1@unsw.edu.au

Wenjie Zhang
University of New South Wales
wenjie.zhang@unsw.edu.au

Xuemin Lin
Shanghai Jiao Tong University
xuemin.lin@sjtu.edu.cn

ABSTRACT

The k nearest neighbors (kNN) join is a crucial operation in data science, retrieving for each point in a query set its kNN from a reference data set. This operation is foundational in domains like vector database and machine learning.In response to the exponentially growing dimensionality and the highly dynamic nature of data driven by the rapid advancement of the information industry, various methods have been developed for dynamic kNN Join on high-dimensional data. We propose a new cluster-based batch strategy to address the problem of dynamic kNN Join. This work is the first to accelerate dynamic kNN Join by leveraging cache locality, effectively keeping data in high-speed storage regions and thereby reducing the processing overhead. The new strategy provided in this paper is supported by rigorous theoretical analysis. Across 8 real-world datasets, our approach demonstrates significant efficiency, outperforming state-of-the-art methods by a factor up to 432%.

PVLDB Reference Format:
Guangjian Zhang, Zhengyi Yang, Xiaoyang Wang, Wenjie Zhang, and Xuemin Lin. Batching for Locality: Cache-Efficient Dynamic kNN Joins over High-Dimensional Data. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zgjgh/High-dimensiona_kNN_by_batch.

1 INTRODUCTION

The k-Nearest Neighbors Join (kNNJ) is a fundamental operation in data science that formalizes the task of retrieving the k nearest neighbors from a reference dataset I for every point within a given query dataset U. Covering all the data points in U rather than only exploring the neighbor relation with I of an individual point, kNN Join establishes a comprehensive mapping of proximity relationships between two datasets. This global perspective is of significant importance in the big data industry, particularly for application domains that necessitate exhaustive coverage for all potential similarity searches. kNN Join demonstrates broad utility across diverse domains. In e-commerce platforms and recommender systems, a common approach is to embed both users and items in a vector space and perform a kNN Join between the user-vector set and the item-vector set to discover potential targets [4, 39, 93]. In healthcare and biomedicine, it supports disease prediction and diagnosis by analyzing connections between preoperative risk variables and diseases, as well as genomic profile matching [11, 24, 50, 77]. In vector databases, kNN Join now serves as the core query primitive of modern AI infrastructure, enabling efficient search and join over billions of high-dimensional embeddings [67, 68]. Beyond these, kNN Join is applied in machine learning tasks such as classification [12, 23, 25, 98], clustering [38, 48, 73, 94], regression [52], predictive systems [9, 46], and anomaly detection [60, 78, 86]. It also supports fraud detection by surfacing transactions whose embeddings are close to historical fraud cases [5, 6, 59, 75], as well as spatial data analysis [13, 90, 95], outlier detection [30, 65], and broader KDD tasks [13]. Modern applications proposes a new demand for kNN Join algorithms with high dimensionality and data dynamism. In e-commerce and media streaming, firms like Amazon and Netflix utilize high-dimensional vectors, which imposes the curse of dimensionality to the kNN operations. Simultaneously, the demand for dynamic adaptivity is driven by sectors with constantly changing data, including social media platforms like TikTok and Twitter (which sees over 300,000 tweets per minute [79]), as well as real-time financial services.

Existing work. To meet these new requirements, a variety of kNN Join approaches have been proposed [8, 13, 14, 21, 22, 28, 33, 58, 76, 90, 95–97, 100]. In order to guarantee high precision in critical scenarios [5, 15, 17, 35, 37, 49, 66], a substantial body of work has focused on exact kNN Join. Representative systems include MuX [13], Gorder [90], iJoin [95], iDistance [96], and the grid-based stream method by Böhm et al. [14], which addressed either high dimensionality or data dynamism, but seldom both simultaneously. Later techniques such as kNNJoin$^+$ [97] and the Δ-Tree [21] extended the scope of existing approaches by explicitly addressing workloads that are both high-dimensional and dynamic. Despite their methodological differences, these exact approaches share the central principle of leveraging indexing structures in combination with dimensionality reduction to ensure efficient and precise kNN Joins.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Motivation. As datasets in modern applications grow rapidly in both volume and dimensionality, the widely adopted point-wise kNN join has become prohibitively computation-intensive. At the same time, advances in hardware—particularly the steady increase in CPU cache capacity and the advent of wide SIMD instructions—have shifted the primary performance bottleneck from computation to data movement, creating new opportunities for cache-efficient designs.

(1) *Intensive Computation of the Point-wise Paradigm.* Real-world data has grown explosively in both volume and dimensionality, imposing unprecedented pressure on kNN Join operations [20, 25, 29]. For a newly arriving query point, the naive complexity scales as $O(D|I|)$. With $|I| = 90{,}000$ and $D = 420$, one query requires scanning about 1.5 GB of data and performing nearly 38 million operations—an overhead unimaginable in early studies but increasingly common today. This renders the computational load of point-wise processing exceedingly heavy. Therefore, in today's setting, deeper exploration of batch-processing strategies for kNN Joins has become particularly important.

(2) *Limited Awareness of Cache Locality.* Older generations of CPUs were equipped with small on-chip caches [36, 41, 42]. As a result, exploiting cache locality provided limited benefits, and most kNN join methods instead focused on pruning techniques to reduce computations and memory accesses. Today, however, the on-chip L1 and L2 caches have expanded substantially[7, 43, 44], thereby amplifying the effectiveness of cache-locality based strategies [32, 62, 71]. At the same time, wide SIMD extensions (e.g., AVX-512) have dramatically lowered arithmetic cost—for instance, the squared-difference-and-accumulation step, the core of Euclidean distance computation in kNN Joins, can now be executed on two 512-bit single-precision vectors in less than 87% [3, 19, 43] of the minimal cycle gap required to move the same data across memory tiers—thereby making data movement the dominant bottleneck. Therefore, cache-locality driven optimization strategies have become a particularly promising avenue for accelerating kNN Joins.

In light of these limitations, we propose a cache-optimized batch-processing strategy for exact kNN Join over dynamic high dimensional data.

Contributions. In response to the aforementioned research gaps, we propose a cluster-based batch framework tailored for dynamic high-dimensional kNN Join. We further establish a cache-aware analytical model that links batch size to execution cost in clock cycles, explicitly capturing latency differences across memory tiers. This model enables effective estimation of the batch size that minimizes overall cost. Our contributions are as follows:

• *A Batch Processing Framework.* Unlike point-wise processing that responds to each newly inserted query individually, our approach allows the accumulation of a substantial number of queries. These queries are then clustered by spatial locality into batches, within which we perform a collective kNN Search strategy proposed in this work. This design enables shared pruning across queries in the same batch. Furthermore, this framework shortens the revisit interval of the same data items, effectively transforming spatial locality into cache locality and enabling data to naturally reside in high-speed caches. As a result, the framework effectively accelerates kNN Joins over high-dimensional dynamic data

• *Cache-Aware Cost Model.* Building upon this framework, we develop a theoretical cost model that mathematically captures the relationship between batch size and execution cost, explicitly accounting for memory hierarchy latencies. Enabling the derivation of optimal batch sizes under any given hardware and data parameters, this model further drives the batch processing framework to its full potential, ensuring that cache utilization approaches the theoretical optimum and overall execution time is minimized.

• *Extensive Experimental Validation.* We evaluate on eight real-world datasets against the state-of-the-art method following the paradigm of point-wise update. Our framework achieves substantial speedups (up to 432%×), and the model's predictions of optimal batch size closely match empirical results.

Applications. Our batch framework for dynamic high-dimensional kNN Join is well-suited to scenarios that demand exact results while tolerating delayed responses, which enables the accumulation of new arrivals into a collection for subsequent batch processing.

Data-Driven Due Diligence. Financial due diligence for illiquid assets requires exact kNN Join against large historical datasets to identify peer projects [27, 59, 72]. This task demands extremely high accuracy to mitigate potential financial risks, but it is offline and latency-tolerant, making it a natural fit for batch processing.

Clinical Decision Support. In healthcare, new patient cases are matched to historical records via kNN Join to support clinical decisions [64, 91]. Since errors could endanger patient safety, exact analysis is mandatory; however, these reports are generated offline, so delayed responses are acceptable.

Biodiversity Monitoring. Bioacoustic monitoring converts soundscapes into high-dimensional vectors to detect ecological patterns [10, 26]. Given the irreversible risks of misjudgment, this task requires extremely high accuracy, yet ecosystem changes unfold slowly, making the workflow latency-tolerant and well-suited for batch processing.

Fingerprint Identification. Forensic fingerprint matching uses kNN Join against national databases [2, 18]. Exact results are required to avoid wrongful convictions, while investigative timelines tolerate hours or days of delay, making this domain ideal for batch exact kNN Join.

High-Energy Physics. Event analysis in particle physics compares collision data with massive simulated datasets [31, 51]. Exactness is vital to avoid false scientific conclusions, and the data arrives in large indivisible batches, making batch-oriented and latency-tolerant processing the only viable solution.

2 BACKGROUND

This section first introduces several important definitions. We then present our problem statement and highlight the key characteristics of PCA and the Δ-Tree, the fundamental index structure of our batch framework for dynamic kNN Join on high-dimensional data.

Definition 1. **kNN Search**. In a D-dimensional space $\mathcal{D}$ and a distance space $\mathcal{L}$ with a distance metric $dist : \mathbb{R}^D \times \mathbb{R}^D \to \mathcal{L}$, given a query point $u \in \mathcal{D}$, an reference set $I \subseteq \mathcal{D}$ and a positive integer $k \leq |I|$, a kNN Search(kNNS) returns a set of k closest neighbors with metric $dist$ in I for u, that is, $kNN(u, I) = \{i | \forall i' \in I - kNN(u, I), dist(u, i') \geq dist(u, i)\}$, $|kNN(u, I)| = k$.

Definition 2. **kNN Join**. Given a dataset $\mathcal{D}$ with distance metric $dist$, a query set $U \subseteq \mathcal{D}$, a reference set $I \subseteq \mathcal{D}$, and a positive integer $k \leq |I|$, kNN Join(kNNJ) returns a join table that is a set including every query point in U attached with its kNN Search result in I:

$$kNN_{\bowtie}(U, I) = \left\{ \left(u,\ kNN(u, I)\right) \ \middle|\ u \in U \right\}. \tag{1}$$

Definition 3. **Dynamic kNN Join**. Dynamic kNN Join (DkNNJ) is the task of maintaining the kNN Join table $kNN_{\bowtie}(U, I)$ between a dynamically updated query set U and a fixed reference set I. Specifically, given the current join table $kNN_{\bowtie}(U, I)$ and an update consisting of either an insertion or a deletion of a collection W of query points in U, the goal is to compute the new join table $kNN_{\bowtie}(U - W, I)$ or $kNN_{\bowtie}(U + W, I)$.

In this work, we use the classic *Euclidean Distance* as our distance metric, which is the L2-Norm [54] of the difference of the two involved vectors.

In dynamic kNN Join, insertions constitute the dominant performance bottleneck, as new queries necessitate costly kNN search against the reference set. In contrast, deletions are relatively trivial, requiring only the removal of existing tuples from $kNN_{\bowtie}(U, I)$. We therefore restrict our attention to the insertion case in this work.

Problem Statement. Given a query set U, a fixed reference set I, a collection W of newly inserted query points, and the current join table $kNN_{\bowtie}(U, I)$, we aim to compute the updated join table $kNN_{\bowtie}(U + W, I)$.

2.1 Principle component analysis

Principle component analysis[1, 16, 85, 89](PCA) is a classic method for dimension reduction. It yields a transformation matrix M that projects vectors from their original coordinate system onto a new orthonormal basis, where the dimensions are organized in descending order of the variance they capture from the projected vectors. The rows of this matrix are known as the principal components. Because the aimed dimensions of PCA are ranked in decreasing sequence of the projected vector's variance on them, a front small section of dimensions in M can contain a large proportion of the distance information of the data in the full-dimensional space. In this paper, We use $PC(i)$ to denote the i-th principle component, and use $\cup_{i=0}^{N} PC(i)$ to represent the space dominated by the first N principle components (N-dimensional PC space). For example, $u_{\cup_{k=1}^{N} PC(i)}$ represents the projection of a data point u in the space dominated by the first N principle components. Besides, we use $\sigma(PC(i))$ to represent the variance of the data corresponding to the i-th principle component. PCA exhibits a property beneficial to simplifying the distance comparison operation that:

Lemma 1. *Decreasing in the dimensions of principle component spaces brings a no-increasing effect on the distance between two data points (a and b), which can be written as:*

$$If\ d_1 \leq d_2,\ \ then,\ dist_{\cup_{i=1}^{d_1} PC(i)}(a, b) \leq dist_{\cup_{i=1}^{d_2} PC(i)}(a, b)$$

The proof for lemma 1 can be found in [82].

