Cracking Vector Search Indexes

Vasilis Mageirakos
Systems Group, ETH Zürich, Switzerland
vmageirakos@inf.ethz.ch

Bowen Wu
Systems Group, ETH Zürich, Switzerland
bowen.wu@inf.ethz.ch

Gustavo Alonso
Systems Group, ETH Zürich, Switzerland
alonso@inf.ethz.ch

ABSTRACT

Retrieval Augmented Generation (RAG) uses vector databases to expand the expertise of an LLM model without having to retrain it. This idea can be applied over data lakes, leading to the notion of embeddings data lakes, i.e., a pool of vector databases ready to be used by RAGs. The key component in these systems is the indexes enabling Approximated Nearest Neighbor Search (ANNS). However, in data lakes, one cannot realistically expect to build indexes for every possible dataset. In this paper, we propose an adaptive, partition-based index, CrackIVF, that performs much better than up-front index building. CrackIVF starts answering queries by near brute force search and only expands as it sees enough queries. It does so by progressively adapting the index to the query workload. That way, queries can be answered right away without having to build a full index first. After seeing enough queries, CrackIVF will produce an index comparable to the best of those built using conventional techniques. As the experimental evaluation shows, CrackIVF can often answer more than 1 million queries before other approaches have even built the index and can start answering queries immediately, achieving 10-1000x faster initialization times. This makes it ideal when working with cold data or infrequently used data or as a way to bootstrap access to unseen datasets.

PVLDB Reference Format:
Vasilis Mageirakos, Bowen Wu, and Gustavo Alonso. Cracking Vector Search Indexes. 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 URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

External data sources can complement Large language models (LLMs) [13] through the use of retrieval-augmented generation (RAG) [47]. In RAG, external data is represented as *vector embeddings* (i.e., learned representations of data that result in a multi-dimensional vector). The embeddings are indexed by a variety of means and searched through approximate nearest neighbor (ANN) search so that LLMs can answer queries on information they were not explicitly trained on. The effectiveness of RAG is tied to that of the ANN index structures, making them a critical component.

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

LLMs have been integrated with search engines [67] (e.g., Google, Bing, and Baidu). However, the vast repository of private, unstructured business data remains largely untapped. It has been estimated that unstructured data constitutes up to 80-90% of the total data volume [54, 57] and is often stored in open data formats on data lakes [55]. However, as much as 70% of this data [19] remains unused, classified as "dark" data [34, 99].

This data can be made discoverable [14], but even then, the actual data often remains unindexed. For instance, as of March 2020, Google Research's Dataset Search [12] had indexed around 28 million structured and unstructured datasets [9] based on their metadata. However, embedding-based vector search and RAG techniques make the promise to go a step further by enabling direct retrieval and question answering on the underlying data itself, inside of each dataset. This requires embedding the data, and the creation of approximate nearest neighbor (ANN) indexes for every single dataset, exposing the data to RAG systems. In this paper, we name such an approach *embedding data lakes* (EDL), where unstructured data is stored alongside its vector representations.

Efforts in this direction are already ongoing in both industry and academia. Databricks has deployed vector search capabilities [20] over their Lakehouse [4], and new custom storage formats like Lance [46] are being used to store embeddings on data lakes. Researchers are also directly exploring RAG-based techniques to query multi-modal data lakes [17, 85], as well as answering queries directly over unstructured data [2, 50]. There is also work to directly define LLM-based data discovery systems over data lakes [1].

The issue we address in this paper is crucial to achieve the promise of embedding lakes (EDL): ***index selection at scale****.* EDL, in principle, would require building millions of ANN indexes across diverse datasets, modalities, workloads, and embeddings, making static index selection and pre-computation infeasible. In data lakes, some datasets may receive millions of queries, while others are rarely accessed. When selecting an index for embedding lakes, there is a trade-off. On the one hand, building an index requires significant upfront costs that might not pay off if the data is rarely used. On the other hand, skipping the indexing process and relying on a brute-force search is not scalable. Hence, data must be indexed using Approximate Nearest Neighbor (ANN) indexes but this raises two questions: **when to index the data and how to index it**.

An optimal ANN index depends on workload patterns and future usage, which are often unknown. As shown in Figure 1, the index that minimizes total cumulative time spent on building and searching varies with the query volume. Larger indexes take longer to build and require many queries to justify their upfront cost (e.g., the large IVFlat configurations (5000, 16000 clusters) in the figure). Brute-force search (as shown in the figure), and smaller indexes (IVFlat 1000 clusters), minimize time-to-query for fast discovery but struggle at scale due to their higher query response times.

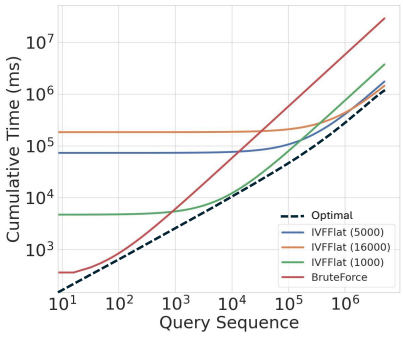


Figure 1: Total time to answer the queries submitted for different indexing strategies vs the number of queries submitted

To tackle index selection at scale, we adopt a well-established strategy: carefully avoiding the need to build the full index upfront. In this, we draw inspiration from deferred data structuring [43] and database cracking [32, 36, 78], which have been successfully applied to single-dimensional database indexing. Similar approaches have been extended to multidimensional data [35, 65, 69, 98]. However, prior work does not scale to the embedding dimensionalities nor supports the k-Nearest Neighbour (k-NN) queries typical in RAG systems. Recently, the AV-Tree was the first adaptive index for high-dimensional k-Nearest Neighbor (k-NN) search [45]. AV-Tree targets short-lived data, with workloads of up to one thousand queries, and is a tree-based Exact Nearest Neighbor (ENN) adaptive index. This makes it impractical for RAG-scale workloads, which rely on Approximate Nearest Neighbor (ANN) index structures to trade off a bit of accuracy for large performance gains. Nonetheless, the idea of dynamically building an index tailored to the workload remains compelling. In this paper, we demonstrate how to achieve this efficiently in the context of RAG and employing the same data structures [82] and systems [23] used for ANN search workloads [6, 81] rather than analytics over cold data.

We introduce **CrackIVF**, an incrementally built, partition-based ANN index that dynamically improves itself to adapt to increasing workload demands, minimizing both time-to-query and cumulative cost associated with suboptimal static index selection. CrackIVF evolves as a side effect of query execution. Each query is a candidate to add a new partition, or crack, while its search region is a candidate for local refinement, allowing the index to grow and improve as more queries arrive. Efficiency is maintained through two controls: one deciding where to apply cracking and refinement in the search space, and another controlling when, to balance indexing and search. In the paper we focus on a static setting (i.e., no data or query distribution shifts) and evaluate CrackIVF on multiple standard open source datasets for ANN search [7, 11, 41, 70].

For partition-based indices, index selection means deciding upfront how many partitions the index will use, which has an impact on the ANN search performance. This decision has to be made without knowing the future query workload. CrackIVF waits to build the index until it sees enough queries and decides on the clusters based on the query distribution over the search space. As more queries arrive, the clusters are redefined, and their number is increased to adapt to the increasing query workload. Asymptotically, CrackIVF builds an index that converges to similar to those pre-built, but it has been able to answer queries along the way.

Across benchmarks, CrackIVF consistently outperforms pre-built partition-based indexes. It converges to near-optimal query response time, yet achieves several orders of magnitude lower startup time than other indexes. It efficiently scales to large workloads and, in some cases, it can process 1 million queries before the baseline indexes have finished building. It is the only index that consistently remains near the Pareto frontier of minimum cumulative time across the number of queries posed to the system.

2 BACKGROUND

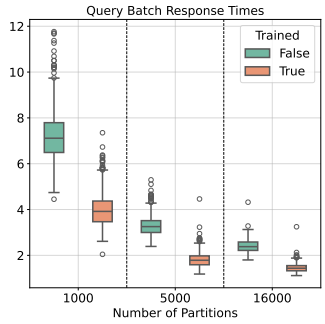
In this section, we briefly cover the basics of approximated nearest neighbor search and IVF indexes.

Nearest Neighbor Search In the *k-nearest neighbor* (*k-NN*) search problem, we are given a set of points $P \in \mathbb{R}^{|P| \times d}$ and a query $q \in \mathbb{R}^d$. The aim is to find the k points in P that are closest to q under some notion of distance or similarity (e.g., minimize the Euclidean distance (L2) or maximize the inner product (IP) to measure closeness). A straightforward Exact Nearest Neighbor (ENN) approach is a linear scan of every point, a.k.a *brute-force search*, incurring in $O(|P| \cdot d)$ complexity. There exist more sophisticated *exact indexing* structures (e.g., KD-trees [10], R-trees [30], or VP-trees [97]). However, they often degrade to near-linear-scan performance in higher dimensions, due to the curse of dimensionality [95], making them impractical when $|P|$ or d are large. With embeddings of hundreds to thousands of dimensions [66] and real-world datasets at the scale of billions of data points [81], RAG systems turn to Approximate Nearest Neighbor (ANN) search techniques. They allow a small retrieval error ε in exchange for better runtime or memory usage. This approximation quality can also be evaluated by the fraction of exact top-k vectors recovered.

