Quake: Adaptive Indexing for Vector Search

Paper #870

Abstract

Vector search, the task of finding the k-nearest neighbors of high-dimensional vectors, underpins many machine learning applications, including recommendation systems and information retrieval. However, existing approximate nearest neighbor (ANN) methods perform poorly under dynamic, skewed workloads where data distributions evolve. We introduce Quake, an adaptive indexing system that maintains low latency and high recall in such environments. Quake employs a hierarchical partitioning scheme that adjusts to updates and changing access patterns, guided by a cost model that predicts query latency based on partition sizes and access frequencies. Quake also dynamically optimizes query execution parameters to meet recall targets using a novel recall estimation model. Furthermore, Quake utilizes optimized query processing, leveraging NUMA-aware parallelism for improved memory bandwidth utilization. To evaluate Quake, we prepare a Wikipedia vector search workload and develop a workload generator to create vector search workloads with configurable access patterns. Our evaluation shows that on dynamic workloads, Quake achieves query latency reductions of 1.5–22× and update latency reductions of 6–83× compared to state-of-the-art indexes SVS, DiskANN, HNSW, and SCANN.

1 Introduction

Vector search, the task of finding the k-nearest neighbors (KNN) of high-dimensional vectors, is fundamental to machine learning applications such as recommendation systems [20,21,27,28,37] and information retrieval [9,12,13,30]. In these applications, each vector represents an item in a metric space, and the distance between vectors reflects semantic similarity. However, performing exact KNN search becomes computationally infeasible on large datasets due to the high dimensionality and volume of data.

To address this challenge, practitioners use approximate nearest neighbor (ANN) indexes, which trade off a controlled amount of search accuracy (recall) for significant reductions in latency. Among these indexes, there are two widely-used types: graph-based indexes and partitioned indexes.

Maintaining low latency, high recall vector search under **dynamic and skewed workloads** remains a significant challenge for existing indexes. Real-world applications often exhibit non-uniform query distributions and evolving data. For example, in an example Wikipedia search application, popular pages like *Lionel Messi* or *LeBron James* receive disproportionately more queries, resulting in *skewed read patterns*. Additionally, pages are frequently added, updated, or deleted, causing *skewed update patterns* that change over time [6]. These factors degrade the performance of existing indexes, leading to increased query latency and reduced recall.

Graph-based indexes, such as HNSW [22] and DiskANN [32,33] construct a proximity graph where each node (vector) is connected to its approximate neighbors. Queries traverse the graph to find approximate nearest neighbors, typically achieving high recall with low latency. However, these indexes face challenges with dynamic workloads because updating the graph structure to accommodate frequent insertions and deletions is computationally intensive [39], due to the random access patterns involved in graph traversal and modification.

Partitioned indexes, such as SCANN [10, 34], SPANN [7,39], and Faiss-IVF [8], partition the vectors using a clustering algorithm (e.g k-means). Queries are processed by scanning a subset of partitions, balancing recall and latency by adjusting the number of partitions scanned (denoted as *nprobe*). While attractive due to their simplicity, partitioned indexes face a significant search latency gap when compared with graph indexes. For example, on the MSTuring10M benchmark [2], we found Faiss-IVF takes 44ms per search query while Faiss-HNSW takes only 6.8ms. On the other hand, supporting updates in partitioned indexes is less expensive than for graph indexes, as the index structure needs minimal modification when adding or removing vectors. But, existing approaches struggle with dynamic and skewed workloads because they do not adapt to changing access patterns, leading to *imbalanced* partitions that degrade query latency. Recent work has been proposed to resolve imbalances in dynamic workloads by splitting and reclustering imbalanced partitions [6,39], however, we find these methods degrade recall as nprobe needs to change as the index structure changes.

In this work, we study the problem of minimizing query latency to meet a fixed recall target for dynamic vector search workloads, where both the queries and the base vectors can change over time. To address this problem, we develop **Quake**, a partitioned index system that minimizes latency by adapting the index structure to the workload. Quake introduces the following contributions:

First, Quake employs an **adaptive hierarchical partitioning** scheme that modifies the partitioning by minimizing the cost (derived from a **cost model**) of a query. The cost model tracks partition sizes and access frequencies as the workload is processed and determines which partitions are most negatively contributing to overall query latency. Once identified, we conduct maintenance on them by splitting or merging them based on what is expected to reduce the cost (latency). We also demonstrate our maintenance procedure is stable and

converges to a local minimum of the cost model.

Second, we design an **adaptive partition scanning** scheme that adjusts the number of partitions scanned on-the-fly to meet recall target for individual queries. We do this by maintaining a recall estimate during query processing based on the a) geometry of the partitioning and b) intermediate results of the query, and once the estimate exceeds the recall target, query processing terminates and the results are returned.

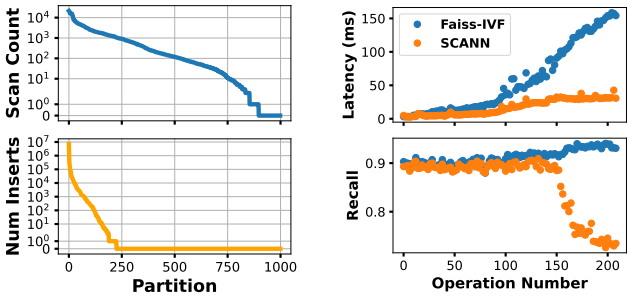
Third, in order to close the performance gap with graph indexes, Quake utilizes **NUMA-aware intra-query parallelism** in order to maximize memory bandwidth used on multi-core machines. Maximizing memory bandwidth allows Quake to minimize query latency without impacting recall.

It is a significant challenge to evaluate indexing approaches due to the lack of availability of benchmarks for online vector search. To address this challenge and comprehensively evaluate our approach, we A) prepare a **Wikipedia vector search workload** derived from publicly available query and update patterns of Wikipedia pages and B) develop a **workload generator** for creating workloads with configurable query and update patterns. We will publicly release the Wikipedia workload and workload generator as evaluation tools for the community to use. Using this, we conduct a comprehensive evaluation of Quake in comparison to seven baseline approaches.

1. Quake achieves the lowest search latency across all dynamic workloads in comparison to state-of-the-art graph indexes, with $1.5-8\times$ lower search latency than HNSW and DiskANN while having $6-83\times$ lower update latency.
2. We also find that APS Scanning matches the nprobe of an oracle across recall targets on Sift1M, with only a 20% increase in latency relative to the oracle. In addition, APS exhibits stable recall on the Wikipedia-12M workload, with only a .005% average deviation from a recall target of 90% over the course of the workload.
3. Quake's NUMA-aware intra-query parallelism exhibits linear scalability and saturates memory bandwidth on the MSTuring100M dataset with 100-million records. Quake achieves $16\times$ lower query latency when compared to a single-threaded version and $4\times$ lower latency compared to a non-NUMA aware configuration.

2 Motivation and Challenges

Efficient vector search is critical for large-scale systems used in recommendation, semantic search, and information retrieval. These applications demand the ability to process a high volume of nearest neighbor queries with low latency, even as the underlying data evolves. To meet these requirements, vector databases—such as Milvus [36], Pinecone [3], Analytic-DBV [38], VBASE [41], and Qdrant [1]—utilize specialized vector indexes (e.g., Faiss-IVF, HNSW, Vamana) that support fast approximate nearest neighbor (ANN) queries. However, serving these dynamic workloads introduces significant challenges in maintaining query performance and accuracy as data and query patterns shift over time.



(a) Read (top) and write skew. (b) Query performance.

Figure 1: Skewed access patterns of Faiss-IVF index partitions in the Wikipedia-12M workload and their effect on query performance for Faiss-IVF and SCANN

2.1 Vector Search Workload

A vector search workload is a continuous, evolving stream of **queries** and **updates**:

- **Queries:** Given a query vector q, the goal is to find the top-k nearest neighbors in a set $\mathbf{X}$. Exact linear search is too slow for large, high-dimensional datasets, so ANN indexes are used. These indexes approximate nearest neighbors with controlled recall to lower latency by orders of magnitude.
- **Updates:** The dataset evolves over time. Insertions add new vectors representing fresh content (e.g., new products, trending news articles), and deletions remove outdated entries. Typically, updates are applied in a batched fashion.