2.2 Δ-Tree

We adopt the Δ-Tree [21] as the foundational indexing model for our batch processing strategy. To the best of our knowledge, the Δ-Tree remains the state-of-the-art solution for the exact dynamic kNN Join problem on high-dimensional data. Although a subsequent method named kNNJoin$^+$ [97] was proposed, it relies on the idistance [96] search kernel. However, idistance has been shown to be significantly less efficient than the Δ-Tree [21] as dimensionality increases, primarily due to its lack of an effective dimensionality reduction mechanism. Given its proven effectiveness for dynamic high-dimensional kNN Join through hierarchical clustering–based pruning and dimensionality-reduction strategies, we adopt the Δ-Tree in our batch framework to make use of these advantages. The properties of the non-leaf nodes and leaf nodes of the Δ-Tree will be described below.

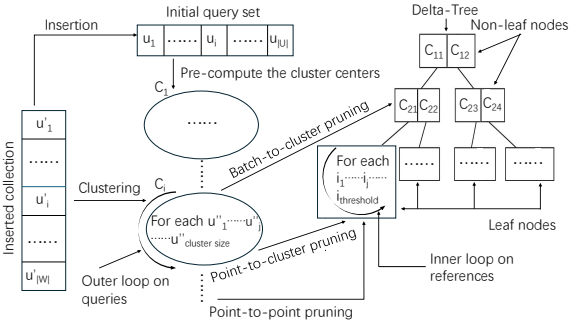


Figure 1: Batch stratrgy for Dynamic kNN Join

Non-leaf node on Δ-Tree A non-leaf node on the l-th level of Δ-Tree accommodate its assigned reference points in the $d(l)$-dimensional PC space, where $d(l)$ is given by

$$d(l) = min\{d | \frac{\sum_{i=1}^{d} \sigma(PC(i))}{\sum_{i=1}^{D} \sigma(PC(i))} \geq \frac{l}{\Lambda}\} \tag{2}$$

, where D is the full dimensionality and Λ is the height of the Δ-Tree. For a non-leaf node on the l-th level, its reference points are divided into f clusters by *k-means* in the PC space of $\cup_{i=1}^{d(l)} PC(i)$, and the pointer pointing to the child node and the center and radius in the corresponding PC space of each of the clusters are stored. For a cluster in a non-leaf node, if the number of its reference points is larger than *thres*, then it will evolve into a non-leaf node on the next level, otherwise its corresponding child node would be a leaf node.

Leaf node on Δ-Tree A leaf-node of a Δ-Tree directly reserves the assigned reference points in the full-dimensional space.

3 CLUSTER-BASED BATCH KNN JOIN

In this section, we present our batch strategy based Δ−Tree for dynamic kNN Join over high-dimensional data.

3.1 Limits of Δ-Tree for new query collections

In dynamic settings where new queries continuously arrive, the Δ-Tree is constrained to processing each query independently, which prevents reuse of data across queries and leads to excessive I/O cost. While spatially adjacent queries often access similar nodes and reference points, their arrivals in the data stream may be separated by many other queries. Consequently, the relevant cache lines

Algorithm 1: Batch kNN-Join Maintenance

Input: A new query collection W, root *root* of Δ−Tree, a length-N list BCH of empty batches
Output: Updated join table $\text{kNN}_{\bowtie}(U + W, I)$

1 **foreach** $newq \in W$ **do**
2 $bch\_in = \arg\min\{dist_{\cup_{i=1}^{d(\Lambda-1)} PC(i)}(bch.anch, newq)\}$
3 bch_in.push($newq$)
4 **foreach** $bch \in BCH$ **do**
5 Collective_KNN_Search($root$, bch)
6 **foreach** $newq \in W$ **do**
7 Insert $(newq, kNN(newq, I))$ into the join table $kNN_{\bowtie}$

are evicted before reuse, forcing repeated DRAM transfers—a particularly expensive process in high dimensions. To overcome this limitation, we cluster new queries into batches, each serving as a query unit on the Δ-Tree to perform collective kNN search. This design enables cache sharing of data access and substantially reduces transfer cost. Moreover, adjacent queries within the same batch can share pruning on the Δ-Tree, further improving processing efficiency.

3.2 Cluster-based Batch strategy for kNN Join

Our approach proceeds in two stages. In the first stage, we precompute the anchors (i.e., cluster centers) of all batches from the initial query set. In the second stage, we (i) assign each newly arriving query to its closest batch and (ii) perform collective kNN search for each batch on the Δ-Tree. This design enables efficient reuse of cached data and reduces redundant DRAM transfers across queries. The whole procedure is illustrated in figure 1

Pre-computation for anchors of the batches.

Most clustering methods (e.g., k-means [40], DBSCAN [74], GMM [99], and hierarchical clustering [47, 70]) incur high computational cost, which could offset the efficiency gains from batch processing, making them impractical for repeatedly clustering every new query collection. To avoid this overhead, we introduce a preprocessing stage that derives anchor points from the initial query set.

Specifically, we run k-means once on the initial queries to produce N clusters, and treat their centers $o_1, o_2, \ldots, o_N$ as anchors. Each incoming query is then assigned to the batch corresponding to its nearest anchor, requiring only lightweight distance comparisons instead of full clustering from scratch. Since preprocessing is performed only once, we execute k-means in the full-dimensional space to maximize accuracy. This design leverages the stability of data distribution, as new queries generally preserve a morphology comparable to that of the initial set, and our experiments later confirm that the framework built on this assumption achieves efficient performance.

kNN Join by batches. Algorithm 1 illustrates the maintenance of the kNN Join table against a collection W of new query points. The process begins by dividing the new queries into N batches (Lines 2–3). Following the principle of grouping spatially similar points, each query is assigned to the batch whose anchor is nearest, and the assignment is carried out in a low-dimensional PCA space to keep the overhead low. Specifically, the projection

Algorithm 2: Collective kNN Search

Input: A batch bch of new queries, root $root$ of Δ−Tree
Output: $kNN(newq, I)$ of each new query $newq$ in bch

1 Initialize a priority queue Q_NLN for the non-leaf nodes on the Δ−Tree; Initialize $GenB \leftarrow \infty$
2 **foreach** $newq \in bch$ **do**
3 Initialize $newq.Cand\_list$; Initialize $newq.B \leftarrow \infty$
4 Q_NLN.push($(root, \infty)$)
5 **while** Q_NLN *is not empty* **do**
6 $nln\_clst, min\_dis = \arg\min_{tuple \in Q\_NLN}(tuple[1])$
7 **if** $min\_dis < GenB$ **then**
8 Q_NLN.pop((nln_clst, min_dis))
9 **foreach** $clust \in nln\_clst$ **do**
10 $d = dist_{PC}(bch.anch, ctr(clust)) - r_{PC}(clust)$
11 **if** $d < GenB$ **then**
12 **if** $clust.child$ *is a Non-leaf node* **then**
13 Q_NLN.push($(clust.child, d)$)
14 **else**
15 **foreach** $newq \in bch$ **do**
16 **if** $dist_{PC}(newq, ctr(clust)) - r_{PC}(clust) < newq.B$ **then**
17 **foreach** $ref \in clust.child$ **do**
18 **if** $dist(newq, ref) < newq.B$ **then**
19 $newq.Cand\_list$.(push)(ref)
20 Adjust $newq.Cand\_list$
21 Adjust $newq.B$
22 $GenB \leftarrow bch.r + (newq.B)_{max}$
23 **else**
24 Break

space $\cup_{i=1}^{d(\Lambda-1)} PC(i)$, corresponding to the highest non-leaf level of the Δ-Tree, is used to preserve pruning precision while reducing distance-computation cost.

For each batch, its low-dimensional radius is then computed as the maximum distance from its anchor to the assigned queries. Next, Algorithm2 is invoked to perform collective kNN search, concurrently retrieving the neighbors of all queries in the batch. Finally, the new queries and their kNN results are inserted into the kNN Join table to complete the update. The details of Algorithm 2 will be discussed later.

Collective kNN Search for a batch. Algorithm 2 performs a collective kNN search for a batch of queries that exhibit spatial locality. All queries in the batch are treated as a single entity, which traverses the Δ-Tree using a priority queue of non-leaf nodes. Each popped node provides clusters whose children are either pushed back into the queue (if non-leaf) or expanded to reference points for candidate updates (if leaf). For each query, the distance to the farthest element in its candidate list defines a pruning bound B,

which guides whether newly visited reference points should be admitted.

Pruning proceeds hierarchically. First, a batch-to-cluster pruning is applied between the query batch and clusters of a popped non-leaf node in the PCA subspace corresponding to that Δ-Tree level (Lines10–11). Clusters that survive are either reinserted as non-leaf children (Lines12–13) or subjected to point-to-cluster pruning, which is likewise performed in the corresponding low-dimensional subspace to significantly reduce processing cost (Lines15–16). Finally, clusters that pass these filters undergo point-to-point pruning in the full-dimensional space, updating the candidate lists of all queries (Lines17–21). This iterative process continues until the queue is empty or the distance bounds of all remaining nodes exceed the global pruning bound. At termination, the candidate lists of all queries converge to their exact kNNs.

Correctness of the batch to cluster pruning A policy for pruning between a query batch and a cluster in low-dimensional space is proposed for algorithm 2(line 10,11), with which, shared pruning of the clusters can be performed across all the queries in the batch. The policy is that, if the general pruning bound of a batch which is equal to the sum of its radius and the largest pruning bound of its new query points is no larger than the difference between the distance from the anchor point of the batch to the center of the cluster in the PC space of the level where the cluster is and the radius of the cluster in the corresponding PC space, then, none of the reference point in the cluster should be added into the candidate lists of the query points in the batch.

In mathematic language, the policy can be written as:

$$
\begin{aligned}
&If\ \ GenB \le dist_{\cup_{i=1}^{d(l(C))} PC(i)}(anch, ctr) - r_{\cup_{i=1}^{d(l(C))} PC(i)}(C),\\
&where\ \ GenB = r + B_{max},\ \ then,\ \ \forall ref \in C,\ \ \forall newq \in batch,\\
&ref \notin Candlist(newq)
\end{aligned}
$$

The proof can be performed as follows. For an arbitrary new query point $newq$ in the batch and an arbitrary reference point ref in the cluster C, we have,

$$
\begin{aligned}
&dist(newq, ref) \ge dist_{\cup_{i=1}^{d(l(C))} PC(i)}(newq, ref)\\
\ge &dist_{\cup_{i=1}^{d(l(C))} PC(i)}(newq, ctr) - r_{\cup_{i=1}^{d(l(C))} PC(i)}(C)\\
\ge &dist_{\cup_{i=1}^{d(l(C))} PC(i)}(anch, ctr) - r_{\cup_{i=1}^{d(l(C))} PC(i)} - r_{\cup_{i=1}^{d(l(C))} PC(i)}(C)
\end{aligned} \tag{3}
$$

Because $l(C) \le \Lambda - 1$, it can be deduced that $d(\Lambda - 1) \ge d(l(C))$ based on lemma 1, and the radius r of the batch is computed in the PC space of $\cup_{i=1}^{d(\Lambda-1)} PC(i)$, as a result, r is no smaller than the radius of the batch in the PC space of $\cup_{i=1}^{d(l(C))} PC(i)$ that $r \ge r_{\cup_{i=1}^{d(l(C))} PC(i)}$. Based on the derivation above and the in-equation 3, the following derivation can be made that:

$$
\begin{aligned}
&dist(newq, ref) \ge dist_{\cup_{i=1}^{d(l(C))} PC(i)}(newq, ref)\\
\ge &dist_{\cup_{i=1}^{d(l(C))} PC(i)}(anch, ctr) - r - r_{\cup_{i=1}^{d(l(C))} PC(i)}(C)
\end{aligned} \tag{4}
$$

Combing in-equation 4 and the precondition of theorem that

$$
r + B_{max} \le dist_{\cup_{i=1}^{d(l(C))} PC(i)}(anch, ctr) - r_{\cup_{i=1}^{d(l(C))} PC(i)}(C) \tag{5}
$$

it can be derived that:

$$
\begin{aligned}
&dist(newq, ref) \ge dist_{\cup_{i=1}^{d(l(C))} PC(i)}(newq, ref)\\
\ge &dist_{\cup_{i=1}^{d(l(C))} PC(i)}(anch, ctr) - r - r_{\cup_{i=1}^{d(l(C))} PC(i)}(C)\\
\ge &r + B_{max} - r = B_{max} \ge B(newq)
\end{aligned} \tag{6}
$$

In conclusion, when the general pruning bound of a batch is no larger than the difference between the distance from the anchor point of the batch to the center of a cluster on the Δ-Tree in the corresponding PC space to the level where the cluster is and the radius of the cluster in the PC space, the distance between an arbitrary query point in the batch and an arbitrary reference point in the cluster must be no smaller than the individual pruning bound of the query point, which is equal to the distance from the query point to the furthest reference point in its candidate list, so that the reference points in the cluster can be entirely pruned for the query batch.