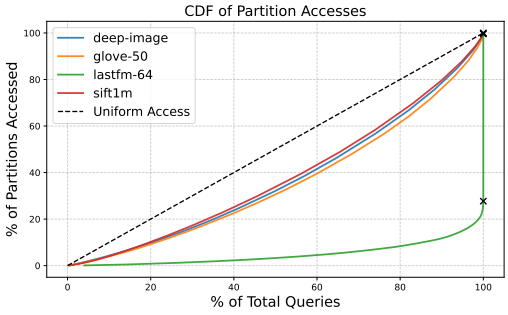
Partition based IVF methods One of the most common ANN search structures is *partition-based IVF indexes* [16, 82, 83]. Instead of scanning all points in $P \subset \mathbb{R}^d$, they divide the vector space into n_{list} disjoint groups, or *partitions*. At query time, the search process first identifies the $n_{\text{probe}} \leq n_{\text{list}}$ nearest partitions to q, whose representative vectors are closest (or most similar) to the query. Once these partitions are selected, the points within them are exhaustively scanned. Since the remaining partitions are skipped, some nearest neighbors may be missed, which is why the result is an *approximate* solution. A partition-based index is built by first selecting n_{list} partition *representative vectors*, typically using k-means on a training subset $P_{\text{train}} \subset P$. Each data point is then assigned to the nearest representative, forming disjoint partitions. These assignments are typically stored in an *inverted file* (IVF) structure [82]. A widely adopted implementation of IVF indexes is the FAISS library from Meta [23], which we use as a baseline. See Section 6 for other ANN index types.

3 SOLUTION OVERVIEW

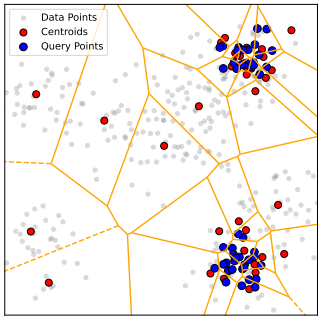
We focus on Partition-based IVF indexes because they offer several advantages for deployment in embedding data lakes. They offer the best trade-off in index build time, query response time, and index size [74]. They have the lowest index build times relative to query



(a) Response Time with increasing Partitions and training K-Means.



(b) Queries exhibit a skewed access pattern over the search space that is often far from uniform.



(c) Finer partitioning of the Vector Space based on queries received.

Figure 2: Observations behind the design: (a) separating number of partitions and refinement of centroids; (b) access to the index is generally skewed towards certain regions; (c) regions queried often can be clustered and refined at a much lower granularity

throughput (QPS). All of the above are crucial for embedding data lakes, where potentially millions of indexes of various sizes and levels of utilization will have to be constructed and stored.

3.1 Cracking in high dimensions

A well known approach to avoid upfront cost for indexing or organizing data is *cracking*. In database cracking [36], there is a final state where a single column is sorted. Over time, fewer points are moved, and over shorter distances, reducing the overhead of the cracking. Additionally, in a naive implementation, the physical reorganization happens with every incoming query.

In high-dimensional spaces, the assumption of a fully sorted state no longer holds, and space filling curves, like Z-order [77], do not scale past a few dimensions. Additionally, enforcing reorganization after every query becomes increasingly difficult, as it is a more expensive operation. Moving points between inverted lists incurs random access overhead, with no eventual locality benefits. High-dimensional vectors require moving multiple bytes per dimension, increasing data movement costs. Furthermore, inverted lists, are often implemented as memory-aligned contiguous arrays to allow for efficient SIMD-based kernels, and they may require linear-time compaction even for a single vector to move. These factors contribute to the significant latency overhead, making physical reorganization after every query difficult.

Based on this, the challenge to address is as follows *A cracking-based IVF index must efficiently manage the significantly higher overhead of physical reorganization in high dimensions, while minimizing time-to-query, continuously improving the index, and ensuring minimum cumulative time across all scales.*

3.2 Outline of the approach

As Figure 2a illustrates, index construction can be divided into two operations that independently improve index performance: (*i*) increasing the number of partitions by randomly selecting partition representatives and (*ii*) refining these representatives into well-placed centroids using k-means training, which helps to distribute points more evenly across partitions and improve the overall representativeness. Although both improve the index independently, the best results consistently come after reaching a maximum index size, even with randomly initialized representatives, followed by k-means training to optimize their placement. Thus, the optimal approach is to first increase the number of partitions up to a limit and then refine their locations using k-means.

Observation 1 behind the approach: *Index construction can be decoupled into two distinct build operations, which can improve the index independently and be* ***separated in time****.* .

Existing partition-based IVF vector search indexes apply build operations evenly across the vector space, with centroids sampled uniformly and K-means refinements performed simultaneously throughout. While simple and intuitive, this approach assumes equal importance across the space, and requires that both operations are applied at the same time on the entire space leading to larger than necessary index build times. However, vector search workloads can exhibit highly skewed access patterns, where certain regions are significantly more important than others. For instance, in an industrial workload [59, 60], up to 85% of centroids are never accessed during search, which would make partitioning or training efforts in those regions unnecessary. As we show in Figure 2b, skewed query distributions also appear in well known open-source vector search datasets [7, 11, 41, 70]. Even on open source datasets showcasing moderate amounts of skew, where 80% of the total queries access 60% of the partitions, the disparity between the most frequently and least frequently accessed partitions is upwards of 50 times. There is a *fundamental inefficiency in current uniform indexing approaches*: a large portion of the vector space may rarely accessed, yet the indexing effort is evenly distributed across the space, and always paid upfront.

Observation 2 behind the approach: *Just as index construction can be* ***separated in time****, it can also be* ***separated in space****. The index build operations do not have to be applied uniformly across the vector space; instead, they can be focused on regions that are accessed more frequently. Since certain areas of the vector space are more critical than others, adapting index construction to follow the query distribution is a natural strategy to optimize efficiency and accelerate indexing.*

Lastly, in partition-based IVF indexes, a query first identifies the nearest partitions and then scans all points within those partitions

to retrieve the k nearest neighbors. This involves the same distance computations needed to update the index, which naturally leads to a *cracking-based* approach, where index refinement occurs as a side effect of query execution.

Observation 3 behind the approach: *We can further amortize the cost of specific build operations, such as assigning vectors to partitions and identifying the local regions to crack and refine, Figure 2c. We can* ***reuse computations from search operations*** *to perform index construction as a side effect of query execution.*

3.3 Outline of the solution

Based on these insights, we propose a partition-based crackable IVF index for ANN search. It is designed to remain near the optimal minimum cumulative time across the number of queries submitted (Figure 1), improving as the workload increases, and addressing the core issue of indexing parts of the search space that are not used.

CrackIVF employs two localized and independent build operations: *CRACK* and *REFINE*. The former introduces new partitions, while the latter applies a localized K-means to improve them. These operations run independently across both time and vector space.

We directly follow the query distribution, where queries themselves act as candidates for *CRACK* and *REFINE*, determining where new partitions should be introduced, so they are not randomly sampled, and which local regions need refinement. Intermediate results from search operations, such as the distances computed to nearby points, can be re-used to amortize the cost of assignments. Using the queries to create partitions, rather than selecting random ones, allows the distances computed during search to be used for stealing points from the local region, that are closer to the query than their current assignment. Additionally, using the queries as candidates for *CRACK* and *REFINE* enables CrackIVF to exactly follow the query distribution in terms of when and where to execute build operations, as partitions that receive a larger number of queries will proportionally be candidates for cracking and refinement by the same larger amount.

Finally, to mitigate the high overhead of storage reorganization in high-dimensional IVF indexes, we do not perform build operations after every query. Instead, *CRACK* operations are buffered and committed when beneficial, while *REFINE* is executed eagerly but infrequently to improve local regions that queries are visiting when necessary. To automate these decisions, we introduce two control mechanisms. The first is a set of heuristic rules, that determine where in the vector space cracks should be buffered or refinements executed. The second is a cost-based approach that estimates the latency overhead of a build operation and constrains it relative to the cumulative runtime of the index since initialization. As more queries arrive and the index is used more frequently, the budget for build operations increases proportionally, allowing the index to grow and refine in sync with the query workload.

4 CRACK-IVF

CrackIVF, is a partition-based crackable IVF index implemented in FAISS [23]. The core logic of the algorithm and operations are covered in Section 4.1 and Algorithm 1. Control mechanisms, including heuristic rules and cost models, are described in Section 4.2 and Section 4.3. They aid in making accurate decisions about where and when to perform the build operations, *CRACK* and *REFINE*, which are shown in Algorithm 2 and Algorithm 3. The notation used in this section is summarized in Table 1.

Symbol	Description
$\|X\|$	Cardinality, e.g., $\|X\| = N$ if $X \in \mathbb{R}^{N \times d}$
$metric(a, b)$	Similarity metric, where in our work $metric \in$ {Euclidean Distance (L2), Inner Product (IP)}
$P \in \mathbb{R}^{\|P\| \times d}$	Data points, d-dimensional
$C \in \mathbb{R}^{\|C\| \times d}$	Committed cracks
$C_{\text{local}} \in \mathbb{Z}^{nprobe}$	Local region centroid IDs
P_{train}	K-means training points, where $\|P_{\text{train}}\| = \min(\|P_{\text{local}}\|, \|C_{\text{local}}\| \times \max_{\text{points}})$
n_{iter}	K-means iterations
$\max_{\text{points}}$	K-means max training sample per centroid
$nlist$	Total inverted lists storing P, where $nlist = \|C\|$
$Q \in \mathbb{R}^{bs \times d}$	Query set per search
$D \in \mathbb{R}^{bs \times \|r\|}$, $I \in \mathbb{Z}^{bs \times \|r\|}$	kNN distances and point indices from *SEARCH*, where $\|r\| \in [k, \|P_{\text{visited}}\|]$
bs	Total queries batched in single *SEARCH*
$nprobe$	Number of partitions scanned per *SEARCH*
k	Number of nearest neighbors to return
α	Ratio of total build time to total time
$A^*_P \in \mathbb{Z}^{\|P\|}$	Point assignments to centroid IDs
$D^*_P \in \mathbb{R}^{\|P\|}$	Point distances to centroids
$H^*_C \in \mathbb{N}^{\|C\|}$	Histogram of points per centroid

Note: $* \in \{\text{dyn}, \text{true}\}$, where *dyn* refers to the dynamic state (includes buffered cracks) and *true* refers to the true index state.