Recall@k is the standard metric for accuracy, defined as: $\frac{|\mathbf{G}\cap\mathbf{R}|}{k}$ where $\mathbf{R}$ is the vectors returned by the approximate search, and $\mathbf{G}$ is the ground truth set. Maintaining a consistent recall target (e.g., > 0.9) and low latency (e.g., milliseconds per query) as both data and query patterns shift is a key challenge. The complexity of these workloads stems from their inherently dynamic and skewed nature, which few existing indexing methods handle gracefully.

2.2 Why Real-World Workloads are Hard

Skewed Read Patterns In practice, user queries concentrate on popular items. For example, queries against a Wikipedia-derived dataset tend to focus on a small subset of entities at any given time. As a result, certain partitions or graph regions of the index are accessed disproportionately often.

Skewed Write Patterns Insertions and deletions are also rarely uniform. New data often arrives in bursts—e.g., new Wikipedia pages added monthly, new products introduced ahead of a shopping season, or newly relevant embeddings generated by continuously updated language models.

Real-World Example: Wikipedia-12M In our evaluation, we prepared Wikipedia-12M, a workload based on a subset of Wikipedia articles derived from publicly available monthly

Table 1: Comparison of updatable vector indexes. **Tuning**: Requires manual parameter tuning in indexing and query processing. **Maintenance**: Restructures index with incremental updates. **Adaptive**: Utilizes query information to inform indexing. **NUMA**: NUMA-aware execution.

Method	Tuning	Maint.	Adaptive	NUMA
Quake **(Ours)**	✗	✓	✓	✓
Faiss-IVF [8]	✓	✗	✗	✗
DeDrift [6]	✓	✓	✗	✗
SpFresh [39]	✓	✓	✗	✗
SCANN [10, 34]	✓	✓	✗	✗
DiskANN [26, 32]	✓	✓	✗	✗
Faiss-HNSW [22]	✓	✗	✗	✗
SVS [5]	✓	✓	✗	✗

pageview statistics [4]. Over 103 months, the dataset grows from millions to tens of millions of vectors. Popular articles dominate query traffic, while embeddings of newly created pages accumulate in certain regions of the embedding space. This workload shows read skew and write skew, as evidenced by Figure 1a, reads and writes predominately affect a small portion of the index.

2.3 Shortcomings of Existing Approaches

Existing indexes were often developed and evaluated under assumptions of static data distributions; conditions not met in real-world use cases. Table 1 compares a range of well-known and state-of-the-art vector indexes. Although they are widely adopted in vector databases, none fully solve the problem of maintaining low-latency, high-recall search under dynamic, skewed workloads without constant manual intervention or offline tuning.

Graph Indexes Graph-based index systems, such as HNSW [22], DiskANN [26, 32], and SVS [5] construct a proximity graph where each node represents a vector connected to its approximate neighbors. These indexes achieve high recall with low latency in static settings by efficiently traversing the graph to locate nearest neighbors using a process known as *greedy traversal*. However, maintaining the graph structure under frequent insertions and deletions is computationally intensive, as each update may require rewiring multiple edges to preserve graph connectivity and proximity properties. Our evaluation (Table 1) shows that update latency can be multiple orders of magnitude higher than partitioned indexes. Additionally, graph traversal involves random memory accesses, leading to poor memory bandwidth utilization.

Partitioned Indexes Partitioned indexes such as Faiss-IVF [8], SCANN [10, 34], and SpFresh [39] divide the vector space into disjoint partitions using a clustering algorithm such as k-means. Queries are processed by scanning a subset of partitions to retrieve approximate nearest neighbors. Partitioned indexes are more update-friendly than graph-based method since insertions and deletions leads to sequential access. For write skewed workloads some partitions become significantly larger, degrading query latency, this can be exacerbated by read skew if large partitions are also more frequently accessed ("hot partitions"). Query processing is *memory-bound*, as achieving high recall requires scanning many megabytes of data across multiple partitions, for example reaching a recall target of 90% on the MSTuring100M dataset requires each query to scan 1GB of vectors. Moreover, most partitioned indexes use a fixed number of partitions to probe (*nprobe*), which does not adapt to changing data distributions or query patterns, leading to either insufficient recall or excessive data scanning. The challenges yield subpar performance for partitioned indexes on real-world workloads. For example, Figure 1b shows the degradation of latency and recall over time when using Faiss-IVF and SCANN with a fixed nprobe on Wikipedia-12M (workload details in Section 7).

2.4 Technical Challenges for Partitioned Indexes

The following technical challenges have yet to be solved by existing partitioned indexes

1. **Adaptation to Queries** Query adaptivity is overlooked by existing partitioned index approaches and exhibits an opportunity for optimization, particularly for maintaining hot partitions induced by read skew.
2. **Online Adjustment of Nprobe** As the index structure and data changes, partitioned indexes need to adjust the number of partitions scanned otherwise recall will suffer.
3. **Performance Gap with Graph Indexes** Standard partitioned indexes such as Faiss-IVF are memory bound, and exhibit an order of magnitude higher query latency in comparison than graph indexes.

Quake is our solution to these technical challenges. Quake A) adapts the index structure to queries by utilizing maintenance that minimizes a cost model for query latency, B) using a recall estimation model, Quake individually sets nprobe for queries to meet recall targets as the index structure changes, and C) uses NUMA-aware parallelism in order to saturate memory bandwidth during query processing, closing the performance gap with graph indexes. We next describe Quake in detail.

3 Solution Overview

Index Structure In Quake, vectors are partitioned into disjoint partitions (using k-means) where each partition has a representative centroid. Centroids can be further partitioned in a similar manner to add additional levels to the index. Search queries scan the index structure in a top-down fashion, finding the nearest centroids to determine the partitions the scan in the next level. Partitions in the bottom level contain the base vectors and subsets of these partitions are scanned to return the k-nearest neighbors. Utilizing a multi-level design mitigates the cost of scanning centroids, allowing us to use fine-grained partitioning of vectors at large scale, which prior

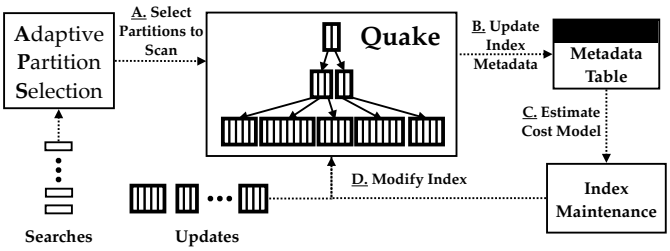


Figure 2: Quake Architecture Diagram. Search queries use Adaptive Partition Selection (APS) to determine which partitions to scan (**A**). Scanning partitions modifies access patterns of the index, tracked in the metadata table (**B**). A cost model is used to determine which maintenance actions to take (**C**) where the chosen maintenance actions modify the index (**D**). This process operates in a continuous online fashion as search and update queries (inserts/deletes) are issued to the index.

work has shown to be beneficial [7].

Adaptive Incremental Maintenance Inserts and deletes modify the Quake data structure by appending vectors to and removing vectors from index partitions. Insertions traverse the index structure top-down to find the nearest partition in the base level to the inserted vector and append to that partition. Deletes use a map to find the partition containing the vector to be deleted and the vector is removed from the partition with immediate compaction. As demonstrated in Section 2, modifications can negatively affect index performance over time, requiring maintenance (Figure 1b). Quake uses the following maintenance actions in order to minimize query latency:

1. **Split Partition:** Uses k-means to split a partition into two, removing the old partition and its centroid and adding two new partitions and centroids. To mitigate potential overlap due to the new partitions, we perform additional iterations of k-means over the partitions neighboring the split partitions (by centroid distance).
2. **Delete Partition:** Removes a partition and its centroid, reassigning the vectors of the deleted partition to the remaining partitions the index.
3. **Add Level:** Adds a level of partitioning to the index by partitioning the current top-level using k-means.
4. **Delete Level:** Removes current top-level and merges the partitions in the next level.