3.3 Advantage of batch kNN Join

There are two merits of the batch kNN Join strategy in comparison of the point-wise paradigm. (i) Shared cluster-level pruning. Processing queries as clustered collections lets spatially close queries reuse the same cluster-level pruning decisions on the Δ-Tree. The batch-to-cluster test—executed in the PCA subspace for the corresponding tree level—tend to eliminate the same non-promising clusters for all queries in a collection, reducing repeated pruning work and pushing early cuts higher in the tree. (ii) Leaf-level cache reuse. During the point-to-point stage (Algorithm 2, Lines 17–21), after scanning all references of a hot leaf for one query, our collective search immediately proceeds to its spatially adjacent peers while that leaf remains cache-resident. Because nearby queries often require the same leaf references, they are served directly from cache rather than incurring a fresh DRAM→cache transfer. By contrast, per-query incremental processing interleaves unrelated queries, so the same leaf is likely evicted before a neighboring query arrives, re-triggering DRAM traffic. Putting these together, the collective design compresses reuse distance at both the cluster and leaf levels, thereby cutting memory traffic—the dominant bottleneck once arithmetic is already amortized by pipelining, SIMD, and out-of-order execution.

4 CACHE AWARE THEORETICAL MODEL FOR TIME COST

In the following discussion, a theoretical model to simulate the relationship between the time cost measured in number of clock cycles and the size of the clusters into which the collection of newly arrived query points is partitioned into would be built, based on which optimization methods can be utilized to achieve the optimal batch size giving the minimal time cost. Before discussion, the following estimations are given below for development of the model. We analyze two primitives used throughout in the model for clock cycles cost: Euclidean distance and min-reduction. All costs are in clock cycles and explicitly account for data movement. We assume an x86 CPU with AVX-512. Let an element have size S bytes and let a 512-bit vector (zmm) contain $V = 64/S$ lanes. For a length-D vector, the number of vector chunks is $N = \lceil D/V \rceil$. Memory tiers are

$Loc \in \{L1, L2, L3, Mem\}$ and registers are $L0$. Let $T_{\text{trans}}(\cdot)$ denote the cycles to move data from the given tier(s) into registers. Instruction costs (per issued instruction) are denoted T_{VSUB_τ}, T_{VFMA_τ}, T_{vadd_τ}, T_{VMIN_τ}, T_{perm_τ}, T_{extract_τ}, and (optionally) T_{sqrt_τ} for data type τ.

Estimation 1 (Euclidean distance)**.** *For two D-dimensional vectors with sources $Loc_1, Loc_2 \in \{L1, L2, L3, Mem\}$, the distance cost is*

$$T_{\text{dist}}(D, Loc_1, Loc_2) = T_{\text{trans}}(D, Loc_1, Loc_2) + N \cdot \left(T_{\text{VSUB}_\tau} + T_{\text{VFMA}_\tau}\right) + T_{\text{reduce}}(V) + T^{(\text{opt})}_{\text{sqrt}_\tau} \tag{7}$$

where a tree-style horizontal reduction over V lanes costs $T_{\text{reduce}}(V) \approx \lceil \log_2 V \rceil \cdot \left(T_{\text{perm}_\tau} + T_{\text{vadd}_\tau}\right)$. When comparing squared distances, T_{sqrt} is omitted.

Estimation 2 (Min-reduction)**.** *For the minimum of a D-element array stored at $Loc \in \{L1, L2, L3, Mem\}$,*

$$T_{\text{minz}}(D, Loc) = T_{\text{trans}}(D, Loc) + (N-1)\, T_{\text{VMIN}_\tau} + \lceil \log_2 V \rceil \cdot \left(T_{\text{perm}_\tau} + T_{\text{VMIN}_\tau}\right). \tag{8}$$

Estimation 3 (Data movement (practical upper bound))**.** *Let the cache/memory links $L_k \rightarrow L_{k-1}$ ($k \in \{1, 2, 3, 4\}$, with L_4=Mem, L_0=Regs) have startup latencies τ_k (cycles) and sustainable bandwidths BW_k (bytes/cycle). Moving D elements (each S bytes) from $Loc = L_k$ to registers is*

$$T_{\text{trans}}(D, L_k) \;\approx\; \max\left(\sum_{j=1}^{k} \tau_j,\; \frac{DS}{\min_{1 \le j \le k} BW_j}\right).$$

For two input vectors at $Loc_1 = L_x$ and $Loc_2 = L_y$ (w.l.o.g. $x \le y$), a conservative bound is

$$T_{\text{trans}}(D, Loc_1, Loc_2) \;\le\; T_{\text{trans}}(D, L_x) + T_{\text{trans}}(D, L_y).$$

All micro-architectural details (exact instruction selections, overlap between transfers, and tighter bounds for the two-input case) are deferred to the Appendix A.2.

In our problem formulation, data attributes and hardware parameters are represented symbolically, as their concrete quantitative relationships are implementation-dependent. Consequently, it is infeasible to resolve the given recurrence formulas in estimations 2, 3, 1 for the time cost of Euclidean Distance calculation and that of the minimization of a list into single, closed-form expressions, which are free of the $\max(A, B)$ operation. Therefore, we simplify the formulas into symbols $T_{minz}(D, Loc)$ and $T_{dist}(D, Loc_1, Loc_2)$, which underscores the determining influence of the length and the original location of the involved data on the time cost in this paper. In practice, once the input cluster size is fixed, the starting positions and dimensions of all involved data can be determined and treated as constants. Hence, retaining the notation above rather than substituting their explicit expressions in terms of the cluster size does not affect the derivations.

In our batch strategy for dynamic kNN Join, the overall execution time is determined by four components:(i) the initial clustering of the newly arrived query collection, line 2-3 in algorithm 1, (ii) cluster-to-cluster pruning on the Δ-Tree, line 10-11 in algorithm 2, (iii) point-to-cluster pruning of query points against leaf nodes in the Δ-Tree, line 16 in algorithm 2, and (iv) point-to-point pruning of query points against reference points within the leaf nodes on the Δ−Tree, line 17-18 in algorithm 2. Therefore, the total consumption of clock cycles can be derived in the followings:

$$E_{clk}(N_c) \approx \frac{|W|}{|N_c|}(E(T_{p2p}) + E(T_{c2c}) + E(T_{p2c})) + E(T_{clus}) \tag{9}$$

For the derivation above, we assume that setting the k-means parameter $k = |W|/N_c$ during preprocessing yields, in subsequent updates, batches whose sizes are approximately N_c. In the estimation above, $E(T_{p2p})$, $E(T_{c2c})$, and $E(T_{p2c})$ respectively represent the expected cost clock cycles during the point-to-point, cluster(batch)-to-cluster, and point-to-point pruning phases for each query cluster, and E_{clus} represents the expected cost for the stage of data assignment to different batches.

Table 1: Some notations in the point-to-point pruning cost model.

Symbol	Meaning
ref	A reference point
$E_{\text{rep}}(T_{dist})$	Cycles of a distance computation with a visited ref
$E_{\text{new}}(T_{dist})$	Cycles of a distance computation with a new ref
$P_{\text{rep}}(N_c)$	Probability that a distance computation reuses a ref
N_c	Cluster size (capacity)
$E(N_l)$	Expected value of N_l
LND	A leaf node
$LND(q_k)$	The set of refs visited at leaf nodes by the query q_k
H_k	The bounding sphere for the query point q_k
$V(H_k)$	The volume of H_k
R_c	Radius of the hypersphere enclosing a query cluster
V_c	Volume of the hypersphere enclosing a query cluster

Cost clock cycles for point-to-point pruning As described in lines 17-18 of Algorithm 2, the point-to-point pruning at the leaf nodes of the Δ-Tree consists of two nested loops: the outer loop traverses the data points in the query cluster, while the inner loop scans the reference points stored at the leaf node. Consequently, each distance computation falls into one of two cases: (i) the reference point has already been accessed by at least one preceding query in the cluster, or (ii) the reference point is accessed for the first time.

The expected cost in clock cycles for this process, denoted by $E(T_{p2p})$, can be expressed as:

$$E(T_{p2p}) = E_{\text{rep}}(T_{dist}) \cdot N_c \cdot E(|NLD|) \cdot P_{\text{rep}}(N_c) + E_{\text{new}}(T_{dist}) \cdot N_c \cdot E(|NLD|) \cdot (1 - P_{\text{rep}}(N_c)), \tag{10}$$

where the notations are summarized in Table 1.

In formula 10, an approximation can be made below:

Approximation 1 (Expected cost for distance calculation)**.** *The expected number of clock cycles for distance calculation can be expressed as a weighted combination of the costs of computing distances between query points and reference points across different memory tiers.*

$$E_{rep}(T_{dist}) \approx \sum_{j=1}^{2} (P(ref \in Lj | rep) \cdot T_{dist}(D, L2, Lj)) \tag{11}$$

In the estimation above, $P(ref \in Lj|rep)$, the probability for the reference point ref to be in Lj cache in the condition ref that has

been visited before by preceding query points can be estimated by the follows:

Approximation 2 (Cache residence probability)**.** *When a reference point in a leaf node is revisited, the probability that it resides in cache tier L_j can be estimated by*

$$P(ref \in Lj \mid rep) \approx \frac{A_j}{E(|LND|)}, \qquad j \in \{1, 2\}, \tag{12}$$

where

$$A_j = \min\Big\{ \max\big(E(|LND|) - \sum_{k=1}^{j-1} |Lk|,\ 0\big),\ |Lj| \Big\}, \tag{13}$$

*that $E|LND| \approx thres \cdot D \cdot S$ is the expected value of the data size $|LND|$ of a leaf node that can be approximated by the multiplication of the clustering threshold value **thres** of $\Delta-$Tree, the dimensionality of the data, and the length of one dimension of data.*

Full derivation and justification are provided in the online supplementary material.

*Intuition. Because (i) the inner-loop accesses at a leaf node are separated only by a very short interval—the data movement and computation required for a few low-dimensional distance evaluations (line 17-18 in algorithm 2)—data of reference points at the involved leaf node remains at the top of the cache hierarchy throughout the nested loop once it is accessed for the first time. (ii) Moreover, the number of reference points in a leaf is bounded by the small clustering threshold **thres** to ensure fine-grained pruning. This design keeps the total leaf size well below the combined capacity of L1 and L2 caches. As a result, revisited reference points are almost always served from L1 or L2 rather than being evicted to L3.*

And in approximation 1, the query points are assumed to be predominantly served from the L2 cache. This is justified by two observations. First, query clusters are repeatedly accessed throughout the collective search procedure and thus constitute hot data that naturally remain in cache. Second, the interval between two consecutive times of accesses to the full-dimensional data in the query cluster corresponds to the data visited for the pruning operations performed on the non-leaf nodes of the Δ-Tree and the data of the reference points at a leaf node. The amount of these intermediate data touched can typically exceed the capacity of L1 but remains well within the range of L2. Consequently, the head portion of the query cluster can be regarded as resident in L2. In the occasional case where part of the cluster spills into L3, the additional latency can be partially hidden by the prefetching behavior [45, 53, 63] of the outer loop. Therefore, it is reasonable to model query clusters as residing in L2. Similar to approximation 2, it can be easily deduced that

$$E_{new}(T_{dist}) \approx T_{dist}(D, L2, Mem) \tag{14}$$

where *Mem* represents the main memory. In the approximation above and approximation 1, computation of the item T_{dist} can be guided by estimation 1.

While N_c is given as an input parameter and $E(|NLD|)$ can be obtained through straightforward sampling, the remaining unknown is $P_{rep}(N_c)$—the probability that a single distance computation involves a previously visited reference point. This quantity must be analyzed in order to establish the connection between N_c and the expected cost $E(T_{p2p})$. It is obvious that P_{rep} is a function with N_c. Before building the estimation for P_{rep} given N_c, the following approximation, assumptions, and lemmas are given.

Assumption 1 (Linear relation between volume and cluster capacity.)**.** *The volume V_c of the hypersphere that encloses all the data points in a query cluster is of positive linear relation with the cluster size N_c that:*

$$V_c = \frac{\pi^{\frac{D}{2}} R_c^D}{\Gamma\left(\frac{D}{2} + 1\right)} = kN_c + b \tag{15}$$

where R_c is the radius of the hypersphere that encloses all the query points in the cluster, and $\frac{\pi^{\frac{D}{2}} r_c^D}{\Gamma(\frac{D}{2}+1)}$ is the formula of the volume of a hypersphere [84], and Γ represents the Gmma function $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\,dt$.

Assumption 2. *Given a series of query points of arbitrary length that q_1, q_2,...,q_N in a query cluster ($N \le N_c$), for a reference point ref visited at a leaf node on the $\Delta-$Tree by q_N that $ref \in LND(q_N)$, whether it is in $LND(q_1) \cup LND(q_2) \cup ... \cup LND(q_{N-2})$ or not is independent of whether it is included in $LND(q_{N-1})$. This assumption can be written in mathematic language as what follows:*

$$\begin{aligned} &P(ref \notin LND(q_1) \cup ... \cup LND(q_{N-1}) | ref \in LND(q_N)) \\ =&P(ref \notin LND(q_1) \cup ... \cup LND(q_{N-2}) | ref \in LND(q_N)) \\ &\times P(ref \notin LND(q_{N-1}) | ref \in LND(q_N)) \qquad (16) \\ \Rightarrow&P(ref \notin LND(q_1) \cup ... \cup LND(q_{N-1}) | ref \in LND(q_N)) \\ =&\Pi_{i=1}^{N-1} P(ref \notin LND(q_i) | ref \in LND(q_N)) \qquad (17) \end{aligned}$$

The deduction from equation 16 to equation 17 is based on mathematic induction.