Table 1: Notation used in algorithms and cost model.

4.1 Algorithm and Operations

CrackIVF performs index construction and storage reorganization operations as a side-effect of query execution to incrementally improve index performance. It has three core operations, *SEARCH* finds the k-NN results to return to the user, and *CRACK*, *REFINE* are the build operations that perform physical reorganization of the IVF structure as well as change the index state.

SEARCH: The search procedure is common among partition based IVF indexes, where the local region to the query is identified by the nearest partitions, whose point assignments are subsequently scanned to find the final k nearest neighbors of the query. In our case, the result are point ids, I, and distances D to the query, of the visited points in the local region sorted by similarity, together with the ids of the cracks that the query visited, $C_{visited}$, as seen in Line 7 of Algorithm 1.

CRACK: Physical reorganization of points between inverted lists, and the mutation of the index does not happen after each query. Each incoming query represents a crack candidate, which is evaluated by heuristic rules to classify it as good or bad (Section 4.2) (Line 9 Algorithm 1). Only good cracks are further buffered and potentially committed at a later time. Each crack is essentially a new partition, where the points assigned to the partition have been "stolen" based on their proximity to the query. We reuse the results of the *SEARCH* operations to compare if the distance of a point to the query is closer than the distance of the point to its current assignment, which could be another buffered crack. If it is, then it's beneficial to reassign the point to the new crack, and this assignment is also buffered, to be committed when *CRACK* is executed, as explained in Section 4.3. *CRACK* is the procedure performing the physical reorganization, syncing the buffered state of the index with the currently working state. The core individual kernels of the procedure are sequentially executed (Algorithm 2). It adds a

Algorithm 1 SearchAndCrack

Require: $k, Q, \alpha, State_{dyn}, State_{true}, T_{buld}, T_{search}$
1: **\\ Step 1: SEARCH**
2: $T_{\text{start\_search}} \leftarrow$ CurrentTime()
3: $(D, I, C_{visited}) \leftarrow SEARCH(Q, k)$
4: $T_{\text{search}} \leftarrow T_{\text{search}}$ + (CurrentTime() − $T_{\text{start\_search}}$)
5: **\\ Step 2: CRACK Decision (Where to crack, see §4.2)**
6: **for** each query $q \in Q$ **do**
7: $(D_{\text{local}}, I_{\text{local}}, C_{\text{local}}) \leftarrow (D[q], I[q], C_{visited}[q])$
8: $I_{\text{steal}} \leftarrow \{p \in I_{\text{local}} \mid D_{\text{local}}(p)$ op $D_P^{\text{dyn}}(p)\}$ {op = < for L2, > for IP}
9: $good\_crack\_candidate$ = CrackHeuristic(I_{steal}, $\text{State}_{\text{dyn}}$)
10: **if** $good\_crack\_candidate$ **then**
11: $C_{\text{buffered}} \leftarrow C_{\text{buffered}} \cup \{q\}$
12: $I_{\text{buffered}} \leftarrow I_{\text{buffered}} \cup I_{\text{steal}}$
13: \\ update dynamic state
14: **end if**
15: **end for**
16: **\\ Step 3: CRACK Decision (When to Commit, see §4.3)**
17: $\hat{T}_{\text{CRACK}} \leftarrow$ EstimateCrackCost()
18: $can\_afford\_crack \leftarrow (T_{\text{build}} + \hat{T}_{\text{CRACK}} \leq \alpha \cdot (T_{\text{build}} + T_{\text{search}} + \hat{T}_{\text{CRACK}}))$
19: $enough\_buffered \leftarrow (|C_{\text{buffered}}| > C_{\min})$
20: **if** can_afford_crack **and** $enough\_buffered$ **then**
21: \\ UNDO if *REFINE* invalidated previously good cracks
22: $C_{\text{buffered}} \leftarrow \{c \in C_{\text{buffered}} \mid$ CrackHeuristic$(c)\}$
23: $\forall p$ where $A_P^{\text{dyn}}(p) \notin C_{\text{buffered}}, A_P^{\text{dyn}}(p) \leftarrow A_P^{\text{true}}(p)$
24: $I_{\text{buffered}} \leftarrow \{p \mid A_P^{\text{dyn}}(p) \neq A_P^{\text{true}}(p)\}$
25: $T_{\text{start\_crack}} \leftarrow$ CurrentTime()
26: CRACK($C_{\text{buffered}}, I_{\text{buffered}}$) {Algorithm 2}
27: $T_{\text{build}} \leftarrow T_{\text{build}}$ + (CurrentTime() − $T_{\text{start\_crack}}$)
28: \\ dynamic index state now matches true state
29: **else**
30: **\\ Step 4: REFINE Decision (Where & When, see §4.2 and §4.3)**
31: **for** each query $q \in Q$ **do**
32: $\hat{T}_{\text{REFINE}} \leftarrow$ EstimateRefineCost()
33: $can\_afford\_refine \leftarrow (T_{\text{build}} + \hat{T}_{\text{REFINE}} \leq \alpha \cdot (T_{\text{build}} + T_{\text{search}} + \hat{T}_{\text{REFINE}}))$
34: **if** can_afford_refine **then**
35: $good\_refine\_candidate \leftarrow$ RefineHeuristic(C_{local}, $\text{State}_{\text{true}}$)
36: **if** $good\_refine\_candidate$ **then**
37: $T_{\text{start\_refine}} \leftarrow$ CurrentTime() {Start timing REFINE}
38: REFINE(...)
39: $T_{\text{build}} \leftarrow T_{\text{build}}$ + (CurrentTime() − $T_{\text{start\_refine}}$)
40: \\ update dynamic state and true state
41: **end if**
42: **end if**
43: **end for**
44: **end if**
45: **return** $D[:, :k], I[:, :k]$

Algorithm 2 CRACK

Require: $C_{\text{buffered}}, I_{\text{buffered}}, State_{dyn}, State_{true}$
Ensure: New cracks committed; points reassigned; index state updated.
1: **Get Local Region:** Retrieve all points indexed by I_{buffered}, i.e., all data points that cracks in C_{buffered} visited.
2: **Commit Reorg (w/ $\mathbf{A_P^{dyn}}$) :** Reorganize storage. Commit C_{buffered} to the index. Reassign points to inverted lists using tracked assignments A_P^{dyn}
3: **Update Centroids:** Recompute each crack's representative to be the centroid of the points assigned to it.
4: **Update Index State:** Synchronize $State_{dyn}$ and $State_{true}$ to reflect the changes.

Algorithm 3 REFINE

Require: $C_{local}, State_{dyn}, State_{true}$
Ensure: Refinement local region; points reassigned; index state updated.
1: **Get Local Region:** Retrieve all points P_{local} in local region.
2: **Local K-Means:** Refine region of C_{local}, producing C'_{local} centroids.
3: **Commit Reorg (w/o $\mathbf{A_P^{dyn}}$):** Replacing C_{local} with refined C'_{local}. Compute new local point assignments. Reorganize storage, with local reassignment of P_{local} to inverted lists.
4: **Update Index State:** Synchronize $State_{dyn}$ and $State_{true}$ to reflect the changes.

new inverted list for each buffered crack, moves stolen points to their new assignments, and updates the partition representatives to be the centroid of the points based on their new assignments. This updates the location of the crack representatives, whose point assignments were stolen, in order for their new location to match the underlying assignment distribution. This procedure is only executed when enough budget for build operations has been accumulated through the usage of the index (Line 18 Algorithm 1). This control mechanism ensures that CrackIVF does not spend a disproportionate amount of time on CRACK operations. The parameter α, the ratio of total build operations time to the total time, is what controls the available budget. Thus, since *CRACK* is lazy, the heuristics governing where a crack is added and the cost-based control mechanism for when it is added are independent.

REFINE: Refinement is the second operation to improve the index. It is performed eagerly, focusing on the local region visited by the query. The goal is to immediately correct suboptimal regions in the current index. Thus, the buffered states and buffered cracks within the local region are not considered. The decision to refine a region is based on heuristic rules that evaluate the imbalance of point assignments in local cracks. Additionally, executing a refinement operation depends on an estimate of the runtime cost, determined using a cost model specific to *REFINE* (Line 33 Algorithm 1).

Unlike cracking, the opportunity to refine a local region is transient. The heuristic rules must decide that the currently visited local region is a good candidate for refinement, and at the same time, there must be sufficient computational budget to execute the operation. If where and when to execute a refine do not simultaneously agree, the next opportunity for the same region will arise only when another query accesses it.

REFINE, as described in Algorithm 3, executes a sequence of kernels implementing a local K-means variation and performs physical reorganization. This process improves the placement of crack representatives within the local region and reassigns points to their nearest representative. Unlike *CRACK*, *REFINE* incurs additional computational costs as it does not rely on precomputed buffered assignments. Instead, assignments are determined dynamically upon completion of the local K-means process.

Index State: Since *CRACK* is a lazy operation, the index can exist in two concurrent states, the true state and a dynamic state. The true state reflects the current structure of the index, while the dynamic one tracks its expected future structure for when *CRACK* is executed, which synchronizes the two states. Maintaining both is essential. *REFINE* operations directly alter the current index state, while cracking decisions are made based on the dynamic one, leading to scenarios that require us to undo buffered crack decisions. Initially, the heuristic rules considered some crack candidates as beneficial, and CrackIVF buffered them, but they have since been altered. Rolling back the dynamic state avoids bad cracks from being committed and having a situation where we commit decisions to the index that we know to be suboptimal (Line 21 Algorithm 1).