Quake uses a cost model that estimates query latency to determine if maintenance actions should be taken and which partitions to apply them to. The cost model is a function of partition access patterns and sizes to determine which partitions are most contributing to the overall query latency. We check for maintenance after each operation by evaluating the cost model, but the maintenance frequency is configurable. Partitions with the largest cost contribution are considered for split or deletion. Intuitively, frequently accessed and/or large partitions are split and infrequently accessed and/or small partitions are deleted as they do not justify the overhead of maintaining a centroid. See Section 4 for details on the cost model and maintenance methodology.

Adaptive Partition Scanning In order to determine the number of partitions a search query should scan to reach a given recall target, we apply *Adaptive Partition Scanning* (APS) at each level of the index. APS solves a critical problem for partitioned indexes when applied to dynamic workloads: as the number and contents of partitions change, the number of partitions scanned needs to change, otherwise recall will degrade (Figure 1b). APS maintains a recall estimator based on the intermediate top-k results of the query and the geometry of neighboring partitions. As more partitions are scanned, the intermediate results and recall model are updated and when the recall estimate exceeds the target recall, the results are returned. To mitigate overheads introduced by the recall estimator, we use pre-computation of expensive geometric functions and only update the estimate when the intermediate results have changed significantly. APS supports both euclidean and inner-product distance metrics. We cover APS in Section 5.

NUMA-Aware Query Processing Modern multi-core servers often use Non-Uniform Memory Access (NUMA) architectures, where memory close to a processor's local node is faster to access than remote memory. Quake is designed to capitalize on this heterogeneous memory. It distributes index partitions across NUMA nodes. To minimize remote memory access, Quake employs affinity-based scheduling, and supports work stealing within a NUMA node to mitigate workload imbalances. By co-locating computation with the relevant data, Quake reduces remote memory accesses, saturates memory bandwidth, and thus lowers query latency. See Section 6 for further details on Quake's NUMA-aware optimizations.

4 Adaptive Incremental Maintenance

We detail our maintenance methodology, beginning with the cost model for estimating query latency, which guides maintenance decisions. We then describe how each maintenance action modifies the index state and affects cost, which is used to prioritize actions based on their expected benefit.

4.1 Cost Model

The cost model estimates the query latency contributed by each partition, aggregated across all levels. This model guides maintenance actions.

Partition Properties: Consider an index with L levels, numbered $l = 0, 1, \ldots, L-1$. Level l contains N_l partitions. The base level corresponds to $l = 0$ and contains partitions of the original dataset vectors. Higher levels contain partitions of centroid vectors that summarize the partitions in the level below. At the top level, $l = L-1$, there is a single partition containing top-level centroids.

Each partition j at level l has a size s_{lj} (the number of vectors it contains) and an access frequency $A_{l,j} \in [0.0, 1.0]$. $A_{l,j}$ denotes the fraction of queries, measured in a sliding window W, that scan the partition j at level l. The cost model is primarily driven by these sizes and access frequencies.

Partition Cost A partition (l, j) contributes latency proportional to its size and how frequently it is accessed. Let $\lambda(s)$ be the latency function for scanning s vectors. This is often linear, e.g. $\lambda(s) = a \cdot s + b$. We measure $\lambda(s)$ through offline profiling. The cost of partition (l, j) is:

$$C_{lj} = A_{lj} \cdot \lambda(s_{lj}) \tag{1}$$

Total Cost: The overall query latency (cost) estimate is the sum across all levels and partitions

$$C = \sum_{l=0}^{L-1} \sum_{j=0}^{N_l - 1} A_{lj} \cdot \lambda(s_{lj}) \tag{2}$$

Interpretation The cost model reflects the relationship between partition size, access frequency, and query latency. The fundamental trade-off that needs to be balanced is the number and size of partitions. Larger partitions require more time to scan, increasing latency, but reducing the total number of partitions and the overhead of scanning centroids. Conversely, smaller, fine-grained partitions reduce the number of vectors needed to scan to reach a high recall but increase the overhead of scanning centroids. The model further captures that frequently accessed partitions dominate the total cost, motivating targeted maintenance actions to balance these trade-offs.

Guiding Maintenance Decisions Maintenance actions such as splitting or deleting aim to reduce the total cost C. Each action is evaluated based on its predicted change in cost:

$$\Delta C = C_{\text{after}} - C_{\text{before}} \tag{3}$$

where C_{before} and C_{after} are the total costs before and after the action, respectively. Actions are applied only if $\Delta C < -\tau$, where τ is a non-negative tunable threshold, ensuring monotonic improvement in query performance. By focusing on reducing C, the index is dynamically restructured to maintain efficient query performance under varying workloads.

4.2 Maintenance Actions

To minimize query latency, Quake employs a series of maintenance actions that dynamically adjust the index structure in response to evolving workloads. Here we first define the index state, and then analyze the impact of each maintenance action on the overall cost model. Due to space constraints, we defer the analysis of adding and removing levels in the supplementary material, but note that the same principles apply to those actions.

Index State: The Quake index is a hierarchy of levels. Each level l contains N_l partitions $P_{l0}, P_{l1}, \ldots, P_{lN_l-1}$. Each partition has a centroid and contains a subset of vectors (or centroid vectors in higher levels). The index state is defined by the number of partitions per level, and their sizes s_{lj} and access frequencies A_{lj}. Queries and updates affect these attributes, while maintenance actions restructure the hierarchy.

4.2.1 Splitting and Deleting Partitions

Split Partition: If a partition (l, j) is too large or frequently accessed, we consider splitting it into two partitions (l, j_L) and (l, j_R). We apply k-means clustering within that partition, forming two smaller partitions with their own centroids. The original partition is removed and its vectors are reassigned. A subsequent *partition refinement* step adjusts vector assignments to ensure minimal overlap and balanced partition sizes.

Partition Refinement After a split, refinement uses k-means (seeded by current centroids) on nearby partitions to mitigate overlap and ensure that each vector is assigned to its most representative partition. Nearby partitions are determined by finding the r nearest centroids to the split centroids, where r is a tunable parameter (typically between 10 and 100). This is a generalization of the reassignment procedure used in SpFresh [39], using additional rounds of k-means prior to reassignment. Refinement ensures vectors are correctly assigned to their most representative partition and avoids performance degradation due to excessive overlap.

Delete Partition If a partition is rarely accessed and below a minimum size threshold, we consider deleting it to remove the cost of maintaining its centroid. After deletion, the vectors are reassigned to their respective nearest existing partitions. This can reduce total cost by removing a low-benefit partition, although the reassignment may increase the size (and thus cost) of other partitions, and therefore careful consideration is needed before conducting a delete.

Cost Change of Splitting and Deleting Consider splitting a partition (l, j) into two partitions (l, j_L) and (l, j_R). Let ΔO^+ be the cost change from adding a centroid at the parent level $(l-1)$. For a single-level index: $\Delta O^+ = \lambda(N_l + 1) - \lambda(N_l)$. The cost delta from splitting is:

$$\Delta Split_{lj} = \Delta O^+ - A_{lj}\lambda(s_{lj}) + A_{lj_L}\lambda(s_{lj_L}) + A_{lj_R}\lambda(s_{lj_R}) \tag{4}$$

For deleting a partition (l, j), let ΔO^- be the overhead change from removing a centroid at level $(l-1)$: $\Delta O^- = \lambda(N_l - 1) - \lambda(N_l)$. If $R_{l,j}$ is the set of partitions that receive vectors from the deleted partition, and for each $m \in R_{lj}$ the size and access frequency change by Δs_m and ΔA_m, the cost delta from deleting is:

$$\begin{aligned}\Delta Del_{lj} = \Delta O^- - A_{lj}\lambda(s_{lj}) \\ + \sum_{m \in R_{lj}} \left[(A_m + \Delta A_m)\lambda(s_m + \Delta s_m) - A_m\lambda(s_m)\right]\end{aligned} \tag{5}$$

4.2.2 When to Take Action

Given the above actions and their effect on the cost, we now show how to use this information to decide which maintenance action to take. The main steps taken when conducting maintenance are:

1. Estimate $\Delta Split_{lj}$ and ΔDel_{lj} for all partitions.
2. Perform actions expected to reduce the cost beyond a threshold τ.
3. Use rejection mechanisms to verify that the actual cost change remains below $-\tau$.
4. Commit actions that pass rejection tests.
5. Estimate $\Delta AddLevel$ and $\Delta DelLevel$; apply the action if beyond $-\tau$ (See supplementary material)

When evaluating maintenance actions such as splitting or deleting partitions, we face a fundamental challenge: certain cost components remain unknown until after the action is performed and new statistics are collected. For example, upon splitting a partition, we do not know the exact resulting partition sizes or their new access frequencies until the split has been finalized and the system has observed queries accessing these new partitions. Similarly, when deleting a partition, the precise reassignments of its vectors to neighboring partitions are not fully determined until the deletion is completed, and their subsequent access frequencies remain uncertain until these changes are observed in practice.