This independence assumption does not strictly hold in all cases, but it is a common approximation in probabilistic analysis and, as shown in our experiments, our model based on this assumption derives a theoretical optimal cluster size that closely approximates the capacity yielding the minimal number of clock cycles in practice.

Approximation 3 (Overlap probability of leaf access)**.** *Consider two distinct query points q_i and q_j. For each query q_k, we define a bounding hypersphere H_k centered at q_k that encloses all reference points accessed by it at the leaf nodes on the $\Delta-$Tree during the collective kNN Search.*

The probability for a reference point ref accessed by q_i at a leaf node on the $\Delta-$Tree that $ref \in LND(q_i)$ to also be accessed by q_j that $P(ref \in LND(q_j) \mid ref \in LND(q_i))$ can be approximated by the ratio of intersection volume to the sphere volume:

$$\begin{aligned} P(ref \in LND(q_j) \mid ref \in LND(q_i)) &\approx \frac{V(H_i \cap H_j)}{V(H_i)} \\ &\approx \frac{V(H_i \cap H_j)}{E(V(H_k))}. \end{aligned} \tag{18}$$

where $E(V(H_k))$ represents the expected value of the volume $V(H_k)$ of the bounding sphere of q_k

Intuition. The search region for each query point forms a hypersphere. The chance that two queries hit the same reference point depends on how much two hyperspheres overlap. Larger overlap $\Rightarrow$ higher probability.

Observation 1 (Intra-cluster distance under high dimensionality)**.** *Let C be a query cluster enclosed by a hypersphere of radius R_c. According to the thin-shell concentration phenomenon [56, 69, 84], as the dimensionality D increases, all query points concentrate in a thin-shell centered at the center of the hypersphere, whose thickness and radius are $\frac{R_c}{\sqrt{D}}(1+O(\frac{1}{D}))$ and R_c respectively. Consequently, the distance between an arbitrary data point in the query cluster to the center of the hypershpere is approximately equal to R_c when $D \to +\infty$ Furthermore, the angle between two radius vectors to arbitrary query points approaches $\pi/2$ as $D \to \infty$.*

Therefore, in the high-dimensional setting considered in this work ($D \geq 128$), the distance between any two query points $q_i, q_j \in C$ can be approximated as $\text{dist}(q_i, q_j) \approx \sqrt{2}\, R_c$

Based on observation 1, it can be concluded that the distance between the centers of the bounding hyperspheres which enclose the reference points at leaf nodes on the Δ−Tree accessed by arbitrary two data points q_i and q_j in the query cluster C is approximately equal to $\sqrt{2}R_c$. Thus, according to approximation 3 and [57] [34], the approximation below can be achieved that:

Approximation 4. *For arbitrary two data points q_i and q_j in a query cluster C, and a reference point ref accessed by q_i at a leaf node that $ref \in LND(q_i)$, the probability $P(ref \notin LND(q_j) \mid ref \in LND(q_i))$ for ref not to be visited by q_j is approximately equal to $I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2})$ that*

$$P(ref \notin LND(q_j) \mid ref \in LND(q_i)) \approx 1 - \frac{V(H_i \cap H_j)}{E(V(H_k))} = 1 - I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2}) \quad (19)$$

where $\alpha \approx arccos(\frac{dist(q_i,q_j)}{2E(R(H_k))}) \approx arccos(\frac{\sqrt{2}R_c}{2E(R(H_k))})$, $E(V(H_k))$ and $E(R(H_k))$ are the expected volume and radius of the bounding sphere H_k for an arbitrary query q_k in C, $I_z(a,b)$ represents the Regularized Incomplete Beta Function computed by $\frac{B_z(a,b)}{B(a,b)}$, $B(a,b)$ represents the Beta Function computed by $\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$, and $B_z(a,b)$ represents the Incomplete Beta Function computed by $\int_0^z t^{a-1}(1-t)^{(b-1)}dt$.

Therefore, according to the deduction above and assumption 2, the probability for an arbitrary reference point visited at a leaf node for the i-th data point q_i in a query cluster not to be accessed by any q_j in the query cluster where $1 \leq j < i$ can be approximated as what follows:

$$P(ref \notin LND(q_1) \cup ... \cup LND(q_{i-1}) | ref \in LND(q_i)) \approx (1 - I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2}))^{(i-1)} \quad (20)$$

As a result, the expected number of reference points visited at a leaf node on the Δ-Tree for the i-th query data point in a cluster that are not visited by the first $i-1$ queries in the cluster can be attained approximately by the following formula:

$$E_{new}(LND(q_i)) \approx E(LND(q))(1 - I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2}))^{i-1} \quad (21)$$

Based on the assumptions and approximation presented above, the function $P_{rep}(N_c)$ can be build.

Theorem 1 (Number of unique reference points)**.** *The expected number $E_{new}(LND(C))$ of unique reference points visited at leaf nodes on the Δ-Tree for a whole query cluster C sized N_c is equal to the sum, over each query $q_i \in C$, of the expected number of reference points visited at the leaf nodes for q_i that are not accessed for any preceding query that:*

$$E_{new}(LND(C)) = \sum_{i=1}^{N_c} E_{new}(LND(q_i)) \approx \frac{E(LND(q_k))(1 - (1 - I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2}))^{N_c-1})}{I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2})} \quad (22)$$

The deduction above involves the formula for the sum of geometric series, because the formula obtained in approximation 21 for the expected number of new reference points visited at leaf nodes provided by the i-th data point in the query cluster sized N_c is a geometric series of length N_c

Based on theorem 1, $P_{rep}(N_c)$ can be directly approximated that:

$$\begin{aligned} &P_{rep}(N_c) \\ \approx &\frac{N_c E(|LND(q_k)|) - \frac{E(|LND(q_k)|)(1-(1-I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2}))^{N_c-1})}{I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2})}}{N_c E(|LND(q_k)|)} \\ = &1 - \frac{1 - (1 - I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2}))^{N_c-1}}{I_{sin^2\alpha}(\frac{D+1}{2}, \frac{1}{2})N_c} \end{aligned} \quad (23)$$

In the formula for $P_{rep}(N_c)$ above, the parameters R_c involved in the computation of the variable α can be sampled based on assumption 1, and $E(R(H_k))$ and $E(LND(q_k))$ can also achieved by sampling. The concrete sampling methods are introduced in the appendix A.1.

Cost for cluster-to-cluster pruning. We analyze cluster-to-cluster pruning under a Δ-Tree, all of whose properties are given. Let the total number of levels (excluding leaves) be Λ. For each level l ($1 \leq l \leq \Lambda$), let $\Upsilon(l)$ denote the cumulative variance (i.e., the cumulative sum of eigenvalues) of the PCA dimensions used up to level l, and let $d(l)$ be the dimensionality at level l. We write $\mathcal{P}(l)$ for the set of reference points pruned at level l (thus $|\mathcal{P}(l)|$ is the pruning count at that level).

Assumption 3 (Variance-proportional pruning)**.** *The pruning power at level l is proportional to its cumulative explained variance:*

$$|\mathcal{P}(l)| \propto \Upsilon(l), \quad 1 \leq l \leq \Lambda.$$

Assumption 4 (Complementarity of Cluster-to-Cluster and Point–to-Point Pruning)**.** *Point-to-cluster between the cluster-to-cluster pruning and point-to-point pruning reduces work for individual queries, but it does not reduce the number of unique reference points visited during the point-to-point pruning stage: once an cluster whose child node is a leaf node on the Δ−Tree survives all upper-level cluster-to-cluster pruning, it is (with high probability) retained by at least one data point in the query batch during its point-to-cluster pruning stage and finally involved in the point-to-point pruning. Consequently, it can be assumed that the set of references pruned by cluster-to-cluster pruning forms the complement of the candidate set retained for point-to-point pruning at the leaf level:*

$$|I| \approx \sum_{i=1}^{\Lambda} |\mathcal{P}(i)| + \Big|\bigcup_{j=1}^{N_c} LND(q_j)\Big|. \quad (24)$$

Under Assumptions 3, 4, and the deduction that the number of unique reference points encountered at the point-to-point pruning stage for a query batch(cluster) can be derived by $N_c \cdot E(LND(q_k))(1-P_{rep}(N_c))$ from the analysis for the cost of the point-to-point pruning, and allocating all the reference points pruned by the query batch(cluster) in clusters across levels in proportion to $\Upsilon(l)$, we can achieve the size of the pruning set for each level that:

$$|\mathcal{P}(l)| \approx \frac{\Upsilon(l)}{\sum_{t=1}^{\Lambda}\Upsilon(t)} \cdot \Big[\, |I| - N_c\big(1 - P_{\text{rep}}(N_c)\big) \cdot E\big(LND(q_k)\big)\Big]$$

For notational simplicity, we denote the variable $\frac{\Upsilon(l)}{\sum_{t=1}^{\Lambda-1}\Upsilon(t)}$ by $\kappa(l)$ in the following discussion. Let f be the branching factor of the Δ-Tree. The number of computations at level l for cluster-to-cluster pruning (i.e., the number of surviving index clusters accessed at level l) can be approximated in what follows

$$\begin{aligned} &Comp(l) \\ \approx &\frac{|I| - \sum_{j=1}^{l-1}\kappa(j)\Big[\, |I| - N_c\big(1 - P_{\text{rep}}(N_c)\big) \cdot E\big(LND(q_k)\big)\Big]}{|I|/f^{\,l}} \end{aligned} \tag{25}$$

Finally, since the low-dimensional centroid of the query cluster is frequently accessed and small in footprint, we assume it resides in L2, because each of the index clusters on the $\Delta-$Tree is only accessed for one time for each query batch(cluster), the clusters on Δ-Tree can be assumed to be fetched from main memory. Consequently, the per-level cost and the total cluster-to-cluster pruning cost can be approximated as follows:

$$\begin{aligned} Cost(l) \approx &T_{dist}(d(l), L2, Mem)\times \\ &\frac{|I| - \sum_{j=1}^{l-1}\kappa(j)\Big[\, |I| - N_c\big(1 - P_{\text{rep}}(N_c)\big) \cdot E\big(LND(q_k)\big)\Big]}{|I|/f^{\,l}} \end{aligned} \tag{26}$$

In conclusion, the expected time cost measured in the number of clock cycles of the cluster-to-cluster pruning for a query batch(cluster) can be approximated below that:

$$\begin{aligned} E(T_{c2c}) \approx &\sum_{l=1}^{\Lambda}\Big(T_{dist}(d(l), L2, Mem)\times \\ &\frac{|I| - \sum_{j=1}^{l-1}\kappa(j)\Big[\, |I| - N_c\big(1 - P_{\text{rep}}(N_c)\big) \cdot E\big(LND(q_k)\big)\Big]}{|I|/f^{\,l}}\Big) \end{aligned} \tag{27}$$

where $P_{rep}(N_c)$ has been derived in the discussion for point-to-point pruning.

Cost for point-to-cluster pruning Similar to the deduction for the cost cycles for cluster-to-cluster pruning, the expected cost cycles for point-to-cluster pruning for each query cluster with N_c queries can be approximated as follows:

$$\begin{aligned} E(T_{p2c}) \approx &\frac{Nc(|I| - \sum_{l=1}^{\Lambda}\mathcal{P}(l))}{|I|/f^{\Lambda}} \\ &\times T_{dist}(d(\Lambda), L2, L1) \end{aligned} \tag{28}$$

The detailed deduction is in appendix A.1.

Cost for data assignment The expected cost in clock cycles for the stage of assigning to queries to different batches can be approximated in the followings:

$$\begin{aligned} E(T_{clus}) \approx &|W|^2/N_c \cdot T_{dist}(d(\Lambda), L2, L1) \\ &+ |W| \cdot T_{minz}(|W|/N_c, L2). \end{aligned} \tag{29}$$

where T_{minz} can be computed based on estimation 2 The detailed deduction is in appendix A.1.

In this stage, the expected cost in clock cycles for update the kNN Join table against W new queries by a batch size of N_c can be computed by formulas 10, 29, 28, and 9, with N_c as the only input variable with other parameters can be attained by sampling methods. Optimization can be carried out by numerically solving for the zeros of the derivative of the derived function for $E_{clk}(N_c)$ and achieve the optimal N_c to gain the theoretical smallest clock cycle cost.

5 EXPERIMENT

To validate the advantage of our newly proposed batch strategy for kNN Join on dynamic high-dimensional data in comparison with that in a single manner, a series of experiments are conducted and the results are presented in this section.

5.1 Datasets and Computing Environment

We evaluate our approach on eight real-world datasets covering images, emails, audio, and music. To test scalability on large-scale data, we include the Million Song Dataset, which contains one million tracks with 420 features each. We also use the 4,096-dimensional Trevi dataset to show the benefit of our scheme over per-query methods in extremely high-dimensional settings. Table 2 summarizes all datasets.