To track the *dynamic, uncommitted, state* of the index, we define:

$$\text{State}_{\text{dyn}} = (A_P^{\text{dyn}}, D_P^{\text{dyn}}, H_C^{\text{dyn}}) \quad (1)$$

which consists of *dynamic assignments* A_P^{dyn}, *dynamic point distances* D_P^{dyn}, and a *histogram* H_C^{dyn} of true and buffered crack sizes.

Similarly, the *true state* is defined as:

$$\text{State}_{\text{true}} = (A_P^{\text{true}}, D_P^{\text{true}}, H_C^{\text{true}}) \tag{2}$$

representing the true, committed state of assignments, distances, and crack sizes in the index.

The initialization of the above states happens during the index setup time. More specifically, in our FAISS-based implementation, during an ".add()" operation. This is when the data vectors are assigned to their nearest partitions by calculating the distances to each partition representative. We extend this operation to return to the assignments and distances computed, which were an intermediate result. The state held in A_P^{dyn},D_P^{dyn} and H_C^{dyn}, is only consistent with the actual state of the index after *CRACK* operations and before buffering any cracks. If no cracks are buffered and only *REFINE* operations are executed, the dynamic and true state are always consistent. All of the above state metadata is used after the SEARCH operation, along with the search results $I, D, C_{visited}$ to classify each query as a good or bad crack candidate (Section 4.2).

Rollback of buffered state: Consider the following scenario. A *CRACK* operation was just executed so at t_0 $State_{true} = State_{dyn}$. After a few queries, at t_1, assume that 2 good crack candidates have been buffered, both in the C_{local} region of the vector space and thus now $State_{true} \neq State_{dyn}$. At a later point, t_2, a *REFINE* is executed, also in C_{local} refining it to C'_{local} and changing S_{true}. Since *REFINE* operates to improve the current state eagerly, the decision that the buffered cracks are good was taken prior, at t_1 based on C_{local}, and thus they may now have had their points stolen back, and assigned to the refined cracks in C'_{local}. Thus even though still $|C_{buffered}| = 2$, it may now contain bad cracks. So when at t_3, it is a good time to execute a *CRACK* operation again, all buffered cracks are re-evaluated, and only the good cracks that remain are committed, ensuring that in the end, we are back to $State_{true} = State_{dyn}$. This logic can be seen in lines 22-24 of Algorithm 1.

Memory Overhead and Optimizations: The additional memory cost of maintaining $State_{\text{dyn}}$ and $State_{\text{true}}$ is $O(4|P| + 2|C|)$, which is negligible since $d \gg 4$, so in our in-memory implementation that data alone are $O(|P| * d)$. Additionally, these array structures are freed once the index converges or incoming queries stop. In our case, it enables an efficient compute-storage trade-off, deferring cracking costs instead of paying them per query.

There are a number of optimizations that a fully fledged implementation can make use of. For embedding lakes, where vectors are stored on disk, state tracking arrays can be stored alongside data points instead of in the index (irrelevant for our in-memory FAISS implementation). Since queries for an individual index are typically infrequent, $State_{\text{dyn}}$ can be committed or flushed between queries, while $State_{\text{true}}$ is recomputed as needed. Full memory overhead is only necessary during bursty query loads, where cracking must be buffered to reduce cumulative overhead and prevent long tail response times. Finally, the cracking overhead can be hidden from query response times since results are available before any cracking operations begin. A separate thread pool can handle cracking operations while queries use the current index state, provided *CRACK* and *REFINE* remain atomic to prevent inconsistencies. Prior work on the related problem of index maintenance has demonstrated similar approaches to minimizing the overhead of mutating the ANN index state with concurrent operations [96], we further cover these approaches in Section 6.

Our work assumes a continuous query load at the system's capacity, leveraging inter-query parallelism. As a result, the memory overhead for index state tracking is unavoidable in our implementation if we aim to maintain high performance.

4.2 Where to apply build operations?

There is a constant stream of crack and refinement candidates, and as observed in Section 3, not all regions of the vector space are equally important, and not all operations applied lead to improvement of index performance. Thus, a set of rules is needed to distinguish good candidates from bad ones. The heuristic rules provided in this section aim to distill the learning from our initial observations to capture independent cases where we want to add a new crack or refine a region. At this stage, they are not an exhaustive list, as we plan to extend them in future work.

Where to CRACK: The *CrackHeuristic* determines whether an incoming query is a *good* or *bad* candidate for cracking, as shown in Algorithm 1. It evaluates the stolen point IDs I_{steal} and the dynamic state $State_{\text{dyn}}$ to return a *True/False* decision. If classified as *good*, the query is buffered for a future *CRACK*. By default, we aim to expand the index size, as optimal performance is usually achieved at some maximum number of partitions (Figure 2a). Since *REFINE* does not add partitions, we treat every incoming query as a good candidate for *CRACK*, which does. This ensures a constant stream of candidates for growing the `CrackIVF` index, rejecting only those that fail our heuristic rules.

CrackHeuristic: A query is considered a *bad* candidate (returning *False*) if *any* of the following conditions hold:

Don't Crack: Rule 1 rejects queries with too few stolen points:

$$|I_{\text{steal}}| < \text{min\_pts} \tag{3}$$

Don't Crack: Rule 2 prevents excessive partitioning, constraining the number of partitions within a region relative to the number of points:

$$\frac{\text{total points in local region}}{\text{total cracks in local region}} > \text{pts\_crack\_thr} \tag{4}$$

For all experiments, we set min_pts = 2 to avoid empty cracks and pts_crack_thr = 64, ensuring at least 64 points exist per crack locally. These values define extreme limits, filtering out crack candidates that are clearly bad. Having these relaxed constraints allows fine-grained partitioning of popular regions, where most crack candidates are, and future *REFINE* operations can always improve their local locations and re-allocate assigned points.

Where to REFINE: The *RefineHeuristic* determines whether a local region in the index is a good candidate for refinement, returning a $True/False$ value for if a refinement should take place. Unlike *CRACK*, where the latency overhead is amortized by buffering, *REFINE* is applied eagerly and is also an expensive operation.

Therefore, we do not want to call *REFINE* repeatedly, unless necessary, and our default behavior is to assume that every incoming query local region is NOT a good candidate for refinement unless it satisfies specific heuristic rules that indicate a significant local partition imbalance. Imbalance affects tail response time negatively and increases its variance, since even though queries may be accessing the same number of partitions, they may be scanning a vastly different number of points. Additionally, running K-means improves the centroid location of the crack representatives, making them better fit the underlying data distribution, which also has downstream benefits [59].

Various measures exist that can be used to detect imbalances in cluster sizes [33]. We selected the following subset as they capture different aspects of imbalance, both locally and globally.

RefineHeuristic: A local region is considered a *good* candidate for refinement (returning *True*) if *any* of the following conditions hold:

Refine: Rule 1 captures local imbalance using the coefficient of variation:

$$\mathrm{CV}_{\mathrm{local}} = \frac{\sigma_{\mathrm{local}}}{\mu_{\mathrm{local}}} > \mathrm{cv\_max} \tag{5}$$

where σ_{local} and μ_{local} are the standard deviation and mean of cluster sizes in the local region. A high coefficient of variation indicates substantial local imbalance, warranting refinement.

Refine: Rule 2 captures extreme values using local spread:

$$\mathrm{Spread}_{\mathrm{local}} = \frac{\mathrm{max}_{\mathrm{local}} - \mathrm{min}_{\mathrm{local}}}{\mu_{\mathrm{local}}} > \mathrm{spread\_max} \tag{6}$$

This ensures that refinement is triggered when large disparities exist between the smallest and largest clusters within a local region.

Refine: Rule 3 captures imbalance based on the global distribution:

$$\exists c_1, c_2 \quad \text{s.t.} \quad \begin{aligned} S(c_1) &\leq \mathrm{size\_pctl\_low}, \\ S(c_2) &\geq \mathrm{size\_pctl\_high} \end{aligned} \tag{7}$$

where c_1, c_2 are cracks in the local region, and $S(c) = H_C^{\mathrm{true}}[c]$ represents the size (i.e., number of assignments) of crack c. This similarly captures imbalances, between small and large partitions, but now as defined by the global cluster sizes.

For all experiments, we set cv_max = 2, spread_max = 10, meaning refinement is triggered if cluster sizes vary by at least twice the local mean or the range of largest to smallest local cluster is 10× the local mean. Finally, small and large cracks globally are defined by the 10th and 90th percentiles, size_pctl_low = 10, size_pctl_high = 90 of the global cluster sizes.

These thresholds ensure that refinement is applied only when clear imbalance exists, maintaining index efficiency. An optimized implementation could relax these rules when more computational budget is available, allowing refinement of less imbalanced regions.

4.3 When to apply build operations?

To mitigate the overhead of constant physical reorganization, which is expensive in high-dimensional IVF indexes, we use a build budget that restricts the fraction of time dedicated to *CRACK* and *REFINE* operations. We define T_{build} as the total measured time spent on previous build operations, which include historical *CRACK* and *REFINE*. Similarly, T_{search} represents the cumulative time spent on all past *SEARCH* operations. When considering a potential build operation, we estimate its execution cost using $\hat{T}_f$, which captures the expected time required to perform either a *CRACK* operation under the current dynamic state ($State_{dyn}$) or a *REFINE* operation on the local region of the current refine candidate. Here, $f \in \{\mathrm{CRACK}, \mathrm{REFINE}\}$ specifies the build operation considered.