Estimates and Assumptions for Splitting Consider a partition (lj) that we propose to split into two partitions, (lj_L) and (lj_R). Our goal is to estimate the change in cost, $\Delta Split_{lj}$, without knowing the exact sizes s_{lj_L}, s_{lj_R} or access frequencies A_{lj_L}, A_{lj_R} beforehand. We make two simplifying assumptions: 1) **Balanced Split:** We assume that the split partitions will be of approximately equal size, i.e., $s_{lj_L} \approx s_{lj_R} \approx \frac{s_{lj}}{2}$. 2) **Proportional Access Frequency Scaling:** We assume that the total access frequency of the original partition is split proportionally between the two new partitions. We introduce a uniform scaling factor α to model how the access frequency distributes between the resulting partitions. Under such assumptions, the delete estimate for splitting is:

$$\Delta Split'_{lj} = \Delta O^{+} - A_{lj}\lambda(s_{lj}) + 2\alpha A_{lj}\lambda\left(\frac{s_{lj}}{2}\right) \tag{6}$$

Here, ΔO^{+} captures the overhead change from adding a centroid at the parent level. This estimate is used to make a preliminary decision about whether to attempt a split. After the split is performed, we know the sizes of the resulting partitions and can utilize the split rejection mechanism to decide whether to commit the split or not.

Estimates and Assumptions for Deletion Estimating the cost delta for deleting a partition (l, j) is also challenging. Prior to deletion, we do not know: A) The exact reassignments Δs_{lm} of vectors from the deleted partition to neighboring partitions (l, m). B) The changes in access frequencies ΔA_{lm} that occur after the reassignment, since these frequencies can only be measured once queries access the updated partitions.

To handle this uncertainty, we assume that the vectors from the deleted partition are *uniformly distributed* among the candidate neighboring partitions. This uniform distribution assumption provides a rough estimate of how much each neighboring partition grows and how their access frequencies may adjust. Thus, we estimate:

$$\Delta Del'_{l,j} = \Delta O^{-} - A_{lj}\lambda(s_{lj}) + \sum_{m \in R_{lj}} (A_{lm} + \frac{A_{lj}}{|R_{lj}|})\lambda(s_{lm} + \frac{s_{lj}}{|R_{lj}|})$$

where $|R_{lj}|$ is the number of receiving partitions. Like the split estimate, the deletion estimate only guides the initial decision.

Rejection Mechanisms Because both $\Delta Split'_{lj}$ and $\Delta Del'_{lj}$ are derived from assumptions and estimates, the actual improvements may differ once the action is tentatively performed. To safeguard against detrimental actions, we implement rejection mechanisms that re-evaluate the cost change using the newly known quantities (e.g., final partition sizes after a split or actual reassignments after a deletion) while maintaining the same frequency assumptions used in the initial estimates.

1. **Split Rejection:** After performing the split, we immediately know the resulting partition sizes s_{lj_L} and s_{lj_R}, we recalculate the cost delta using these now-known sizes and the same proportional frequency scaling factor α assumed previously. If the re-calculated $\Delta Split_{lj}$, under these assumptions, is not less than $-\tau$, we reject the action and do not commit the split.
2. **Delete Rejection:** Prior to deletion, we compute the reassignments and Δs_{lm} for neighboring partitions. Using this, we re-compute ΔDel_{lj} and if the delta is not less than $-\tau$, we reject the deletion and do not commit it.

These rejection mechanisms ensure that only actions that still appear beneficial once their immediate known consequences (such as final partition sizes or reassignment distributions) are taken into account will be retained. They do not rely on post-action query statistics; instead, they use the same frequency assumptions as the initial estimates, but with updated, now-known partition sizes and vector assignments.

By combining initial estimates, threshold-based acceptance, and rejection mechanisms that rely solely on known partition characteristics, our incremental maintenance process converges toward an efficient steady state under fixed query distributions. We analyze the convergence of adaptive incremental maintenance in the supplementary material.

5 Adaptive Partition Scanning (APS)

This section introduces Adaptive Partition Scanning (APS), a technique used in Quake to dynamically adjust the number of partitions (clusters) scanned to achieve a target recall while minimizing per-query latency. Instead of fixing *nprobe a priori*, APS estimates the probability that a partition contains the query's nearest neighbors and uses it to decide when to stop scanning additional partitions.

To achieve this, APS approximates the probability of finding nearest neighbors within a given partition by using a geo-

metric model. We treat the query as defining a *query hypersphere*—a hypersphere centered at the query point with a radius equal to the current k-th nearest neighbor distance—and the partitions as regions defined by Voronoi cells around their centroids. The key idea here is that the *intersection volume* of the query hypersphere with a partition provides a proxy for the likelihood of that partition containing nearest neighbors. By accumulating these probabilities as partitions are scanned, APS maintains a running estimate of query recall and stops scanning once the target recall is reached.

For ease of exposition, we focus primarily on Euclidean (L2) distance, but we also show how to extend APS to inner product and angular similarity metrics. We also present performance optimizations that reduce the overhead of dynamically computing recall estimates. The notation used in this section is summarized in Table 2. Throughout this section, we assume a single-level index structure to simplify the notation. In the multi-level setting, APS is applied individually to each level.

Table 2: Notation Summary

Symbol	Description
d	Dimensionality of the vector space
$\mathbf{q}$	Query vector in $\mathbb{R}^d$
k	Number of nearest neighbors to retrieve
ρ	Query radius (dist. to current k-th nearest neighbor)
K	Total number of partitions (clusters)
$\mathcal{P}$	Set of all partitions
$\mathcal{C}$	Set of all centroids
$\mathcal{P}_i$	The i-th partition
$\mathbf{c}_i$	Centroid of partition $\mathcal{P}_i$
h_i	Boundary distance from $\mathbf{q}$ to boundary of $\mathcal{P}_i$
v_i	Intersection volume of query hypersphere and $\mathcal{P}_i$
$\hat{v}_i$	Normalized intersection volume for $\mathcal{P}_i$
R_i	Estimated recall after scanning i partitions
τ_R	Target recall

5.1 Recall Estimation

The dataset of d-dimensional vectors is partitioned into K Voronoi cells (partitions) $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{K-1}$, each with a centroid $\mathbf{c}_i \in \mathbb{R}^d$. Given a query vector $\mathbf{q} \in \mathbb{R}^d$ and a current query radius ρ, we want to estimate how likely it is to find the query's nearest neighbors in each partition. We approximate this by considering the volume of intersection between the *query hypersphere* (centered at $\mathbf{q}$ with radius ρ) and the half-space defined by each partition boundary. Figure 3 illustrates a query hypersphere intersecting two partition boundaries.