Table 2: Dataset Summary

Dataset	Records' No	Dimensions' No	Type
NUS-WIDE	260,000	128	Image
Fashion-MNIST	60,000	784	Image
Audio	53,000	192	Audio
Cifar	50,000	512	Image
Trevi	100,000	4096	Image
Sun	79,000	512	Image
Enron	95,000	1369	Text
Millionsong	1,000,000	420	Audio

All algorithms were implemented in C++ and compiled with g++ using the `-O3` optimization flag. Experiments ran on a server with dual Intel(R) Xeon(R) Gold 6342 CPUs (2.80 GHz base, 24 cores each, hyper-threading enabled for 48 threads per CPU, totaling 96 threads), 500 GB RAM, and a cache hierarchy comprising 64 KB L1 and 1 MB L2 per core, plus 36 MB shared L3 per CPU. The system operated on Ubuntu 22.04.1 LTS.

5.2 Experimental Settings

We compare three methods: (i) single-query (point-wise) updates on the Δ-Tree (state-of-the-art for dynamic high-dimensional kNN); (ii) our batch framework with a default heuristic cluster capacity

Table 3: Table shows the parameter settings, where one parameter (initial query set size, number of inserted queries, or k value) is varied at a time while the others are fixed; $X\%$ denotes a random sample of $X\%$ of the entire dataset.

Initial queries	Inserted queries	Reference set	k
25%	25%	{10%, 20%, 30%, 40%, 50%}	10
25%	{5%, 10%, 15%, 20%, 25%}	50%	10
25%	25%	50%	{5, 10, 15, 20, 25}

$|W|/150$, where W represents the collection of newly inserted query points; and (iii) our batch framework with the theoretically derived optimal batch (cluster) capacity. All Δ-Tree parameters (branching factor f, leaf threshold *thres*) are tuned for the fastest point-wise baseline.

Five experiments are conducted. In the first experiment, we plot update cost (clock cycles) against the batch size, marking the clock cycle cost for the batch size of $|W|/150$ and that for the single-query update, and comparing the optimal cluster size derived by our theoretical model and the empirical one, and their corresponding cost, on all the involved datasets. Datasets are split into 25% initial queries, 25% inserts, and 50% references, with $k = 10$. For all the datasets except for Millionsong and Trevi, the batch sizes are sampled at intervals of 5. Due to extreme runtime, for the dataset Millionsong, we sample cluster sizes at intervals of 5 around the valley segment, and 50/5000 for the local descent/long ascent segments before/after the valley; for Trevi, we use 5 in the early descent and valley phases and 50 in the later long ascent. In the second experiment, Time-cost curves are plotted for all datasets, with the cost for the point-wise update strategy and that for the batch framework with the default $|W|/150$ cluster size, and the theoretical and empirical optima compared. To study sensitivity to workload parameters, we design three experiments on NUS-WIDE, which is a classical dataset for kNN Join [80, 81, 83], as summarized in Table 3. In each experiment, one parameter is varied while the others are fixed. Specifically, we examine the impact of varying the reference set size, the number of inserted queries, and the value of k in kNN, respectively. In all cases, the initial queries, inserted queries, and reference sets are formed by non-overlapping random sampling from the entire dataset, ensuring no intersection among the three parts.

Experiment-1: Effect of batch capacity on cost cycles Figure 2 exhibits the very similar behaviors that across all datasets, the total update cost (in number of clock cycles and in time) decreases sharply as the cluster capacity increases in the low-capacity regime. The primary reason is improved data reuse at the leaf nodes of the Δ-Tree: when clusters are small, query points are tightly grouped, their bounding spheres overlap significantly, and newly added queries introduce only a few unseen reference points. This increases the revisit probability (P_{rep}) and reduces the cost for data access during distance computation cost.

Two secondary factors also contribute. (i) Although larger clusters make internal points more dispersed, their queries still share similar search paths on the Δ-Tree, so the added pruning overhead at non-leaf nodes remains small in this regime. (ii) Larger clusters

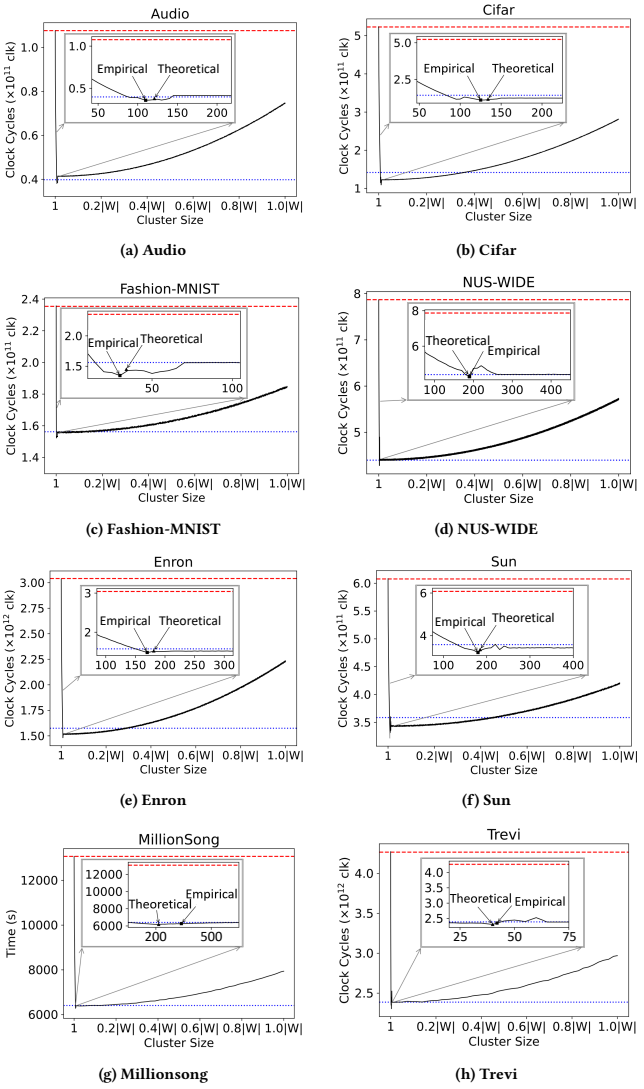
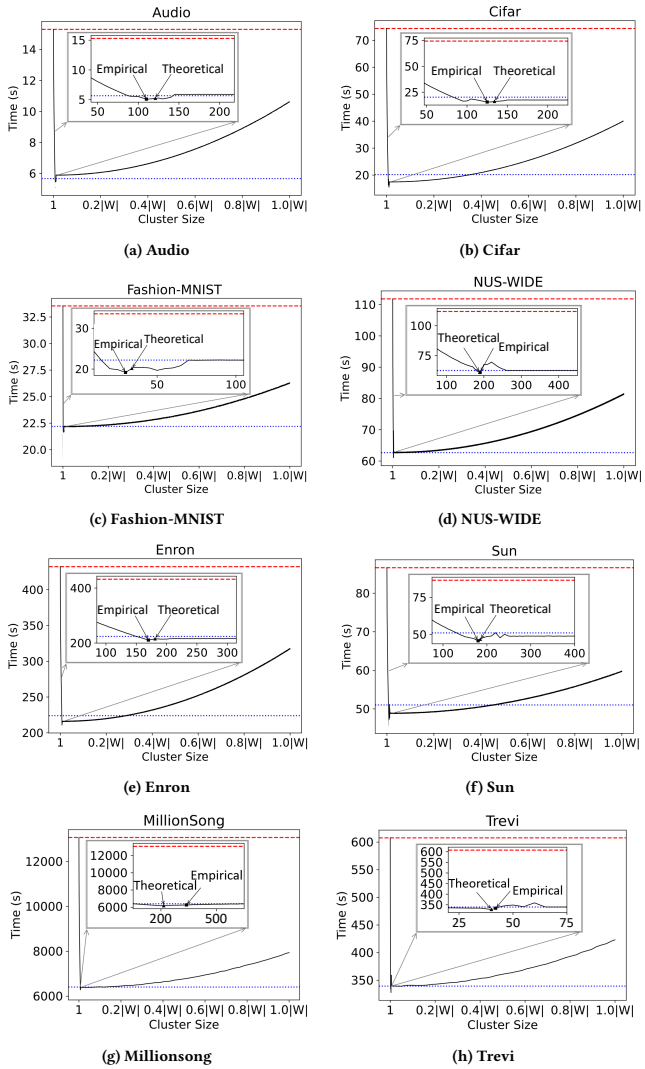


Figure 2: Curves of cost clock VS cluster capacity.

reduce the number of clusters needed to partition the query batch, lowering the clustering cost.

In summary, within the low-capacity regime, the strong benefits of higher data reuse and reduced clustering overhead outweigh the modest increase in pruning cost, leading to a consistent decline in total time cost as capacity grows.

As cluster capacity increases, the performance curve for dynamic kNN Join exhibits a brief, quasi-stable plateau across all datasets. This plateau represents a short-lived point of diminishing returns, where the initial benefits of clustering the new query points are temporarily offset by emerging overheads before further growth in cost.



(a) Audio (b) Cifar

(c) Fashion-MNIST (d) NUS-WIDE

(e) Enron (f) Sun

(g) Millionsong (h) Trevi

Legends: ▲ Theoretical optimal point ■ Empirical optimal point
Time cost vs cluster size Time cost (cluster size=$|W|/150$)
Time cost (cluster size=1)

Figure 3: Curves of Time cost VS cluster capacity.

On one hand, larger clusters have greater radii, reducing the volumetric overlap among bounding hyperspheres of queries. As a result, new queries are more likely to access reference points not already cached, and the benefit of data reuse at leaf nodes diminishes. On the other hand, larger intra-cluster distances increase the number of clusters traversed at the non-leaf nodes on the Δ-Tree and intensify the cluster-to-cluster and point-to-cluster pruning, thereby raising computational cost.

In essence, a balance is briefly reached between the diminishing cache-reuse benefit at the leaf level and the growing overhead at non-leaf levels. This explains why, after an initial rapid decline, the total cost shows a short-lived plateau with little variation before resuming its upward trend. While some datasets exhibit performance fluctuations within this short phase due to their specific data distributions, they still do not show a clear monotonic increasing or decreasing trend.

As shown consistently on all the datasets in figure 2, when the cluster capacity enters the high-value range, performance begins to degrade. While our cost model assumes that reference points at the leaf nodes typically remain in the L1 and L2 cache and the query points in the cluster are reserved in L2 cache—a reasonable assumption in the low-capacity regime where distances between reusing the reference points are very short—this assumption gradually breaks down as capacity grows. The large size of high-capacity clusters causes many reference points to spill into lower-level caches due to excessively long reuse distances, which slows their retrieval during point-to-point pruning. Meanwhile, high-capacity clusters also trigger evictions of many query points from L2, forcing them into slower memory tiers and thereby increasing memory-access cost. Together, these effects explain the observed slowdown of dynamic kNN Join under large cluster capacities.

From both figure 2 and table 4, it can be observed evidently that our cluster-based batch kNN join framework consistently outperforms the point-wise strategy. Even when the batch size is chosen heuristically as $|W|/150$ rather than using the theoretically derived optimum, the batch framework achieves an average speedup of 202.6% over single-query updates, with the best case reaching 369%. The relative deviation between the theoretical and empirical optimal batch capacities exceeds 10% on only three datasets, and remains within 10% on all others. Even in the worst case, the cycle cost at the theoretical optimum is only 8% higher than that at the empirical optimum, since both optima lie within a plateau region of the performance curve. Moreover, with the theoretical optimal batch capacity, the framework delivers an average speedup of 224.4% over single-query updates, with the maximum speedup reaching 423%.

Experiment-2: Effect of batch capacity on cost time. From the figure 3 and table 5, we observe that the relationship between execution time and batch capacity closely mirrors that of cycle cost: a rapid decline, followed by a short plateau or mild fluctuations, and finally an increase as capacity grows. The table further shows that even with the heuristic batch size, our batch framework significantly accelerates dynamic kNN Join compared to point-wise updates, achieving an average speedup of 205% and a maximum of 368%. The gap between the theoretically predicted cycle-optimal batch capacity and the empirically observed time-optimal capacity exceeds 10% on only three datasets, with the largest deviation reaching 13%. Notably, on the dataset with this maximum discrepancy, the cycle-optimal batch capacity still yields the highest observed time speedup of 431% over the point-wise baseline. Overall, under the theoretical cycle-optimal capacity, our framework achieves an average time-based speedup of 229%, with the maximum again reaching 431%.

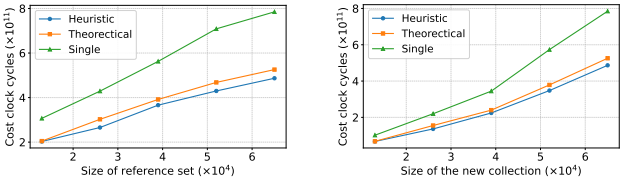
It is natural that the empirically observed optimal batch capacity may deviate from the theoretical prediction. First, our cycle-cost model is derived under a set of assumptions, and any mismatch between these assumptions and practical conditions can introduce deviations. Second, when estimating the cycle cost of operations such as distance computations, minimum selection, and data transfers, we assume full utilization of the available bandwidth. In practice,

Table 4: Relative errors and acceleration in experiment 1 (all in %). o': theoretical optimum (cycles); o: empirical optimum (cycles); T: cycle cost; $SU(x)$: speedup over single-query updates (in cycles).