To enforce the budget constraint, we ensure that the total time spent on past build operations in addition to the projected time for the current one remains within a fraction α of the system's total runtime:

$$T_{\mathrm{build}} + \hat{T}_f \leq \alpha \cdot (T_{\mathrm{build}} + T_{\mathrm{search}} + \hat{T}_f). \tag{8}$$

T_{build} and T_{search} are directly measured from execution, while $\hat{T}_f$ is derived from a predictive cost model, which is explained in the following subsection. The parameter α governs the proportion of total execution time that can be allocated to indexing operations, ensuring a balanced trade-off between indexing efficiency and query performance. This approach allows for a consistent and adaptive mechanism to regulate both cracking and refining decisions under varying workload conditions.

Varying α is a way of expressing an expectation about the number of queries that will arrive. A large α allocates a larger build budget, which is useful if we are optimistic about receiving a large number of queries in the future, thus allocating more time now for build operations would be beneficial. A smaller α restricts the build budget, indicating that we are pessimistic and do not expect many more queries to arrive. To have a balance, in all our experiments, we set $\alpha = 0.5$, to ensure a maximum of 50% of the total time at any given point is spent on build operations. This budget ensures that the build operation overhead does not dominate the overall execution time. As more queries arrive, CrackIVF will progressively have a larger budget to apply for physical reorganization operations. In this way, we can remain near the optimal minimum of cumulative time across all query scales. Finally, there is a minimum size constraint on the number of buffered cracks before a *CRACK* is executed. Specifically, in our implementation, we set that minimum to 20% of the current index size, $|C_{buffered}| > C_{min}$, where $C_{min} = 0.2 * |C|$ (Line 19 Algorithm 1). This helps avoid situations where a *REFINE* operation applied on the current index achieves similar performance improvements, while allowing us to keeping the current index size without increasing it.

Predictive Cost Model

The cost estimation of both *CRACK* and *REFINE* operations happens after every incoming query. Thus, it is crucial to maintain low latency when making a prediction while at the same time maintaining an accurate enough estimate to satisfy our usage needs. To achieve this balance, we use a simple first-order linear model and fit it using multivariate regression [39].

Each build function f comprises a sequence of procedures, or kernels, indexed by i and executed sequentially, as shown in Algorithm 2 and Algorithm 3. Among these, updating the index state is negligible, and the following kernels are the dominant factors of each function's execution time:

CRACK: (i) *Get Local Region*, (ii) *Commit Reorg* (w/ A_P^{dyn}), (iii) *Update Centroids*.

REFINE: (i) *Get Local Region*, (ii) *Local k-Means*, (iii) *Commit Reorg* (w/o A_P^{dyn}).

Since the above kernels in our implementation execute sequentially without overlap, the total execution time T_f is simply the sum of the execution times of its kernels.

$$T_f = \sum_i T_{f,i}, \quad \forall f \in \{CRACK, REFINE\}. \tag{9}$$

Each kernel i contributes a latency $T_{f,i}$, which we model as a linear function of its two dominant cost factors: computational complexity and data movement. Specifically, for each kernel i in function f, we train the following simple predictive model:

$$T_{f,i} = w_{f,i}^{(1)} \cdot C_{f,i} + w_{f,i}^{(2)} \cdot D_{f,i} + b_{f,i}, \tag{10}$$

where $C_{f,i}$ represents the dominant computational complexity of kernel i, and $D_{f,i}$ denotes the dominant data movement cost. The terms $w_{f,i}^{(1)}$ and $w_{f,i}^{(2)}$ are learned coefficients corresponding to computation and data movement, respectively. Finally, $b_{f,i}$ is a learned parameter that accounts for fixed kernel execution overheads.

We gather measurement with micro-benchmarks for each kernel, by varying input parameters and measurig execution latency to fit a regression model, learning $w_{f,i}^{(1)}, w_{f,i}^{(2)}, b_{f,i}$.

Our approach eliminates the need for handcrafted analytical performance models that map the kernels into the number of cycles per operation and are tied to specific implementations and hardware configurations, making them hard to adapt or extend. With our method, switching hardware or optimizing kernels only requires re-running microbenchmarks and refitting the linear model. Another advantage is the low computational cost of inference, enabling continuous use during query execution. However, modeling execution time as a simple linear function has limitations. While it captures dominant compute and memory trends, it overlooks system-level nonlinearities. Factors like caching effects, NUMA constraints, SIMD optimizations (e.g., in FAISS), and thread contention introduce discrete performance variations that a linear model cannot fully express.

Nevertheless, our empirical results show that the cost model performs well across all benchmarks. Importantly, we always use the true measured times T_{build} and T_{search} in our budget constraint Equation 8. This ensuring any overestimation by the predictive model is corrected after execution, and an underestimation will hold any future build operations until sufficient queries arrive, increasing the total budget. In our testing, the model achieved moderate to high R^2, explaining between 40 and 95% of the total variation, with the *RMSE* remaining in the millisecond to low second range. We find this to be acceptable performance from our model since we benefit from low inference latency, and the cost budget-based control mechanism works effectively on all datasets, as we show in our experiments.

Kernel	Compute	Data Movement
Get Local Region*	$\lvert C_{\text{buffered}}\rvert \times \lvert C\rvert$	$\lvert P_{\text{local}}\rvert \times d$
Commit Reorg (w/o A_P^{dyn})	$\lvert P_{\text{local}}\rvert \times \lvert C_{\text{local}}\rvert \times d + \lvert C\rvert$	$\lvert P_{\text{local}}\rvert \times d$
Commit Reorg (w/ A_P^{dyn})	$\lvert C\rvert$	$\lvert P_{\text{local}}\rvert \times d$
Update Centroids	$\lvert P\rvert \times d$	$(\lvert P\rvert + \lvert C\rvert) \times d$
Local K-Means	$n_{\text{iter}} \times \lvert P_{train}\rvert \times \lvert C_{\text{local}}\rvert \times d$	$\lvert C\rvert \times d$

Table 2: Features we used for each kernel. * For REFINE, $C_{\text{buffered}} = 1$

Table 3: AV-Tree vs. CrackIVF

	Cumulative Time at i-th Query (seconds)					
	10^0	10^1	10^2	10^3	10^4	10^5 (Total)
AV-Tree	0.078	0.747	7.601	75.369	825.709	1264.175
CrackIVF	1.107	1.166	1.738	15.983	98.295	516.881

5 EXPERIMENTS

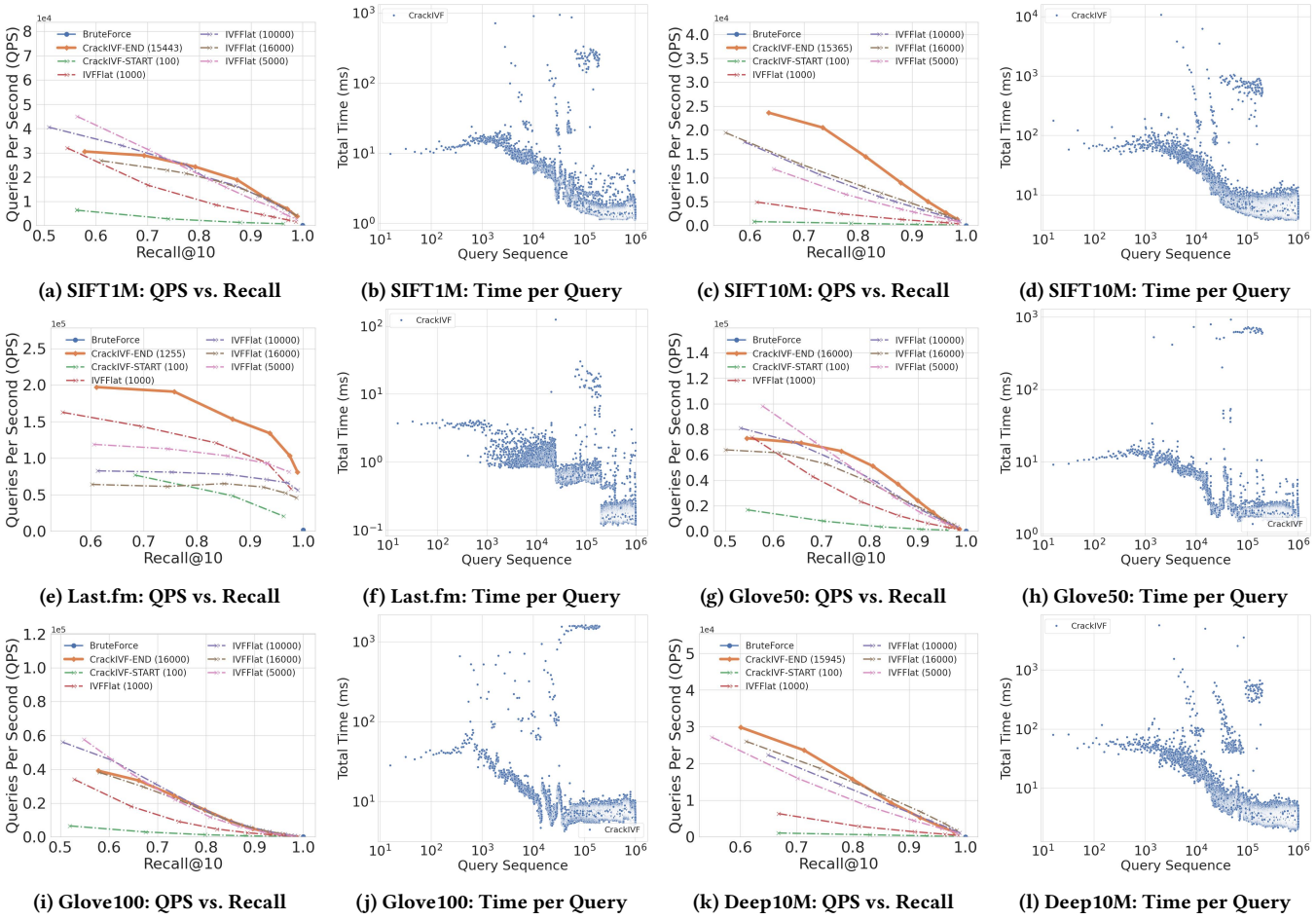
5.1 Experimental Setup

We run on a dual-socket AMD EPYC 7V13 system (128 cores, 3.7 GHz, 512GB RAM) from the AMD-ETHZ HACC cluster [61–63]. Unless otherwise stated, experiments use 16 cores, the number at which the memory bandwidth saturates, and use an equal query batch size to benefit from inter-query parallelism.