Boundary Distance Computation Consider the boundary between the nearest partition $\mathcal{P}_0$ (with centroid $\mathbf{c}_0$) and a neighboring partition $\mathcal{P}_i$ (with centroid $\mathbf{c}_i$): it defines the hyperplane equidistant from $\mathbf{c}_0$ and $\mathbf{c}_i$. We define the *boundary distance* h_i as the distance from $\mathbf{q}$ to this boundary hyperplane:

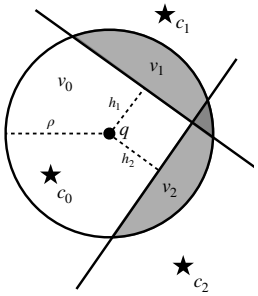


Figure 3: The query hypersphere (centered at $\mathbf{q}$ with radius ρ) intersecting partition boundaries. The intersection volumes v_1 and v_2 correspond to the probability of finding a nearest neighbor in partitions $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively.

$$h_i = \frac{(\mathbf{c}_i - \mathbf{c}_0)^\top (\mathbf{q} - \mathbf{m}_i)}{|\mathbf{c}_i - \mathbf{c}_0|}, \tag{7}$$

where $\mathbf{m}_i = \frac{1}{2}(\mathbf{c}_0 + \mathbf{c}_i)$ is the midpoint between the centroids $\mathbf{c}_0$ and $\mathbf{c}_i$, and $|\cdot|$ is the Euclidean norm. Intuitively, h_i tells us how far $\mathbf{q}$ is inside (or outside) a given partition region relative to its nearest centroid.

Intersection Volume Estimation Given h_i, we now estimate the intersection volume v_i between the query hypersphere of radius ρ and the half-space beyond the boundary hyperplane toward $\mathcal{P}_i$. This intersection volume provides a geometric proxy for the probability that partition $\mathcal{P}_i$ contains at least one of the k nearest neighbors. The volume of the hyperspherical cap formed by cutting a hypersphere at distance h_i from its center is given by [18]:

$$v_i = \frac{1}{2} I\left(1 - \left(\frac{h_i}{\rho}\right)^2; \tfrac{d}{2}, \tfrac{1}{2}\right), \tag{8}$$

where $I(x; a, b)$ is the regularized incomplete beta function. We normalize these volumes to obtain $\hat{v}_i$:

$$\hat{v}_i = \frac{v_i}{\sum_j v_j}. \tag{9}$$

The normalized volumes $\hat{v}_i$ sum to 1 and can be interpreted as relative probabilities that each partition contains the query's nearest neighbors. By scanning partitions in descending order, we can quickly accumulate a large portion of the recall. After scanning m partitions, the cumulative estimated recall is:

$$R_m = \sum_{j=0}^{m} \hat{v}_j. \tag{10}$$

Once $R_m \geq \tau_R$, the target recall τ_R is considered met, and we can stop scanning further partitions.

Extension to Other Metrics While we focus on Euclidean distance for clarity, APS extends naturally to inner product and angular similarity metrics. By normalizing all vectors to unit length, an inner product query is transformed into an angular distance one. The geometric interpretation then shifts to a unit hypersphere $\mathbb{S}^d$, where intersection volumes become spherical caps on the unit sphere's surface [16]. The boundary distance h_i is replaced by an angular distance between the query direction and the partition boundary.

5.2 APS Algorithm

Algorithm 1 outlines the APS procedure. The algorithm takes as input the query vector $\mathbf{q}$, a target recall τ_R, an initial number of partitions to scan, n_{probe}, the set of centroids $\mathcal{C}$, and the desired number of nearest neighbors k. It returns the top-k nearest neighbors after adaptively scanning partitions until the target recall is reached.

Algorithm 1 Adaptive Partition Scanning (APS)

Input: Query $\mathbf{q}$, target recall τ_R, re-computation threshold τ_ρ, initial n_{probe}, centroids $\mathcal{C}$, number of neighbors k
Output: Top-k nearest neighbors

```
Initialize a top-k priority queue B
Retrieve an initial set of n_probe candidate partitions P_init closest to q
Find the nearest partition P_0, scan it, and update B and ρ
for each partition Pi in Pinit do
    Compute h_i using Equation (7)
    Compute v_i using Equation (8)
Normalize v_i to get v̂_i using Equation (9)
Compute initial R_m using Equation (10)
m ← 0
while R_m < τ_R do
    Select Pi = arg max P_j unscanned v̂_j
    Scan P_i and update B and ρ
    m ← m + 1
    if ρ changes beyond τ_ρ then
        Recompute v_j and v̂_j for unscanned partitions
        Update R_m accordingly
return top-k nearest neighbors from B
```

In practice, we begin by identifying a small set of candidate partitions (retrieved by the parent index) and scanning the closest one to find an initial set of neighbors and establish the query radius ρ. Using ρ, we estimate volumes v_i and compute $\hat{v}_i$ and R_m. While the cumulative recall estimate R_m is below the target τ_R, we iteratively select and scan the partition with the largest $\hat{v}_i$. Each scan potentially updates the query radius ρ if closer neighbors are found. When ρ changes significantly, we update the volume estimates and recalculate R_m. This adaptive process ensures that we do not waste time scanning partitions that are unlikely to contain nearest neighbors.

Performance Optimizations To minimize overhead, we adopt two key optimizations. First, we precompute values of the regularized incomplete beta function $I(x;a,b)$ for a range of x and store them in a lookup table, enabling fast volume estimation at runtime. Second, we only recompute intersection volumes when ρ changes beyond a predetermined τ_ρ. This avoids frequent recalculations when ρ remains stable, reducing unnecessary overhead in adaptive scanning. These optimizations are evaluated in Table 3, demonstrating that enabling the above optimizations with APS reduces query latency by 25% on Sift1M with no impact on recall.

Table 3: Mean single-threaded query latency and recall for APS on Sift1M with and without optimizations for a recall target of .9. APS-RP recomputes the recall model after each partition scan without precomputation of $I(x;a,b)$, APS-R computes the recall model after each partition scan with precomputation of $I(x;a,b)$, APS computes the recall model if ρ has changed beyond $\tau_\rho = 1\%$ and uses precomputation.

Configuration	Recall	Search Latency
APS	.901	.52 ms
APS-R	.901	.61 ms
APS-RP	.901	.70 ms

6 Quake Implementation

Here we discuss NUMA-aware query processing and implementation details of Quake.

NUMA Data Placement and Query Processing Query processing in partitioned vector indexes is memory-bound, and therefore increasing the effective memory bandwidth available to the system will reduce query latency. NUMA-aware intra-query parallelism has been applied in the context of relational database systems to great success [17, 29], but has yet to be applied to vector databases.

In order to maximize memory bandwidth utilization, Quake distributes index partitions across NUMA nodes and ensures that cores only scan partitions resident in their respective node. Quake assigns index partitions to specific NUMA nodes using round-robin assignment. This assignment procedure allows for simple load balancing as partitions are added to the index by the maintenance procedure.

To maximize the benefits of data placement, Quake employs partition affinity and NUMA-aware work scheduling. Partitions are bound to specific CPU cores. This binding ensures that partitions are always scanned by the same core to maximize cache utilization. Queries are scheduled to worker threads based on the location of the data partitions they need to access. When a query requires scanning multiple partitions, the work is divided among threads on the relevant NUMA nodes where the partitions reside. By aligning thread execution with data placement, Quake minimizes remote memory accesses and ensures maximum utilization of memory bandwidth.

NUMA-Aware Query Execution with APS Quake integrates NUMA-aware processing with Adaptive Partition Selection (APS) to dynamically select which partitions to scan

based on query requirements and desired recall. The query processing involves both worker threads scanning local partitions and a main thread coordinating the process.