Dataset	$\frac{\lvert o-o'\rvert}{o}$ (↓)	$\frac{\Delta T(o',o)}{T(o)}$ (↓)	$SU(o')$ (↑)	$SU(\lvert W\rvert/150)$ (↑)
Audio	11	8	282	269
Cifar	9	12	423	369
Enron	9	10	200	183
Minist	14	15	168	151
Msong	63	8	203	183
NUSWIDE	7	9	161	150
Sun	8	10	177	153
Trevi	10	10	181	163

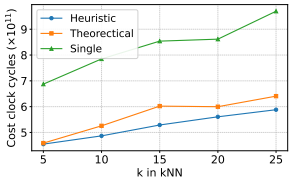
Table 5: Relative errors and acceleration in experiment 2 (all in %). o': theoretical optimum (cycles); o: empirical optimum (time); T: elapsed time; $SU(x)$: speedup over single-query updates (time).

Dataset	$\frac{\lvert o-o_t\rvert}{o_t}$ (↓)	$\frac{\Delta T(o_t,o)}{T(o)}$ (↓)	$SU(o')$ (↑)	$SU(\lvert W\rvert/150)$ (↑)
Audio	10	9	290	270
Cifar	9	13	431	368
Enron	9	10	202	187
Minist	14	13	172	153
Msong	68	9	208	189
NUSWIDE	8	9	165	156
Sun	8	9	179	151
Trevi	10	11	185	166



(a) Vary the reference set size

(b) Vary the size of the new collection



(c) Vary k

Figure 4: Effect of the dataset parameters.

however, memory-hierarchy bandwidth is often partially occupied by system activities or competing applications, preventing sustained peak throughput and thereby introducing additional modeling error. Nevertheless, these factors primarily affect absolute cycle counts rather than the relative trends, and the model remains effective in predicting near-optimal batch capacities.

Experiment-3 Effect of the reference set size: As the reference set grows (Figure 4(a)), all three methods exhibit higher cycle cost because more reference points must be scanned and more distances evaluated to determine the kNN results. Across the full range of sizes, both batch variants—using the default batch size and the theoretically optimal batch size—consistently outperform the point-wise baseline.

Experiment-4 Effect of the insertion number: We observe a monotonic increase in cycle cost for all three methods as the number of insertions rises (Figure 4(b)); each additional query triggers a kNN search (and, for the batch approaches, collective searches over more batches). Nevertheless, at every tested insertion level, the two batch configurations remain consistently faster than the point-wise strategy.

Experiment-5 Effect of the k value: Increasing k (Figure 4(b)) expands the candidate neighborhood, leading to more reference points tested and greater data access; consequently, all methods incur higher cycle cost. Throughout the evaluated k values, both batch frameworks—default and theoretically optimal—consistently surpass the point-wise baseline.

6 CONCLUSION

We presented a clustering-based batch framework for dynamic high-dimensional kNN Join that turns spatial proximity into cache reuse. By (i) precomputing anchors to align processing order with geometry, (ii) performing level-aware pruning in PCA subspaces with proven safety, and (iii) executing a collective kNN search that services co-located queries while leaves remain hot, our design compresses reuse distance and cuts DRAM↔LLC traffic—the true bottleneck once arithmetic is amortized by SIMD. In summary, our collection-based framework provides a simple and effective way to accelerate dynamic high-dimensional kNN Join, and we expect it to serve as a useful building block for future systems and applications.

7 APPENDIX

A.1

Cache residence probability When a reference point in a leaf node is revisited, the probability that it resides in cache tier L_j can be estimated by

$$P(ref \in Lj \mid rep) \approx \frac{A_j}{E(|LND|)}, \qquad j \in \{1,2\}, \tag{30}$$

where

$$A_j = \min\Big\{ \max\big(E(|LND|) - \sum_{k=1}^{j-1} |Lk|,\ 0\big),\ |Lj| \Big\}, \tag{31}$$

that $E|LND| \approx thres \cdot D \cdot S$ is the expected value of the data size $|LND|$ of a leaf node that can be approximated by the multiplication of the clustering threshold value *thres* of Δ−Tree, the dimensionality of the data, and the length of one dimension of data.

The conclusion above is derived from the geometric probability model, predicated on a key observation: during point-to-point pruning, the extremely short revisit intervals for reference points within a leaf node ensure they remain resident at the top of the cache hierarchy. This effect is further supported by the fact that the total data volume of a leaf node is generally contained within the combined capacity of L1 and L2 caches.

Cost for point-to-cluster pruning. To further estimate the cost of point-to-cluster pruning in the process of the collective kNN Search on a Δ−Tree of a query batch within the batch framework of dynamic kNN Join, we leverage the pruning counts $\mathcal{P}(l)$ $(1 \leq l \leq \Lambda)$ that were already derived for the cluster-to-cluster pruning stage at each non-leaf level of the Δ-Tree. Since $\mathcal{P}(l)$ can be expressed in terms of the batch (cluster) size N_c and other variables that can be sampled, the following approximation in terms of $\mathcal{P}(l)$ and the known size $|I|$ of the whole reference set I can be used to express the expected number $E(R_{c2c})$ of reference points that remain for a single query point in the batch after the cluster-to-cluster pruning:

$$E(R_{c2c}) \approx |I| - \sum_{l=1}^{\Lambda} \mathcal{P}(l). \tag{32}$$

At the last non-leaf level of the Δ-Tree, each cluster has an expected capacity of $\frac{|I|}{f^{\Lambda}}$, where f is the fanout number of the Δ-Tree. Consequently, the expected number of clusters visited by a single data point in the query batch (cluster) during the point-to-cluster pruning stage can be approximated as

$$E(C_{p2c}) \approx \frac{|I| - \sum_{l=1}^{\Lambda} \mathcal{P}(l)}{|I|/f^{\Lambda}}. \tag{33}$$

During the point-to-cluster pruning phase, the access interval for the low-dimensional center of the cluster on the Δ−Tree whose child node is a leaf node is extremely shortly bounded by the combined size of one full-dimensional query point and all full-dimensional reference points stored in the leaf node. Hence, the low-dimensional cluster center can be reasonably assumed to remain in the L1 cache. Meanwhile, the low-dimensional versions of the query points constitute hot data throughout the collective kNN Search, being repeatedly accessed across the whole process. Although the access interval for one query is longer, including the data involved in portions of all three pruning steps, it still falls within the range suitable for L2 cache residency. Therefore, we conservatively assume that the low-dimensional queries reside in L2 cache. As a result, the expected time cost measured in clock cycles for the point-to-cluster pruning for a query cluster(batch) can be approximately estimated as follows:

$$\begin{aligned} E(T_{p2c}) \approx & \frac{Nc(|I| - \sum_{l=1}^{\Lambda} \mathcal{P}(l))}{|I|/f^{\Lambda}} \\ & \times T_{dist}(d(\Lambda), L2, L1) \end{aligned} \tag{34}$$

Correction to Equation 28 in Section 4.2. Equation 28 in Section 4 contains a minor typographical error. The correct form should be as what is presented above. This error is due to a minor proofreading oversight; the experimental results and conclusions remain unaffected.

Cost for Data Assignment. As is assumed in section 4, setting the number of batches (clusters) to be equal to $\frac{|W|}{N_c}$ yields batches with sizes approximately equal to N_c, therefore, the number of batches (clusters) that corresponds to a batch size of N_c can be estimated as $\frac{|W|}{N_c}$. Each new query must compute its distances to all batch anchors, resulting in $\frac{|W|^2}{N_c}$ distance computations in total. In addition, each query requires one minimization step to identify its closest batch, incurring $|W|$ minimization operations overall.

The cycle cost for a single distance computation in this stage for assignment of the new query points to different batches can be modeled as $T_{dist}(d(\Lambda), L2, L1)$. This is because distances are computed in the reduced space spanned by the first $d(\Lambda)$ principal components of the covariance matrix derived from the union of the initial queries and the reference set, as is designed in subsection 3.2. During this process, access of one query point has extremely short intervals (bounded by the size of one low-dimensional anchor plus one distance), ensuring that query vectors remain in L1 cache. Batch anchors are also hot data but exhibit longer reuse intervals, upper-bounded by the aggregate size of all low-dimensional batch anchors and associated distances; hence, they are likely evicted from L1 but still reside in L2 under reasonable numbers of clusters.

The cycle cost for a minimization operation can be approximated by $T_{minz}(|W|/N_c, L2)$, since the number of candidate distances is $|W|/N_c$. These distances are already generated and retained in cache when minimization occurs, but their aggregate size typically exceeds L1 capacity, forcing them into L2 cache.

In summary, the expected cycle cost for assigning new queries into clusters can be expressed as:

$$\begin{aligned} E(T_{clus}) \approx & |W|^2/N_c \cdot T_{dist}(d(\Lambda), L2, L1) \\ & + |W| \cdot T_{minz}(|W|/N_c, L2). \end{aligned} \tag{35}$$

Sampling Strategies for Parameter Estimation. To construct the cost model for the batch kNN strategy, three parameters remain to be estimated by sampling: the batch (cluster) radius R_c, the expected radius $E(R(H_k))$ of the minimal bounding hypersphere for a query, and the expected number $E(|LND(q_k)|)$ of reference points visited at the leaf nodes on the Δ-Tree by a query point. All the samplings are on the initial query set U before the insertion of new query points starts. This choice is justified by the observation that the distribution of subsequently inserted queries is similar to that of U.

Both $E(R(H_k))$ and $E(|LND(q_k)|)$ can be approximated by random sampling from the initial query set. Specifically, $E(R(H_k))$ is estimated as the average radius of minimal bounding hyperspheres enclosing the reference points visited at the leaf nodes on the Δ−Tree by sampled queries:

$$E(R(H_k)) \approx \overline{R(H)_{\text{sample}}}. \tag{36}$$

Similarly, $E(|LND(q_k)|)$ is obtained as the average number of references accessed at the leaf nodes on the Δ−Tree for sampled queries:

$$E(|LND(q_k)|) \approx \overline{|LND(q)|_{\text{sample}}}. \tag{37}$$

The cluster radius R_c is estimated via the high-dimensional sphere formula and the assumption that cluster volume V_c grows linearly with cluster size N_c, which can be written as $V_c = \frac{\pi^{D/2} R_c^D}{\Gamma(\frac{D}{2}+1)} \approx kN_c + b$. To calibrate this relation, we apply k-means with two different cluster counts k_1 and k_2 on the initial query set, yielding average radii $\overline{R_c^1}$ and $\overline{R_c^2}$. The corresponding pairs $(|U|/k_1, \overline{R_c^1})$ and $(|U|/k_2, \overline{R_c^2})$ are then used to solve for the linear parameters, which allows us to derive the functional relation between R_c and N_c.

A.2

All costs for Euclidean Distance calculation and minimization, which are two core operations in our batch framework, are measured in clock cycles and explicitly account for data movement

Table 6: Mapping of symbols to AVX-512 instructions / Hardware parameters.

Symbol	AVX-512 /Hardware parameters	Semantics
VSUB_τ	VSUBPS, VSUBPD VPSUBD, VPSUBQ	Packed subtraction
VFMA_τ	VFMADDPS, VFMADDPD	Fused multiply-add
vadd_τ	VADDPS, VADDPD VPADDD, VPADDQ	Packed addition
perm_τ	VSHUFPS, VSHUFPD VPERMD, VPERMQ VEXTRACTF32x4, VEXTRACTF64x4 VPEXTRACT*	Permute / shuffle / extract
VMIN_τ	VMINPS, VMINPD VPMIN*	Packed minimum
sqrt_τ	VSQRTPS, VSQRTPD VRSQRT14PS	Square root
τ_k (latency)	Mem→L3, L3→L2 L2→L1, L1→Regs	Link startup latency
BW_k (bandwidth)	Mem→L3, L3→L2 L2→L1, L1→Regs	Link throughput

across the memory hierarchy. We assume an x86 server-class CPU with AVX-512 support. In this work, data size for one element in the involved vectors is defined as S bytes, therefore, a 512-bit vector register (zmm) accommodates $V = 64/S$ lanes, and the number of vector chunks is $N = \lceil D/V \rceil$ for a $D-$dimensional vector. Memory tiers are denoted $Loc \in \{L1, L2, L3, Mem\}$ and registers as $L0$. The data transfer cost from tier(s) into registers is represented by $T_{\text{trans}}(\cdot)$.

Because different data types (e.g., FP32, FP64, integers) invoke different AVX-512 instructions, we use generic symbols (e.g., VSUB, VFMA) in our cost formulas to keep the derivation concise. Table 6 summarizes the mapping between these abstract symbols and the representative AVX-512 instructions they correspond to.