Datasets: We evaluate on standard ANN search benchmarks for text and image-based retrieval, GloVe[70], SIFT[41], DEEP[7], and Last.fm[11]. The latter, being an example with a high query skew, is used for music recommendation. The SIFT1M and SIFT10M, contain 128-dimensional SIFT descriptors, while DEEP10M consists of 96-dimensional embeddings from a fully connected GoogLeNet model [84]. These datasets are subsets of the original billion-scale collections, with 1M and 10M referring to the number of data points in each slice. Both use L2 distance as a metric. GloVe, sourced from ANN-Benchmarks [6], represents text retrieval. It contains 1.18M varying dimensional data points and uses cosine similarity, which we implement by *l2-normalizing* and computing inner product (IP) similarity. Last.fm is also taken from [6] contains 64-dimensional vectors and uses the IP similarity metric. We found that the provided query set is highly skewed, allowing us to test how CrackIVF behaves in such a scenario. Last.fm provides 50k unique queries, while the other datasets provide 10K unique queries. Thus, we replicate them until we match our target scales (up to 1M).

Baselines: Our comparisons cover both exact nearest neighbor (ENN) and approximate nearest neighbor (ANN) search approaches. For ENN, we use Brute Force, i.e., a linear scan of the data and AV-Tree [45], as a cracking baseline. For ANN search, we compare against FAISS IVF indexes, which in our figures are indicated as IVFFlat, with the number of partitions chosen in parentheses, e.g., IVFFlat (1000). We vary the number of partitions from 1,000 to 16,000, following FAISS guidelines [24], which recommend this range for our dataset scales. Our method, CrackIVF is initialized with 100 partitions to maintain near brute force startup time.

All experiments have the same fixed parameters, with values set as explained in (Sections 4.2, 4.3). Finally, for the partition-based indices, we have CrackIVF and IVFFlat baselines, where the *nprobe* is set to always achieve 90–95% Recall@10. The QPS-Recall plots show the Pareto frontier when varying *nprobe* across all indexes.



(a) SIFT1M: QPS vs. Recall **(b) SIFT1M: Time per Query** **(c) SIFT10M: QPS vs. Recall** **(d) SIFT10M: Time per Query**

(e) Last.fm: QPS vs. Recall **(f) Last.fm: Time per Query** **(g) Glove50: QPS vs. Recall** **(h) Glove50: Time per Query**

(i) Glove100: QPS vs. Recall **(j) Glove100: Time per Query** **(k) Deep10M: QPS vs. Recall** **(l) Deep10M: Time per Query**

Figure 3: Queries Per Second (QPS) vs. Recall and Time per each Query batch for the entire trace across different datasets.

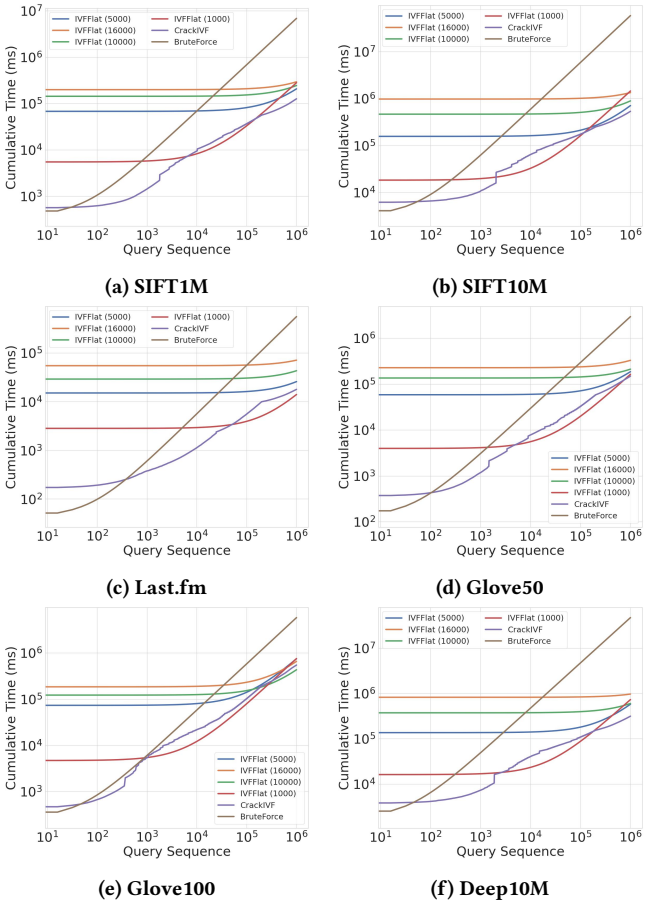
5.2 Comparison with AV-Tree:

To compare with AV-Tree [45] fairly, we set our CrackIVF to run with a single thread and make sure we probe enough partitions to get an average of more than 99% recall across all query scales. We set the cracking threshold of AV-Tree to 128, which we find gives the best total runtime. We show the cumulative time for a trace of 100K queries over the SIFT 1M dataset in Table 3. Unlike ours, AV-Tree does not pre-build the index, therefore, it has an advantage over us for the first 10 queries. However, our method processes each query faster and outperforms AV-Tree by a large margin after the first few queries. The cumulative time of AV-Tree increases at a slower rate with more queries, meaning that the query time is getting faster, and AV-Tree is improving over time. Nevertheless, our CrackIVF is 2.45x faster than AV-Tree in the end, even on a single-threaded ENN search. This shows the effectiveness of ANN approaches, where a less than 0.01% Recall error leads to a large gain in performance. For the rest of the experiments, we focus on the ANN search setting.

5.3 Does CrackIVF improve over time?

To measure how CrackIVF improves over time, we provide two graphs for each of the datasets and group them in Figure 3. The QPS vs. Recall plots measure the maximum achievable Queries Per Second (QPS) across varying Recall targets. This is the trade-off under which the ANN search operates. Having a large gain in search speed (QPS) for a small decrease in search quality (Recall). By the end of the 1 million query trace, indicated by *CrackIVF-END* (orange line), we are **consistently at the Pareto frontier of QPS-Recall across on all datasets**. On SIFT10M and Last.fm, CrackIVF is by far the best-performing index. Specifically on Last.fm, CrackIVF with 1255 final partitions, achieves 50% higher QPS for Recall@10 = 0.9, than a 10x larger *IVFFlat(16000)* index. We attribute this to CrackIVF's approach of only allocating new cracks based on the query distribution, and since Last.fm is the dataset with the highest skew, where only 30% of the vector spaces is accessed (Figure 2b), we can fully partition that 30% of the space with a relatively small number of cracks to achieve the best performance.

In Figure 3, we also plot the Response Time per query batch across the entire 1 million query trace. We can see that CrackIVF consistently improves over time, showing drops in response time

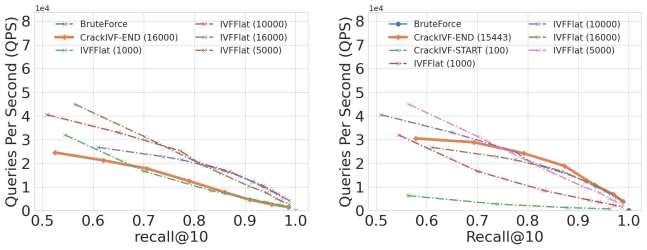


(a) SIFT1M (b) SIFT10M

(c) Last.fm (d) Glove50

(e) Glove100 (f) Deep10M

Figure 4: Cumulative time plots across datasets.



(a) WHERE=OFF: QPS vs Recall (b) WHERE=ON: QPS vs Recall

Figure 5: Comparison of QPS-Recall, showing effect of removing the "where" to CRACK and REFINE mechanism

after incremental reorganization operations. The outliers can all be attributed to these *CRACK* and *REFINE* reorganization operations. This is evidence for having control mechanisms to ensure expensive operations are only applied when necessary and why CrackIVF relies on buffering crack operations. The downstream effect of buffering and committing a large number of multiple cracks at once is only visible in one dataset, Last.fm, where it creates a step-like response in the plot, as seen in Figure 3f. This sudden drop in response time when adding a lot of cracks at once can also be seen in Figure 2a, when going from 1000 to 5000 partitions.

5.4 Does CrackIVF Minimize Cumulative Time?

Our core motivation in this work has been to provide an index that can be deployed across data lakes under scenarios where the number of queries that will arrive for each dataset and utilize the index is unknown. The ideal index for such a scenario should be able to minimize the cumulative time spent on both build and search operations in cases where a dataset may receive a few queries as well as millions.

We evaluate this specific aspect of index performance, across all datasets, with the cumulative time experiments, shown in Figure 4. For our measurements, we run a trace of 1 million queries for each dataset. In all baseline methods, we include the upfront build cost as the time before they answer the first query. The smaller *IVFFlat (1000)* indexes have relatively low startup costs but do not scale well past 10k queries since their cumulative time rises sharply due to having higher query batch response times than the alternatives. The larger indexes, beyond 5000 partitions, have a huge upfront build cost, which may pay off but only if a dataset receives >1 million queries. The "no index" approach of the *Brute-force* method has the minimum initialization time and is thus the best choice when the number of queries is low (<100).