Algorithm 2 NUMA-Aware Query Processing with Adaptive Partition Selection

Input: Query vector q, Index partitions P_i with locations $Node_j$, Recall threshold τ_R, Recall model M, Period to check recall T_{wait}
Output: Top-k nearest neighbors to q satisfying recall threshold τ

Initialize: $R \leftarrow \emptyset$ (global result set), $S \leftarrow$ sorted list of partitions based on distance to q (obtained from searching parent)
Distribute q to local memory of NUMA nodes
for all NUMA nodes $Node_j$ **in parallel do**
    $W_j \leftarrow$ worker threads on $Node_j$
    $P_j \leftarrow$ partitions on $Node_j$ from S
    Enqueue partitions P_j to local job queue
while not **all** partitions in S have been processed **do**
    Main Thread:
    Wait for a predefined interval T_{wait}
    Merge partial results from worker threads into R
    Estimate current recall r using model M and results in R
    if $r \geq \tau_R$ **then**
        Break and terminate worker threads
Return top-k results from R
function WORKERTHREAD(q, Local Job Queue)
    while Job Queue not empty **do**
        $P_i \leftarrow$ **Dequeue** next partition from Job Queue
        Compute distances between q and vectors in P_i
        Update local partial results R_j
        Signal Main Thread about new partial results

Algorithm Explanation: In Algorithm 2, the main steps are:

1. **Initialization**: The query vector q is distributed to the local memory of NUMA nodes with relevant partitions. Partitions are sorted using their centroid distance to q.
2. **Worker Threads Execution**: Each NUMA node has worker threads that process partitions assigned to that node. They compute distances between q and vectors in their local partitions, updating their local partial results.
3. **Main Thread Coordination**: The main thread periodically merges partial results from all worker threads. It uses a recall model M to estimate the current recall based on the results accumulated so far.
4. **Adaptive Termination**: If the estimated recall meets or exceeds the threshold τ_R, the main thread returns the top-k results and signals the worker threads to terminate the processing of remaining partitions.
5. **Continuation or Termination**: If the recall threshold is not met and all partitions have been processed, the algorithm returns the current results.

This adaptive approach ensures that the system processes only as much data as needed to meet the recall requirements, improving efficiency and reducing query latency.

Implementation Details We implemented Quake in 7,500 lines of C++ and provide a Python API for ease-of-use. We used primitives in Faiss [8], PyTorch [19], and SimSIMD [25] to enable high-performance management of inverted lists, efficient batch tensor operations, and AVX512 intrinsics for fast distance comparisons. We also used a high performance concurrent queue [24] to prevent contention during coordination of query processing. In addition, we developed a workload generator and evaluation framework in Python to create and evaluate vector search workloads. We will release Quake as an open-source system along with its evaluation tools, the Wikipedia-12M workload, and Workload Generator.

7 Experiments

We next evaluate Quake using a number of benchmarks and summarize our main findings:

1. Quake achieves the lowest search latency across all dynamic workloads in comparison to state-of-the-art graph indexes, with $1.5-8\times$ lower search latency than HNSW, DiskANN, and SVS while having $6-83\times$ lower update latency.
2. We also find that APS Scanning matches the nprobe of an oracle across recall targets on Sift1M, with only a 20% increase in latency relative to the oracle. In addition, APS exhibits stable recall on the Wikipedia-12M workload, with only a .005% average deviation from a recall target .90% over the course of the workload.
3. Quake's NUMA-aware intra-query parallelism exhibits linear scalability and saturates memory bandwidth on the MSTuring100M dataset with 100-million records. Quake achieves $16\times$ lower query latency when compared to a single-threaded version and $4\times$ lower latency compared to a non-NUMA aware configuration.

7.1 Workloads

We performed our evaluation on a diverse set of real-world and synthetic workloads, focusing on search and update latency, recall, and scalability.

Wikipedia-12M This dataset and workload trace are derived from monthly Wikipedia page additions and page-view [4] frequencies between April 2013 and December 2021. We consider only pages about people or those linking to people. The dataset begins with 1.6 million pages and grows to 12 million after 103 updates, and therefore the average update size is $\approx$ 100,000 vectors. Embeddings are generated by training DistMult [40] graph embeddings (via Marius [23, 35]) on the Wikipedia link structure, and use the inner product metric.

The workload simulates monthly inserts of new pages, followed by 1,000 search queries sampling page embeddings with probability proportional to their page views. This setting imitates evolving interest and periodic growth of the dataset.

OpenImages-13M Using the methodology described by SVS [5], we generate a workload of 13M images from the Open Images dataset [15]. Embeddings are produced using

Clip [31] in an inner product metric space. The workload maintains a sliding window of 2 million resident vectors and inserts and deletes vectors based on class labels until all 13 million vectors have been indexed at least once. Each insert and delete affects roughly 110K vectors. After each insert and delete operation, we run 1,000 queries randomly sampled from the entire vector set. This scenario stresses both insertion and deletion performance as well as sustained query latency.

Workload Generator To test performance under different workload characteristics, we employ a configurable workload generator applicable to any vector dataset. The key parameters include: number of vectors per operation, operation count, operation mix (insert/delete/search ratio), and spatial skew. For skewed workloads, the generator forms clusters and samples queries and updates from these clusters, reflecting hot spots in the vector space.

We construct two example workloads from a 10M vector subset of the MSTuring [2] dataset using L2 distance:

- **MSTuring-RO**: A pure search workload. We uniformly sample from 10,000 provided query vectors and execute 100 search operations, each querying 1,000 vectors. This setup tests query efficiency in a static setting.
- **MSTuring-IH**: A dynamic workload interleaving inserts and searches. Beginning with 1 million vectors, the dataset grows to 10 million as we process 1,000 operations with a 90% insert and 10% search ratio. This tests the ability to handle large-scale growth while maintaining query quality.

We also used the vector datasets Sift1M [14] and MSTuring100M [2] to conduct microbenchmarks in the static setting.

7.2 Experimental Setup

Experiments are run on a machine with 80 cores, 500GB of memory with four NUMA regions providing a total of 300 GB/s memory bandwidth. Search queries are processed one at a time and we report the average latency per query to reach a recall target of 90% recall for $k = 100$. For updates, both Quake and the baselines process updates in batches; we report the average update latency per vector (i.e., total batch update time divided by batch size). This setup simulates an online environment where queries arrive individually, and updates are applied in batches. We report maintenance time separately from update latency, as maintenance can be conducted in the background in online systems [39].

Baselines: We compare Quake against several state-of-the-art methods, including both partitioned and graph-based indexes:

- **Faiss-IVF** [8]: A popular inverted file (IVF) index in Faiss. It handles updates but does no maintenance.
- **DeDrift** [6]: An incremental maintenance strategy designed to reduce clustering drift by periodically reclustering large partitions together with small ones. We implement DeDrift's logic within Quake.
- **LIRE** [39]: Maintenance procedure used by SpFresh. LIRE incrementally splits large clusters and deletes small clusters after updates, followed by local reassignments. We implement LIRE's approach within Quake.
- **ScaNN** [11]: A state-of-the-art highly optimized partitioned index system. It uses an unpublished incremental maintenance procedure similar to LIRE.
- **Faiss-HNSW** [22]: A graph-based approach (HNSW) implemented in Faiss. It supports incremental inserts but not deletes. Thus, for workloads with deletions, we omit Faiss-HNSW from those comparisons.
- **DiskANN** [32]: System built around the Vamana [33] index with support for dynamic updates.
- **SVS** [5]: A recently released optimized implementation of the Vamana index with support for dynamic updates.

We configure the main parameters of Quake and the baselines as follows. We disable vector quantization/compression for all baselines, as not all baselines support it. For partitioned indexes we use $sqrt(|X_0|)$ partitions where $|X_0|$ is the initial number of vectors in the workload. For the graph indexes, we use a graph degree of 128. For LIRE and Quake, we set the partition refinement radius $r = 25$. For Quake we set $\tau = 1$ microsecond and use one iteration of k-means for refinement. All systems use 16 threads for updates and maintenance (if applicable). SCANN, DiskANN, and SVS perform maintenance eagerly during an update, therefore we do not measure maintenance time separately from update time. We consider maintenance after each operation for all methods. Throughout all experiments, indexes are tuned to achieve an average of 90% recall for $k = 100$ across the workloads.