Cycle cost for Euclidean Distance. Given two D-dimensional vectors stored at $Loc_1, Loc_2 \in \{L1, L2, L3, Mem\}$, the cost of computing their Euclidean distance is

$$T_{\text{dist}}(D, Loc_1, Loc_2) = T_{\text{trans}}(D, Loc_1, Loc_2) + N \cdot \left(T_{\text{VSUB}_\tau} + T_{\text{VFMA}_\tau}\right) + T_{\text{reduce}}(V) + T^{(\text{opt})}_{\text{sqrt}_\tau} \tag{38}$$

where a tree-style horizontal reduction over V lanes costs $T_{\text{reduce}}(V)$ is approximately equal to $\lceil \log_2 V \rceil \cdot \left(T_{\text{perm}_\tau} + T_{\text{vadd}_\tau}\right)$. When comparing squared distances, T_{sqrt} is omitted.

The cost of computing the Euclidean distance between two D-dimensional vectors is decomposed into the following pipeline. First, the two input vectors are transferred from their residing tiers $Loc_1, Loc_2 \in \{L1, L2, L3, Mem\}$ into CPU registers, which incurs the data movement cost $T_{\text{trans}}(D, Loc_1, Loc_2)$. Once in registers, each pair of V-lane zmm vectors (where $V = 64/S$ for element width S) is processed by one packed subtraction instruction VSUB_τ to compute the elementwise differences. The squared contributions are accumulated in a register-local accumulator using fused multiply-add (VFMA_τ), so that for each chunk the operation $acc \leftarrow acc + (a_i - b_i)^2$ is realized in a single instruction. Across $N = \lceil D/V \rceil$ such chunks, the total arithmetic contribution is therefore $N \cdot (T_{\text{VSUB}_\tau} + T_{\text{VFMA}_\tau})$.

After accumulation at the register level, the partial sums across lanes must be reduced into a single scalar. This is achieved by a tree-style horizontal reduction, which consists of $\lceil \log_2 V \rceil$ stages of lane permutations (perm_τ) followed by packed additions (vadd_τ). We denote this overhead as $T_{\text{reduce}}(V) \approx \lceil \log_2 V \rceil \cdot (T_{\text{perm}_\tau} + T_{\text{vadd}_\tau})$. Finally, if the exact Euclidean distance (rather than squared distance) is required, an additional square-root instruction T_{sqrt_τ} is applied to the reduced scalar.

This formula explicitly incorporates both data movement and vectorized arithmetic, providing a calibrated basis for the subsequent development of the cost model for the batch kNN framework.

Cycle cost for minimization. For computing the minimum over a D-element array stored at $Loc \in \{L1, L2, L3, Mem\}$, the cycle cost is

$$T_{\text{minz}}(D, Loc) = T_{\text{trans}}(D, Loc) + (N-1)\, T_{\text{VMIN}_\tau} + \lceil \log_2 V \rceil \cdot \left(T_{\text{perm}_\tau} + T_{\text{VMIN}_\tau}\right). \tag{39}$$

To compute the minimum of a D-element array stored at $Loc \in \{L1, L2, L3, Mem\}$, the vector must first be transferred into CPU registers, which contributes the movement cost $T_{\text{trans}}(D, Loc)$. Once in registers, the array is partitioned into $N = \lceil D/V \rceil$ vector chunks, each of V elements. The first chunk initializes the accumulator. For each of the remaining $N-1$ chunks, a packed minimum instruction (VMIN_τ) is applied between the accumulator and the current chunk, updating the accumulator with the elementwise minima. This process yields the $(N-1)\, T_{\text{VMIN}_\tau}$ term in the formula.

After all chunks are processed, the accumulator holds the minimum values across all V lanes. These must be reduced to a single scalar result. Similar to the distance case, a tree-style horizontal reduction is performed with $\lceil \log_2 V \rceil$ stages, where each stage consists of a lane permutation (perm_τ) followed by a packed minimum (VMIN_τ). The resulting cost is captured by the term $\lceil \log_2 V \rceil \cdot (T_{\text{perm}_\tau} + T_{\text{VMIN}_\tau})$.

Cycle cost for data transfer. Let the cache/memory links $L_k \rightarrow L_{k-1}$ ($k \in \{1, 2, 3, 4\}$, with $L_4 = Mem$ and $L_0 = \text{Regs}$) have startup latencies τ_k (cycles) and sustainable bandwidths BW_k (bytes/cycle). Transferring D elements (each S bytes) from $Loc = L_k$ to registers is bounded by

$$T_{\text{trans}}(D, L_k) \;\approx\; \max\left(\sum_{j=1}^{k} \tau_j,\; \frac{DS}{\min_{1 \le j \le k} BW_j}\right).$$

For two input vectors at $Loc_1 = L_x$ and $Loc_2 = L_y$ (w.l.o.g. $x \le y$), a conservative bound is

$$T_{\text{trans}}(D, Loc_1, Loc_2) \;\le\; T_{\text{trans}}(D, L_x) + T_{\text{trans}}(D, L_y).$$

The transfer of D elements of width S bytes from tier L_k (e.g., L_1 for L1 cache, L_4 for main memory) to the CPU registers L_0 can be modeled as the cost of traversing k links in the memory hierarchy. Each link $L_j \rightarrow L_{j-1}$ is characterized by a startup latency τ_j (in cycles) and a sustainable throughput BW_j (bytes per cycle). In practice, the observed cost is governed both by the cumulative startup latency of the chain and by the effective throughput of the slowest link. To conservatively capture both effects, we use the maximum

of the two quantities:

$$T_{\text{trans}}(D, L_k) \;\approx\; \max\Bigg(\underbrace{\sum_{j=1}^{k} \tau_j}_{\text{cumulative latency}} \;,\; \underbrace{\frac{DS}{\min_{1 \le j \le k} BW_j}}_{\text{bandwidth bottleneck}} \Bigg).$$

This approximation is consistent with cost models adopted in architecture aware performance analysis [55, 61, 87, 88, 92], where the two terms reflect the latency-bound and throughput-bound regimes respectively.

When two input vectors are located at different tiers, say $Loc_1 = L_x$ and $Loc_2 = L_y$ with $x \le y$, we upper bound the total transfer cost as

$$T_{\text{trans}}(D, Loc_1, Loc_2) \;\le\; T_{\text{trans}}(D, L_x) + T_{\text{trans}}(D, L_y),$$

which ensures safety regardless of possible overlap between the two transfers. While this bound may be looser than the true overlapped cost, it simplifies analysis and suffices for our asymptotic conclusions in later sections. Moreover, our experimental results demonstrate that our newly proposed cost model for the batch kNN framework based on this estimation of the cost for movement of two vectors can predict the batch size that is consistently near the empirically optimal choice.

REFERENCES

[1] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.

[2] Mohamed Abdul-Al, George Kumi Kyeremeh, Naser Ojaroudi Parchin, Raed A Abd-Alhameed, Rami Qahwaji, and Jonathan Rodriguez. 2021. Performance of multimodal biometric systems using face and fingerprints (short survey). In *2021 IEEE 26th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. IEEE, 1–6.

[3] Andreas Abel and Jan Reineke. 2018. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. *arXiv preprint arXiv:1810.04610* (2018). https://arxiv.org/abs/1810.04610 Detailed modeling of instruction performance, including SIMD instructions.

[4] Rishabh Ahuja, Arun Solanki, and Anand Nayyar. 2019. Movie recommender system using k-means clustering and k-nearest neighbor. In *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. IEEE, 263–268.

[5] Khaled Gubran Al-Hashedi and Pritheega Magalingam. 2021. Financial fraud detection applying data mining techniques: A comprehensive review from 2009 to 2019. *Computer Science Review* 40 (2021), 100402.

[6] Huda Aldosari. 2024. Garra Rufa Fish Optimization-based K-Nearest Neighbor for Credit Card Fraud Detection. In *2024 International Conference on Distributed Computing and Optimization Techniques (ICDCOT)*. IEEE, 1–5.

[7] AMD. 2023. Next Generation Zen 4 Core and 4th Gen AMD EPYC. In *Proceedings of Hot Chips 35 (HC35)*. https://hc2023.hotchips.org/assets/program/conference/day1/CPU1/HC_Zen4_Epyc_Final_20230825%20-%20Embargoed%20until%20Aug%2029%202023.pdf Official vendor deck; shows L2 increased to ~1 MiB per core.

[8] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51, 1 (2008), 117–122.

[9] Yahia Baashar, Gamal Alkawsi, Nor'ashikin Ali, Hitham Alhussian, and Hussein T Bahbouh. 2021. Predicting student's performance using machine learning methods: A systematic literature review. In *2021 International Conference on Computer & Information Sciences (ICCOINS)*. IEEE, 357–362.

[10] Kristen M Bellisario, Taylor Broadhead, David Savage, Zhao Zhao, Hichem Omrani, Saihua Zhang, John Springer, and Bryan C Pijanowski. 2019. Contributions of MIR to soundscape ecology. Part 3: Tagging and classifying audio features using a multi-labeling k-nearest neighbor approach. *Ecological Informatics* 51 (2019), 103–111.

[11] Uzair Aslam Bhatti, Linwang Yuan, Zhaoyuan Yu, Saqib Ali Nawaz, Anum Mehmood, Mughair Aslam Bhatti, Mir M Nizamani, Shengjun Xiao, et al. 2021. Predictive data modeling using sp-kNN for risk factor evaluation in urban demographical healthcare data. *Journal of Medical Imaging and Health Informatics* 11, 1 (2021), 7–14.

[12] Vishwanath Bijalwan, Vinay Kumar, Pinki Kumari, and Jordan Pascual. 2014. KNN based machine learning approach for text and document mining. *International Journal of Database Theory and Application* 7, 1 (2014), 61–70.

[13] Christian Böhm and Florian Krebs. 2004. The k-nearest neighbour join: Turbo charging the kdd process. *Knowledge and Information Systems* 6 (2004), 728–749.

[14] Christian Bohm, Beng Chin Ooi, Claudia Plant, and Ying Yan. 2006. Efficiently processing continuous k-nn queries on data streams. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 156–165.

[15] Giovanni Bonetta, Rossella Cancelliere, Ding Liu, and Paul Vozila. 2021. Retrieval-augmented Transformer-XL for close-domain dialog generation. *arXiv preprint arXiv:2105.09235* (2021).

[16] Kaushik Chakrabarti and Sharad Mehrotra. 2000. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *VLDB Conference*.

[17] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. 2023. Finger: Fast inference for graph-based approximate nearest neighbor search. In *Proceedings of the ACM Web Conference 2023*. 3225–3235.

[18] Vanajaroselin Chirchi, Emmanvelraj Chirchi, and Khushi E Chirchi. 2024. Pattern matching for the iris biometric recognition system uses KNN and fuzzy logic classifier techniques. *International Journal of Information Technology* 16, 5 (2024), 2937–2944.

[19] Intel Corporation. 2025. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Lists intrinsics and associated instruction latency and throughput across Intel microarchitectures.

[20] Adolfo Crespo Márquez. 2022. The curse of dimensionality. In *Digital maintenance management: guiding digital transformation in maintenance*. Springer, 67–86.

[21] Bin Cui, Beng Chin Ooi, Jianwen Su, and Kian-Lee Tan. 2003. Contorting high dimensional data for efficient main memory KNN processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 479–490.

[22] Bin Cui, Heng Tao Shen, Jialie Shen, and Kian Lee Tan. 2005. Exploring bit-difference for approximate knn search in high-dimensional databases. (2005).

[23] Belur V Dasarathy. 1991. Nearest neighbor (NN) norms: NN pattern classification techniques. *IEEE Computer Society Tutorial* (1991).

[24] BL Deekshatulu, Priti Chandra, et al. 2013. Classification of heart disease using k-nearest neighbor and genetic algorithm. *Procedia technology* 10 (2013), 85–94.

[25] Zhenyun Deng, Xiaoshu Zhu, Debo Cheng, Ming Zong, and Shichao Zhang. 2016. Efficient kNN classification algorithm for big data. *Neurocomputing* 195 (2016), 143–148.

[26] Yuanyuan Fu, Hong S He, Todd J Hawbaker, Paul D Henne, Zhiliang Zhu, and David R Larsen. 2019. Evaluating k-Nearest Neighbor (k NN) Imputation Models for Species-Level Aboveground Forest Biomass Mapping in Northeast China. *Remote sensing* 11, 17 (2019), 2005.

[27] Hanyao Gao, Gang Kou, Haiming Liang, Hengjie Zhang, Xiangrui Chao, Cong-Cong Li, and Yucheng Dong. 2024. Machine learning in business and finance: a literature review and research opportunities. *Financial Innovation* 10, 1 (2024), 86.

[28] Jianyang Gao and Cheng Long. 2023. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[29] Soumya Suvra Ghosal, Yiyou Sun, and Yixuan Li. 2024. How to overcome curse-of-dimensionality for out-of-distribution detection?. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 19849–19857.

[30] Amol Ghoting, Srinivasan Parthasarathy, and Matthew Eric Otey. 2008. Fast mining of distance-based outliers in high-dimensional datasets. *Data Mining and Knowledge Discovery* 16, 3 (2008), 349–364.

[31] Luca Giommi. 2023. Machine learning as a service for high energy physics (MLaaS4HEP): a service for ML-based data analyses. (2023).