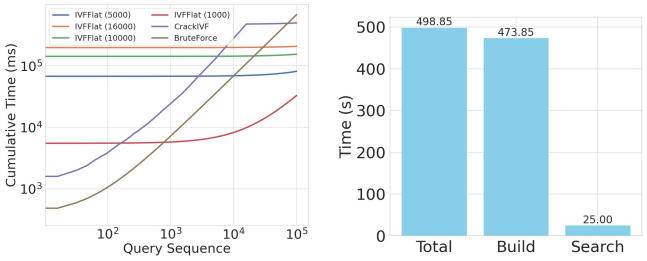
CrackIVF consistently achieves near-minimum cumulative time across all query scales. It achieves up to 1000x lower initialization time than larger indexes (Figure 4d) and can consistently answer 100K to 1M queries before comparable indexes finish building. For example on SIFT10M (Figure 4b), the initialization time cost is 10x lower than *IndexFlat(5000)* and 100x lower than *IndexFlat(16000)*, yet it processes 100K and 1M queries respectively, before they baselines indexes complete construction, while at the same time converging to higher QPS-Recall performance. Similar results can be observed in all other datasets.

The fact that CrackIVF achieves the Pareto optimal result in QPS-Recall by the end of our trace means that, by extrapolation, it will always maintain the minimum cumulative time across all query scales beyond 1M queries.

5.5 Control Mechanisms Ablation Study

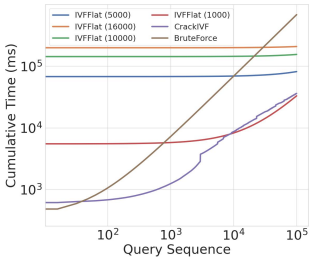
CrackIVF uses two control mechanisms for the *CRACK* and *REFINE* build operations, as detailed in Section 4.2 and Section 4.3. The first mechanism, controlling where they should be applied, uses a set of heuristic rules. The second mechanism, controlling when they are executed, is a cost-budget-based approach combined with a predictive model to balance search and build times. This section demonstrates the importance of these controls in achieving optimal performance and additionally highlights the effect of changing the parameter default value $min\_pts = 2$, which changes the heuristic rule "Don't Crack: Rule 1" (Section 4.2).

Turning off the control mechanism for "where": To illustrate the importance and accuracy of the heuristic rules for where to *CRACK* and *REFINE*, we switch them OFF and see the effect (Figure 5). Specifically, when "WHERE=OFF", and no heuristic rule is used to decide where a crack or refine should happen, the index defaults to trying to apply these operations whenever there is enough

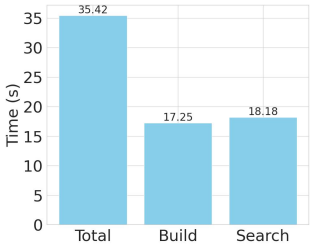


(a) Cumulative Time (OFF)

(b) Total Time Breakdown (OFF)



(c) Cumulative Time (ON)



(d) Total Time Breakdown (ON)

Figure 6: Comparison of cumulative time and total time breakdown, showing effect of removing the "when" to CRACK and REFINE mechanism

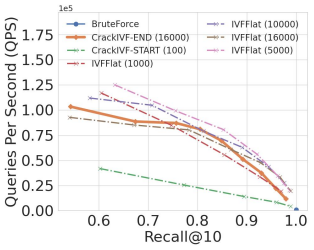
budget for them (the "when" mechanism is still ON). This seemingly random addition of new cracks and local refines leads to a final performance that is no better than the initial state of CrackIVF, before any query has been received (Figure 5a). On the other hand, when "WHERE=ON", and our heuristic rules are applied to make decisions, CrackIVF converges to the Pareto optimal QPS-Recall trade-off performance (Figure 5b). In both examples, we send an equal number of queries to the index and in the same order.

Turning off the control mechanism for "when":

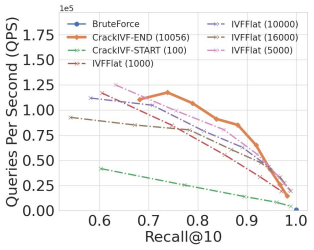
The budgeting mechanism, controlling when build operations are executed, helps balance the time spent on build vs search operations. By default, we use the parameter $a = 0.5$, i.e., at most 50% of the total time may be spent on *CRACK* or *REFINE*. The effect of turning this mechanism "WHEN=OFF" is shown in Figure 6a and Figure 6b. CrackIVF with the mechanism off defaults to cracking and refining after almost every query, and it only stops when the maximum number of partitions is reached (16,000). So even though the cracks and refines happen in regions that require them, they happen disproportionally often compared to the search overhead, leading to a final time spent on build operations that completely dominates the total time. On the other hand, when the mechanism is enabled ("WHEN=ON"), CrackIVF is able to efficiently balance the search and build costs (Figure 6b and Figure 6c). These runs are for the same 100k query trace.

Varying heuristic rule parameters:

CrackIVF's heuristic rules use default parameters (Section 4.2). To avoid parameter tuning, we did not vary these values across experiments. However, testing revealed that the *min_pts* parameter can impact performance, dependent on the dataset. *min_pts* defines the minimum number of points that must be stolen by a new crack



(a) QPS vs Recall (*min_pts* = 2)



(b) QPS vs Recall (*min_pts* = 32)

Figure 7: Comparison of QPS vs Recall for different *min_pts* values.

for it to be considered a "good" and subsequently buffered. On the GloVe-25 dataset, changing *min_pts* from 2 to 32 notably improved the QPS-Recall performance of the final index (Figure 7). Originally, we set *min_pts* = 2, a very small value, to ensure we avoid adding empty cracks to the index. We assumed that any imbalances could be fixed by future *REFINE* operations that redistribute points locally, which works for all datasets except for GloVe-25.

Although we can not make a definite statement, we hypothesize that this can be attributed to the fact that this dataset only has 25-dimensional embeddings. This leads to search times that are very low, and subsequently, our build operations budget grows slower. Thus, there are too many imbalances introduced with *min_pts* = 2, and a smaller budget for future *REFINE* operations to correct them. By increasing *min_pts* to 32, we reduced the total number of local imbalances by avoiding to ever commit excessively small cracks, which in turn reduces the number of *REFINE* needed to reach a good final performance.

A second hypothesis comes from closer inspection of the Figures, in which the better performing index converges to 10056 partition vs the default 16000. (Figure 7a, Figure 7b). But, we believe because GloVe-25 queries access the entire vector space evenly (Figure 2b) and it is a dataset with (1.18M points), we do not think a larger number of partitions hinders performance.

Nevertheless, we show that parameter values do matter across datasets, so index tuning can become a consideration. But for this work, across all the datasets we tested, outside of GloVe-25, the parameter defaults we have chosen have reliable performance.

Takeaways: Our results show that CrackIVF minimizes the cumulative costs across a large variety of individual datasets tested, but to fully grasp the potential performance benefits in embedding lakes, it's essential to consider the aggregate across all datasets' cumulative time savings and as the number of queries varies. For systems managing potentially thousands or millions of datasets, all saving are scaled equivalently, and choosing a separate index for each dataset is near-impossible, so the benefit from using CrackIVF is greater, as there is no one-size-fits all solution from the pre-built indexes.

From our experiments, we make the following suggestion: Brute Force should be used as the default method up until a dataset surpasses 10–100 queries, at which point CrackIVF becomes the optimal choice. Even though CrackIVF can start with one partition to simulate a Brute Force approach, it has a slight memory and

runtime overhead compared to it, from the setup time and storage of internal object structures and keeping track of the first few partition representatives. At scale, for embedding lakes with millions of datasets, even this small overhead can accumulate. Therefore, we recommend using CrackIVF once a dataset exceeds 100 queries and is thus likely to receive even more.

6 RELATED WORK

Vector Databases: Vector Data Management Systems (VDBMS) [15, 28, 72, 90, 94, 100], are systems that support efficient ANN search, and have gained significant popularity in both research and industry. This coincides with the increase in popularity of Large Language Models (LLMs) [13], that use Retrieval-Augmented Generation (RAG) techniques [5, 47], to extend their capabilities beyond what they have been trained on, in order to answer queries based on external data sources. Vector Database Management System applications extend beyond the above use-case, as they are also used as platforms for semantic search [58, 76], recommendation systems [48] and product search in e-commerce [86]. Given the importance of these operations, many existing data management systems are incorporating vector search capabilities [18, 71]. Many of these systems also support extensions to dense vector ANN search, through specialized methods, allowing for specific query types such as filtered search [27], sparse vector search [25], and hybrid search [87], which combines vector search with more traditional information retrieval techniques like BM-25 [75]. A recent survey of vector database management systems is found in [68].

Vector Search Indexes: Approximate Nearest Neighbor (ANN) search is a well-studied field with decades of research, resulting in a large number of indexes. Broadly, the two dominant categories of ANN vector search indexes are partition-based and graph-based indices. Inverted index (IVF) based methods, which fall under partition based category, include SCANN and SOAR [29, 83], which identify key mathematical properties of ANNS and use them to improve IVF index efficiency, SPANN [16], which shows how to leverage disk-based storage for search over billion-scale datasets along with supporting updates [96], and FAISS-IVF [23], an in-memory implementation utilizing GPUs [42] and quantization techniques like PQ [26, 40, 82]. In this work, we focus on IVF methods, and our cracking based approach could be extended for the above implementations. IVF indexes offer the best trade-off between index build time, search performance, and memory overhead, all of which are critical to balance for potential large scale deployments in embedding data lakes. Hash based methods [3, 21, 37, 38, 91] and tree based methods [51, 64, 89] also broadly fall under the partition-based index category. Finally, graph-based methods [22, 31, 52, 53, 88, 89] currently deliver the state-of-the-art performance in terms of the Queries Per Second vs. Recall trade-off, but this comes at high memory and index construction cost. Thus, graph based indexes are for now ill-suited for scenarios where minimizing the index construction cost and low memory overhead is a important, for example in embedding lakes where a very large number of indexes is expected to be constructed.