7.3 End-to-End Evaluation

Comparison with Baselines Table 4 shows that Quake consistently achieves lower search and update latency across all workloads. On the Wikipedia-12M workload, where the dataset grows over time and partitions can become unbalanced, Quake maintains a 0.5ms query latency. In contrast, Faiss-IVF climbs to 57.4ms due to the lack of maintenance, DeDrift reaches 45.9ms despite its rebalancing efforts, LIRE is unable to meet the recall target even with 17ms latency and SCANN performs similarly with poor update latency due to over-eager maintenance applied during updates. Even the best-performing graph-based method, DiskANN, takes 4.19ms. Thus, Quake is 8× faster than the strongest baseline on this workload derived from real-world access patterns.

On the OpenImages-13M workload, which includes both insertions and deletions, Quake's search latency is 1.01ms. The best competing approach, DiskANN, records 7.99ms, making Quake 8× faster. Faiss-HNSW cannot support deletions, and both SVS's and DiskANN's delete consolidation is expensive, leading to multiple orders of magnitude slower update latency than partitioned indexes, illustrating that graph-based indexes struggle with dynamic operations. Quake and other partitioned index baselines are two orders of magnitude faster in comparison. Quake's continuous maintenance and adaptive

Method	Wikipedia-12M			OpenImages-13M			MSTuring10M-RO		MSTuring10M-IH		
	Search	Update	Maint.	Search	Update	Maint.	Search	Maint.	Search	Update	Maint.
Quake	**.53 ms**	.004 ms	7.6 s	**1.01 ms**	.006 ms	1.71 s	2.28 ms	2.89 s	**1.94 ms**	.01 ms	.486 s
Faiss-IVF	57.4 ms	.002 ms	-	16.1 ms	.003 ms	-	44.1 ms	-	49.4 ms	.005 ms	-
DeDrift	45.9 ms	.01 ms	34.5 s	8.34 ms	.011 ms	3.43 s	-	-	69.0 ms	.01 ms	1.98 s
LIRE	15.3* ms	.01 ms	6.64 s	5.34 ms	.016 ms	1.90 s	-	-	32.7* ms	.009 ms	0.77 s
ScaNN	17.4 ms	.60 ms	-	14.8 ms	.07 ms	-	10.7 ms	-	24.1 ms	.037 ms	-
Faiss-HNSW	5.07 ms	.06 ms	-	-	-	-	6.83 ms	-	4.57 ms	.55 ms	-
DiskANN	4.19 ms	.11 ms	-	7.99* ms	.5 ms	-	4.17 ms	-	2.9 ms	.19 ms	-
SVS	7.11* ms	.197 ms	-	10.34 ms	.758ms	-	**1.18 ms**	-	7.61* ms	.095 ms	-

Table 4: Workload performance: showing single-query search latency to reach a recall of .9, mean single-update latency, and mean maintenance latency per operation. *Denotes the config is unable to meet the recall target with static query parameters.

parameter selection keep partitions balanced, achieving low latency and stable recall.

For the static, read-only, MSTuring10M-RO workload, Quake's maintenance improves the index structure even without data changes, adapting partitions to the query pattern. Combined with NUMA-aware parallelism, this yields a search latency of 2.28ms. However, the MSTuring10m dataset is especially challenging for partitioned indexes, as they need to scan roughly 10% of all partitions in order to meet the recall target. In contrast, the well-optimized SVS library exhibits a superior search latency of 1.18ms, demonstrating that in static settings, well-optimized graph indexes are strong competition.

On MSTuring10M-IH, where the dataset grows from one to ten million vectors, Quake achieves an average search latency of 1.94, ms. DiskANN, the second-best performer, has a latency of 2.9ms, making Quake 1.5× faster. The other baselines fail to maintain the recall target or suffer from high latency due to their static parameters and inability to prevent partition skew.

Overall, these results demonstrate that Quake's combination of adaptive partition scanning, incremental maintenance, and NUMA-aware parallelism consistently delivers low-latency queries at the desired recall. Systems without maintenance (Faiss-IVF) suffer from skew-induced latency increases, those tied to static search parameters (LIRE) struggle to maintain recall without incurring higher query times, and graph-based methods (Faiss-HNSW, DiskANN) face substantial overheads when handling updates and deletions. By integrating these components, Quake matches the low update cost of partitioned indexes while outperforming graph indexes in search latency in dynamic workloads. Our design represents a significant advancement to the state-of-the-art, providing stable, efficient performance across diverse, and evolving workloads.

Comparison with Partitioned Index Maintenance Methods Here we perform a detailed comparison with LIRE and DeDrift, measuring the latency, recall, and number of partitions over time on the Wikipedia-12M workload. For a fair comparison, we disable NUMA to highlight the advantages of APS and maintenance in Quake. The results are shown in Figure 4. First looking at recall, we see that Quake maintains a stable recall of near .9, while LIRE's recall degrades over time as it uses a static nprobe. DeDrift's recall stays relatively constant, as it does not adjust the number of partitions and therefore does not need to adjust nprobe. However, when turning our attention to latency, we see that Quake has near-constant stable latency, even as the dataset grows, while DeDrift's latency increases significantly with time. In terms of the number of partitions, we see DeDrift stays constant while Quake and Lire increase by 2.5× and 10× respectively. LIRE uses significantly more partitions because it uses size thresholding to determine when to split, regardless of whether a given partition is hot or not. Quake on the other hand only splits partitions if their contribution to the cost model is high, allowing for more efficient maintenance. These results show that Quake's approach to maintenance is superior to existing methods for partitioned index maintenance in minimizing query latency and recall stability.

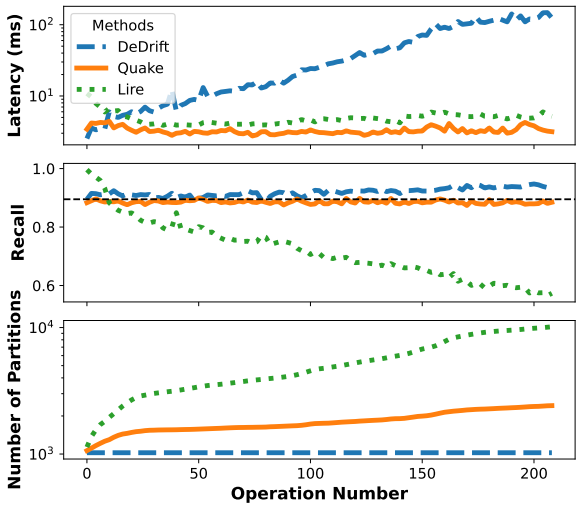


Figure 4: Comparison of single-threaded search latency, recall and number of partitions for Quake vs. maintenance approaches LIRE and DeDrift on Wikipedia-12M. Quake maintains stable latency and recall throughout the workload.

7.4 Ablation Study

To quantify the contributions of Quake components, we disabled key features and measured the impact on Wikipedia-12M workload in Table 5. We see that disabling APS has little impact on the query latency, as Quake can achieve a low latency even in the static nprobe setting. However, APS provides significantly more recall stability, as evidenced by the increase in standard deviation when APS is disabled. Disabling NUMA parallelism, however, shows a 6× increase in query latency, demonstrating the benefit of parallelization of partition scans. We note that even with NUMA disabled, Quake achieves a lower latency than the best baseline. Finally, we disable maintenance and see a significant increase in latency, similar to the latency of Faiss-IVF; here partitions become extremely imbalanced due to the skew in the workload (see Figure 1a) causing queries to scan more vectors and therefore increasing latency. This further demonstrates the necessity for maintenance for dynamic workloads. In conclusion, each piece of Quake contributes to its performance in terms of both recall stability and minimal query latency.

Table 5: Ablation Study on Wikipedia-12M showing mean search latency and the standard deviation of recall.