[32] Rong Gu, Simian Li, Haipeng Dai, Hancheng Wang, Yili Luo, Bin Fan, Ran Ben Basat, Ke Wang, Zhenyu Song, Shouwei Chen, et al. 2023. Adaptive online cache capacity optimization via lightweight working set size estimation at scale. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 467–484.

[33] Yu Gu, Yandan Guo, Yang Song, Xiangmin Zhou, and Ge Yu. 2018. Approximate order-sensitive K-NN queries over correlated high-dimensional data. *IEEE Transactions on Knowledge and Data Engineering* 30, 11 (2018), 2037–2050.

[34] Olivier Guédon and Emanuel Milman. 2011. Interpolating thin-shell and sharp large-deviation estimates for lsotropic log-concave measures. *Geometric and Functional Analysis* 21, 5 (2011), 1043–1068.

[35] Vivek Gupta, Akshat Shrivastava, Adithya Sagar, Armen Aghajanyan, and Denis Savenkov. 2021. Retronlu: Retrieval augmented task-oriented semantic parsing. *arXiv preprint arXiv:2109.10410* (2021).

[36] Erik Hämmerlund et al. 2014. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro* (2014). https://pages.cs.wisc.edu/~rajwar/papers/ieee_micro_haswell.pdf Peer-reviewed overview; discusses L1/L2 sizes/bandwidth and latency trends.

[37] Yikun Han, Chunjiang Liu, and Pengfei Wang. 2023. A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge. *arXiv preprint arXiv:2310.11703* (2023).

[38] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)* 28, 1 (1979), 100–108.

[39] Yupeng Hu, Chong Yang, Peng Zhan, Jia Zhao, Yujun Li, and Xueqing Li. 2021. Efficient continuous KNN join processing for real-time recommendation. *Personal and Ubiquitous Computing* 25 (2021), 1001–1011.

[40] Abiodun M Ikotun, Absalom E Ezugwu, Laith Abualigah, Belal Abuhaija, and Jia Heming. 2023. K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences* 622 (2023), 178–210.

[41] Intel. 2013. 4th Generation Intel Core Processor (Haswell). Hot Chips 25 Presentation. https://old.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.80-Processors2-epub/HC25.27.820-Haswell-Hammarlund-Intel.pdf Official vendor deck; generational cache/bandwidth context.

[42] Intel Corporation 2015. *Intel® Xeon® Processor E5-1600, E5-2600, and E5-4600 v3 Product Families: Datasheet, Volume 1.* Intel Corporation. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v3-datasheet-vol-1.pdf Haswell-EP generation; includes L1/L2/L3 cache specifications.

[43] Intel Corporation 2024. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation. https://cdrdv2-public.intel.com/671488/248966-Software-Optimization-Manual-V1-048.pdf Official optimization guide; cache hierarchy, bandwidth/latency guidance.

[44] Intel Corporation 2024. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C, 3D): System Programming Guide.* Intel Corporation. https://cdrdv2-public.intel.com/825749/325384-sdm-vol-3abcd.pdf Authoritative architectural reference; memory/cache behavior, MSRs.

[45] Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut Taylan Kandemir, et al. 2023. Optimizing cpu performance for recommendation systems at-scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[46] Moohanad Jawthari and Veronika Stoffová. 2021. Predicting students' academic performance using a modified kNN algorithm. *Pollack Periodica* 16, 3 (2021), 20–26.

[47] Haochen Ji, Zongyu Zuo, and Qing-Long Han. 2022. A divisive hierarchical clustering approach to hyperspectral band selection. *IEEE Transactions on Instrumentation and Measurement* 71 (2022), 1–12.

[48] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. 2002. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence* 24, 7 (2002), 881–892.

[49] Nora Kassner and Hinrich Schütze. 2020. BERT-kNN: Adding a kNN search component to pretrained language models for better QA. *arXiv preprint arXiv:2005.00766* (2020).

[50] Nida Khateeb and Muhammad Usman. 2017. Efficient heart disease prediction system using K-nearest neighbor classification technique. In *Proceedings of the international conference on big data and internet of thing*. 21–26.

[51] H Kheddar, Y Himeur, A Amira, and R Soualah. [n.d.]. Image classification in high-energy physics: A comprehensive survey of applications to jet analysis. *arXiv preprint arXiv:2403.11934* ([n. d.]).

[52] Shreya Kohli, Gracia Tabitha Godwin, and Siddhaling Urolagin. 2020. Sales prediction using linear and KNN regression. In *Advances in Machine Learning and Computational Intelligence: Proceedings of ICMLCI 2019*. Springer, 321–329.

[53] Roland Kühn, Jan Mühlig, and Jens Teubner. 2024. How to Be Fast and Not Furious: Looking Under the Hood of CPU Cache Prefetching. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. 1–10.

[54] Ajay Kumar, Nirav Patel, Nitin Gupta, and Vikas Gupta. 2022. L2 norm enabled adaptive LMS control for grid-connected photovoltaic converters. *IEEE Transactions on Industry Applications* 58, 4 (2022), 5328–5339.

[55] Monica D Lam, Edward E Rothberg, and Michael E Wolf. 1991. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review* 25, Special Issue (1991), 63–74.

[56] Michel Ledoux. 2001. *The concentration of measure phenomenon*. Number 89. American Mathematical Soc.

[57] Shengqiao Li. 2010. Concise formulas for the area and volume of a hyperspherical cap. *Asian Journal of Mathematics & Statistics* 4, 1 (2010), 66–70.

[58] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.

[59] Xuemei Li, Alexander Sigov, Leonid Ratkin, Leonid A Ivanov, and Ling Li. 2023. Artificial intelligence applications in finance: a survey. *Journal of Management Analytics* 10, 4 (2023), 676–692.

[60] Yang Li, Binxing Fang, Li Guo, and You Chen. 2007. Network anomaly detection based on TCM-KNN algorithm. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. 13–19.

[61] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S Berger, and Huaicheng Li. 2024. Dissecting cxl memory performance at scale: Analysis, modeling, and optimization. *arXiv preprint arXiv:2409.14317* (2024).

[62] Xiaoyang Lu, Hamed Najafi, Jason Liu, and Xian-He Sun. 2024. CHROME: Concurrency-aware holistic cache management framework with online reinforcement learning. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1154–1167.

[63] Fabian Mahling, Marcel Weisgut, and Tilmann Rabl. 2025. Fetch Me If You Can: Evaluating CPU Cache Prefetching and Its Reliability on High Latency Memory. In *Proceedings of the 21st International Workshop on Data Management on New Hardware*. 1–9.

[64] Sabyasachi Mohanty, Astha Mishra, and Ankur Saxena. 2020. Medical data analysis using machine learning with KNN. In *International Conference on Innovative Computing and Communications: Proceedings of ICICC 2020, Volume 2*. Springer, 473–485.

[65] Jin Ning, Leiting Chen, Chuan Zhou, and Yang Wen. 2018. Parameter k search strategy in outlier detection. *Pattern Recognition Letters* 112 (2018), 56–62.

[66] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of Vector Database Management Systems. *arXiv preprint arXiv:2310.14021* (2023).

[67] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *The VLDB Journal* 33, 5 (2024), 1591–1615.

[68] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector database management techniques and systems. In *Companion of the 2024 International Conference on Management of Data*. 597–604.

[69] Grigoris Paouris. 2006. Concentration of mass on convex bodies. *Geometric & Functional Analysis GAFA* 16, 5 (2006), 1021–1049.

[70] Punyaban Patel, Borra Sivaiah, and Riyam Patel. 2022. Approaches for finding optimal number of clusters using k-means and agglomerative hierarchical clustering techniques. In *2022 international conference on intelligent controller and computing for smart power (ICICCSP)*. IEEE, 1–6.

[71] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. 2023. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 557–573.

[72] Francesco Rundo, Francesca Trenta, Agatino Luigi Di Stallo, and Sebastiano Battiato. 2019. Machine learning for quantitative finance applications: A survey. *Applied Sciences* 9, 24 (2019), 5574.

[73] Bing Shi, Lixin Han, and Hong Yan. 2018. Adaptive clustering algorithm based on kNN and density. *Pattern Recognition Letters* 104 (2018), 37–44.

[74] Harsh Vardhan Singh, Ashwin Girdhar, and Sonika Dahiya. 2022. A Literature survey based on DBSCAN algorithms. In *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE, 751–758.

[75] Ananya Singhai, S Aanjankumar, and S Poonkuntran. 2023. A Novel Methodology for Credit Card Fraud Detection using KNN Dependent Machine Learning Methodology. In *2023 2nd International Conference on Applied Artificial Intelligence and Computing (ICAAIC)*. IEEE, 878–884.

[76] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment* (2014).

[77] Shahab Tayeb, Matin Pirouz, Johann Sun, Kaylee Hall, Andrew Chang, Jessica Li, Connor Song, Apoorva Chauhan, Michael Ferra, Theresa Sager, et al. 2017. Toward predicting medical conditions using k-nearest neighbors. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 3897–3903.

[78] Jing Tian, Michael H Azarian, and Michael Pecht. 2014. Anomaly detection using self-organizing maps-based k-nearest neighbor algorithm. In *PHM society European conference*, Vol. 2.

[79] Measuring Tweets. 2010. Twitter Official Blog [Web-page]. 22 feb. *Electronic resource https://blog. twitter. com/official/en_us/a/2010/measuring-tweets. html* (2010).

[80] Nimish Ukey, Zhengyi Yang, Wenke Yang, Binghao Li, and Runze Li. 2023. kNN Join for Dynamic High-Dimensional Data: A Parallel Approach. In *Australasian Database Conference*. Springer, 3–16.

[81] Nimish Ukey, Zhengyi Yang, Guangjian Zhang, Boge Liu, Binghao Li, and Wenjie Zhang. 2022. Efficient kNN Join over Dynamic High-Dimensional Data. In *Australasian Database Conference*. Springer, 63–75.

[82] Nimish Ukey, Guangjian Zhang, Zhengyi Yang, Binghao Li, Wei Li, and Wenjie Zhang. 2023. Efficient continuous kNN join over dynamic high-dimensional data. *World Wide Web* 26, 6 (2023), 3759–3794.

[83] Nimish Ukey, Guangjian Zhang, Zhengyi Yang, Xiaoyang Wang, Binghao Li, Serkan Saydam, and Wenjie Zhang. 2024. A Cluster-Based Approach to kNN Join Over Batch-Dynamic High-Dimensional Data. In *International Conference on Advanced Data Mining and Applications*. Springer, 81–96.

[84] Roman Vershynin. 2009. High-dimensional probability.

[85] René Vidal, Yi Ma, S Shankar Sastry, René Vidal, Yi Ma, and S Shankar Sastry. 2016. Principal component analysis. *Generalized principal component analysis* (2016), 25–62.

[86] Bingming Wang, Shi Ying, Guoli Cheng, Rui Wang, Zhe Yang, and Bo Dong. 2020. Log-based anomaly detection with the improved K-nearest neighbor. *International Journal of Software Engineering and Knowledge Engineering* 30, 02

(2020), 239–262.
[87] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment* 2, 1 (2009), 385–394.
[88] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.
[89] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.
[90] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jing Hu. 2004. Gorder: an efficient method for knn join processing. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 756–767.
[91] Wenchao Xing and Yilin Bei. 2019. Medical health big data classification based on KNN classification algorithm. *Ieee Access* 8 (2019), 28808–28819.
[92] Dian Xiong, Li Chen, Youhe Jiang, Dan Li, Shuai Wang, and Songtao Wang. 2024. Revisiting the Time Cost Model of AllReduce. *arXiv preprint arXiv:2409.04202* (2024).
[93] Chong Yang, Xiaohui Yu, and Yang Liu. 2014. Continuous KNN join processing for real-time recommendation. In *2014 IEEE International Conference on Data Mining*. IEEE, 640–649.
[94] Jingli Yang, Zhen Sun, and Yinsheng Chen. 2016. Fault detection using the clustering-kNN rule for gas sensor arrays. *Sensors* 16, 12 (2016), 2069.
[95] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. 2007. Efficient index-based KNN join processing for high-dimensional data. *Information and Software Technology* 49, 4 (2007), 332–344.
[96] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and HV Jagadish. 2001. Indexing the distance: An efficient method to knn processing. In *Vldb*, Vol. 1. 421–430.
[97] Cui Yu, Rui Zhang, Yaochun Huang, and Hui Xiong. 2010. High-dimensional knn joins with incremental updates. *Geoinformatica* 14 (2010), 55–82.
[98] Shichao Zhang, Xuelong Li, Ming Zong, Xiaofeng Zhu, and Ruili Wang. 2017. Efficient kNN classification with different numbers of nearest neighbors. *IEEE transactions on neural networks and learning systems* 29, 5 (2017), 1774–1785.
[99] Yi Zhang, Miaomiao Li, Siwei Wang, Sisi Dai, Lei Luo, En Zhu, Huiying Xu, Xinzhong Zhu, Chaoyun Yao, and Haoran Zhou. 2021. Gaussian mixture model clustering with incomplete data. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 17, 1s (2021), 1–14.
[100] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1979–1991.