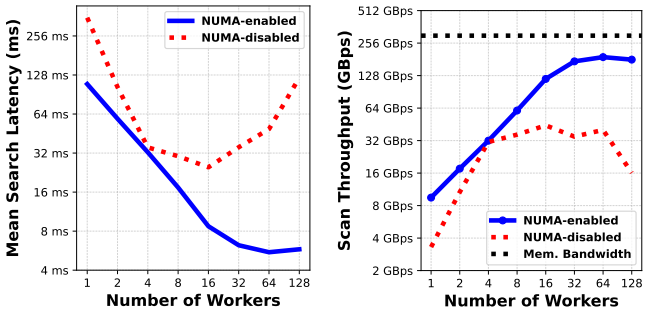
Configuration	Search Latency	Recall Std.
Quake (Full)	.53 ms	.008 %
Quake w/o APS	.5 ms	.025 %
Quake w/o NUMA	3.28 ms	.005 %
Quake w/o NUMA/APS	3.18 ms	.025 %
Quake w/o Maint/NUMA/APS	45.2 ms	.014 %

7.5 Scalability

We tested Quake's parallel scalability by varying the number of threads. In Figure 5 we measure the mean search latency and scan throughput (bytes scanned / query latency) on MS-Turing100M to reach a recall of 0.9. Note that this dataset has 100 million vectors and is 10× larger than the datasets we compared against previously. We compare our NUMA-aware parallelism with one in which NUMA is disabled. For both configurations, we see near linear scalability up to around 8 workers, where the non-NUMA latency performs best (28ms). The NUMA configuration however further improves and at 64 workers achieves a latency of 6ms. Looking at the scan throughput, we see that NUMA achieves a peak throughput of 200GBps. We do not completely saturate memory bandwidth due to other overheads involved in query processing (topk sorting, memory allocations, coordination). In conclusion, NUMA-aware intra-query parallelism is an effective mechanism for decreasing query latency by utilizing the full memory capabilities of multicore machines.

7.6 Varying Recall Targets with APS

We compared APS to an oracle that knows the exact number of partitions (nprobe) required for each query. As shown in



(a) Mean Search Latency (b) Scan Throughput

Figure 5: MSTuring100M: Scaling the number of threads with and without NUMA.

Figure 6, APS closely matches the oracle's nprobe, recall, and latency, across all recall targets, proving APS is effective despite its lack of prior knowledge. Because of the precomputation and recomputation performance optimizations (Table 3), the recall estimation in APS introduces minimal overheads which is evidenced by the query latency, which stays within 20% of the oracle.

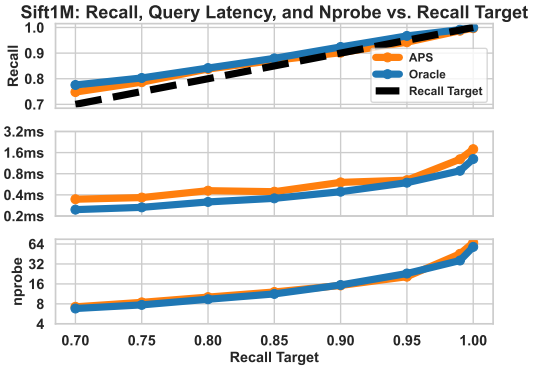


Figure 6: APS vs. Oracle on SIFT1M. APS achieves similar recall and latency w.r.t. an oracle tuned for each recall target.

8 Conclusion

Experimental results show that Quake reduces query latency compared to baseline approaches under dynamic and skewed workloads, without requiring manual tuning. It achieves high recall, matching the performance of an oracle for setting the query parameter nprobe. Compared to existing partitioned indexes like Faiss and SCANN, Quake reduces query latency by A) adaptively maintaining index partitions and B) maximizing memory bandwidth during query processing. Compared to graph indexes like SVS, HNSW, and DiskANN, Quake offers more efficient indexing and updates while matching or reducing query latency. In summary, our evaluation shows Quake minimizes query latency while meeting recall targets on dynamic workloads with skewed access patterns.

References

[1] Qdrant - Vector Database. https://qdrant.tech/.

[2] Billion-scale approximate nearest neighbor search challenge: Neurips'21 competition track. https://big-ann-benchmarks.com/, 2021.

[3] Vector database for vector search | pinecone. https://www.pinecone.io, 2024. Accessed on December 4, 2023.

[4] Wikipedia:pageview statistics. https://en.wikipedia.org/wiki/Wikipedia:Pageview_statistics, 2024.

[5] Cecilia Aguerrebere, Mark Hildebrand, Ishwar Singh Bhati, Theodore Willke, and Mariano Tepper. Locally-adaptive quantization for streaming vector search. *arXiv preprint arXiv:2402.02044*, 2024.

[6] Dmitry Baranchuk, Matthijs Douze, Yash Upadhyay, and I. Zeki Yalniz. DeDrift: Robust Similarity Search under Content Drift, August 2023. arXiv:2308.02752 [cs].

[7] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search.

[8] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library, 2024.

[9] Mihajlo Grbovic and Haibin Cheng. Real-time personalization using embeddings for search ranking at airbnb. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 311–320, 2018.

[10] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, 2020.

[11] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning*, pages 3887–3896. PMLR, November 2020. ISSN: 2640-3498.

[12] Malay Haldar, Mustafa Abdool, Prashant Ramanathan, Tao Xu, Shulin Yang, Huizhong Duan, Qing Zhang, Nick Barrow-Williams, Bradley C Turnbull, Brendan M Collins, et al. Applying deep learning to airbnb search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1927–1935, 2019.

[13] Helia Hashemi, Aasish Pappu, Mi Tian, Praveen Chandar, Mounia Lalmas, and Benjamin Carterette. Neural instant search for music and podcast. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2984–2992, 2021.

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

[15] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *International Journal of Computer Vision*, 128(7):1956–1981, March 2020.

[16] Yongjae Lee and Woo Chang Kim. Concise formulas for the surface area of the intersection of two hyperspherical caps. *KAIST Technical Report*, 2014.

[17] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 743–754, New York, NY, USA, 2014. Association for Computing Machinery.

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

[19] LibTorch: PyTorch C++ API. https://pytorch.org/cppdocs.

[20] David C Liu, Stephanie Rogers, Raymond Shiau, Dmitry Kislyuk, Kevin C Ma, Zhigang Zhong, Jenny Liu, and Yushi Jing. Related pins at pinterest: The evolution of a real-world recommender system. In *Proceedings of the 26th international conference on world wide web companion*, pages 583–592, 2017.

[21] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, and Youlong Cheng. Monolith: Real time recommendation system with collisionless embedding table. In *5th Workshop on Online Recommender Systems and User Modeling (ORSUM2022), in conjunction with the 16th ACM Conference on Recommender Systems*, 2022.

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

[23] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 533–549, 2021.

[24] moodycamel::ConcurrentQueue. https://github.com/cameron314/concurrentqueue.

[25] SimSIMD. https://github.com/ashvardanian/SimSIMD.

[26] Jiongkang Ni, Xiaoliang Xu, Yuxiang Wang, Can Li, Jiajie Yao, Shihai Xiao, and Xuecang Zhang. DiskANN++: Efficient Page-based Search over Isomorphic Mapped Graph Index using Query-sensitivity Entry Vertex, November 2023. arXiv:2310.00402 [cs].

[27] Shumpei Okura, Yukihiro Tagami, Shingo Ono, and Akira Tajima. Embedding-based news recommendation for millions of users. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1933–1942, 2017.

[28] Aditya Pal, Chantat Eksombatchai, Yitong Zhou, Bo Zhao, Charles Rosenberg, and Jure Leskovec. Pinnersage: Multi-modal user embedding framework for recommendations at pinterest. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2311–2320, 2020.

[29] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.*, 10(2):37–48, October 2016.

[30] An Qin, Mengbai Xiao, Yongwei Wu, Xinjie Huang, and Xiaodong Zhang. Mixer: efficiently understanding and retrieving visual content at web-scale. *Proceedings of the VLDB Endowment*, 14(12):2906–2917, 2021.

[31] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[32] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search. *arXiv preprint arXiv:2105.09613*, 2021.

[33] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[34] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. Soar: Improved indexing for approximate nearest neighbor search. In *Neural Information Processing Systems*, 2023.

[35] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2023.

[36] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

[37] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 839–848, 2018.

[38] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment*, 13(12):3152–3165, 2020.

[39] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 545–561, New York, NY, USA, October 2023. Association for Computing Machinery.

[40] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575*, 2014.

[41] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. {VBASE}: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 377–395, 2023.