Adaptive indexes for specialized data types: Adaptive indexing techniques, such as Dumpy [92, 93] and ADS [101], have demonstrated significant effectiveness in data series similarity search. These indexes are specifically designed for time-series data and include methods like Symbolic Aggregate Approximation (SAX) [49] and indexable SAX (iSAX) [80]. In our work, we focus on ANN indexing methods that are the standard used by RAG systems, and due to their broad applicability across various data modalities which lack sequential relationships. That said, specialized, adaptive and incrementally constructed indexes are relevant candidates for embedding lake deployments, particularly for the subset of the datasets with inherent sequential dependencies. This includes energy, weather and financial data, as well as audio and video recordings [44, 56, 73, 79].

ANN index maintenance: Work on index maintenance focuses on mitigating performance degradation caused by data distribution shifts due to data updates (insertions and deletions). DeDrift [8] handles data or query distribution shifts and identifies small and large partitions to re-cluster together, but it keeps the total number of partitions fixed. LIRE [96] performs localized split and merge operations when partitions grow too large or become too small after insertions and deletions. Ada-IVF [60] monitors partition size changes from updates and the partition access distribution from the queries. It combines this information to prioritize the maintenance operations that will improve index efficiency. Similar to us, these approaches demonstrate the effectiveness of localized maintenance operations for IVF indexes and the advantages of adapting to query distribution. However, they cannot be applied directly for our usage, which is incremental index construction as the number of queries increases. These works all trigger maintenance operations directly or implicitly from incoming updates (insertions/deletions). Thus they will never trigger in our static setting, whereas we do since we follow the database cracking paradigm, that states "index maintenance should be a byproduct of query processing, not of updates" [36], and CrackIVF operations are triggered solely from the queries.

7 CONCLUSION AND FUTURE WORK

We introduced CrackIVF, a cracking-based IVF index for Approximate Nearest Neighbor Search (ANNS), aimed to be used with RAG systems over embedding data lakes. By dynamically adapting to increasing query workloads, it eliminates the need for costly up-front indexing, allowing immediate query processing. Our experimental results show that CrackIVF can process over 1 million queries before conventional methods even finish building an index, achieving 10-1000x faster initialization times. This makes it particularly effective for cold data, infrequently accessed datasets, and rapid access to unseen data. As future work, we want to adapt CrackIVF to handle query and data distribution shifts and identify reliable methods to dynamically set parameters across diverse datasets.

8 ACKNOWLEDGMENTS

We would like to thank AMD Research for the generous donation of the equipment in the ETHZ-AMD Heterogeneous Accelerated Compute Cluster (HACC) (https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html) that was used to run the experiments reported in this paper.

REFERENCES

[1] Qi An, Chihua Ying, Yuqing Zhu, Yihao Xu, Manwei Zhang, and Jianmin Wang. 2025. LEDD: Large Language Model-Empowered Data Discovery in Data Lakes. *arXiv preprint arXiv:2502.15182* (2025). https://arxiv.org/abs/2502.15182 Presented at the ACM SIGMOD International Conference on Management of Data, June 22–27, 2025, Berlin, Germany.

[2] Eric Anderson, Jonathan Fritz, Austin Lee, Bohou Li, Mark Lindblad, Henry Lindeman, Alex Meyer, Parth Parmar, Tanvi Ranade, Mehul A Shah, et al. 2024. The Design of an LLM-powered Unstructured Analytics System. *arXiv preprint arXiv:2409.00847* (2024).

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. *Advances in neural information processing systems* 28 (2015).

[4] Michael Armbrust, Reynold S. Xin, Shixiong Lian, Yin Huai, Cheng Li, Tathagata Mukherjee, Erik Nijkamp, Kay Ousterhout, Ruchi Paranjpye, Kartik Ramasamy, Dalton Soh, Alexey Tumanov, Burak Yavuz, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. *CIDR* (2021). https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[5] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. Retrieval-based language models and applications. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 6: Tutorial Abstracts)*. 41–46.

[6] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

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

[8] Dmitry Baranchuk, Matthijs Douze, Yash Upadhyay, and I Zeki Yalniz. 2023. Dedrift: Robust similarity search under content drift. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 11026–11035.

[9] Omar Benjelloun, Shiyu Chen, and Natasha Noy. 2020. Google dataset search by the numbers. In *International Semantic Web Conference*. Springer, 667–682.

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

[11] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*.

[12] Dan Brickley, Matthew Burgess, and Natasha Noy. 2019. Google Dataset Search: Building a search engine for datasets in an open Web ecosystem. In *The world wide web conference*. 1365–1375.

[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[14] Adriane Chapman, Elena Simperl, Laura Koesten, George Konstantinidis, Luis-Daniel Ibáñez, Emilia Kacprzak, and Paul Groth. 2020. Dataset search: a survey. *The VLDB Journal* 29, 1 (2020), 251–272.

[15] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3772–3785.

[16] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[17] Zui Chen, Zihui Gu, Lei Cao, Ju Fan, Samuel Madden, and Nan Tang. 2023. Symphony: Towards Natural Language Query Answering over Multi-modal Data Lakes.. In *CIDR*.

[18] Oracle Corporation. 2025. Oracle AI Vector Search. https://www.oracle.com/database/ai-vector-search/. Accessed: 2025-03-01.

[19] Data Science Association. 2016. Hadoop Vendor Evaluations 2016. https://www.datascienceassn.org/sites/default/files/Hadoop%20Vendor%20Evaluations%202016.pdf Accessed: 2025-02-16.

[20] Databricks. 2024. Vector Search in Databricks. https://docs.databricks.com/en/generative-ai/vector-search.html?utm_source=chatgpt.com Accessed: 2025-02-16.

[21] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[22] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. 577–586.

[23] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[24] Facebook Research. 2025. Guidelines to Choose an Index. https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index. Accessed: 2025-02-17.

[25] Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE: Sparse lexical and expansion model for first stage ranking. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2288–2292.

[26] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2013), 744–755.

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

[28] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. 2022. Manu: a cloud native vector database management system. *arXiv preprint arXiv:2206.13843* (2022).

[29] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.

[30] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[31] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, Vol. 22. 1312.

[32] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland HC Yap. 2012. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *arXiv preprint arXiv:1203.0055* (2012).

[33] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. 2009. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer.

[34] P Bryan Heidorn. 2008. Shedding light on the dark data in the long tail of science. *Library trends* 57, 2 (2008), 280–299.

[35] Pedro Holanda, Matheus Nerone, Eduardo C de Almeida, and Stefan Manegold. 2018. Cracking KD-Tree: The First Multidimensional Adaptive Indexing (Position Paper).. In *DATA*. 393–399.

[36] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

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

[38] Prateek Jain, Brian Kulis, and Kristen Grauman. 2008. Fast image search for learned metrics. In *2008 IEEE Conference on computer vision and pattern recognition*. IEEE, 1–8.

[39] Raj Jain. 1990. *The art of computer systems performance analysis*. john wiley & sons.

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

[41] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 861–864.

[42] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[43] Richard M Karp, Rajeev Motwani, and Prabhakar Raghavan. 1988. Deferred data structuring. *SIAM J. Comput.* 17, 5 (1988), 883–902.

[44] Kunio Kashino, Gavin Smith, and Hiroshi Murase. 1999. Time-series active search for quick retrieval of audio and video. In *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings. ICASSP99 (Cat. No. 99CH36258)*, Vol. 6. IEEE, 2993–2996.

[45] Konstantinos Lampropoulos, Fatemeh Zardbani, Nikos Mamoulis, and Panagiotis Karras. 2023. Adaptive indexing in high-dimensional metric spaces. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2525–2537.

[46] LanceDB Developers. 2024. Lance: Modern Columnar Data Format for ML. https://github.com/lancedb/lance Accessed: February 16, 2025.

[47] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[48] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: fast and exact inner product retrieval in recommender systems. In *Proceedings*

of the 2017 ACM International Conference on Management of Data. 835–850.

[49] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. 2003. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*. 2–11.

[50] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. *Conference on Innovative Data Systems Research (CIDR)* (2025).

[51] Ting Liu, Andrew Moore, Ke Yang, and Alexander Gray. 2004. An investigation of practical approximate nearest neighbor algorithms. *Advances in neural information processing systems* 17 (2004).

[52] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.

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

[54] Merrill Lynch. 1998. *Enterprise Information Portals: Industry Overview*. Technical Report. Merrill Lynch. https://web.archive.org/web/20110724175845/http://ikt.hia.no/perep/eip_ind.pdf Accessed: 2025-02-16.

[55] Renée J Miller. 2018. Open data integration. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2130–2139.

[56] Katsiaryna Mirylenka, Vassilis Christophides, Themis Palpanas, Ioannis Pefkianakis, and Martin May. 2016. Characterizing home device usage from wireless traffic time series. In *19th International Conference on Extending Database Technology (EDBT)*.

[57] MIT Sloan School of Management. 2021. *Tapping the Power of Unstructured Data*. https://mitsloan.mit.edu/ideas-made-to-matter/tapping-power-unstructured-data Accessed: 2025-02-16.

[58] Bhaskar Mitra, Nick Craswell, et al. 2018. An introduction to neural information retrieval. *Foundations and Trends® in Information Retrieval* 13, 1 (2018), 1–126.

[59] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Umar Farooq Minhas, Jeffery Pound, Cedric Renggli, Nima Reyhani, Ihab F Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. 2024. Incremental IVF Index Maintenance for Streaming Vector Search. *arXiv preprint arXiv:2411.00970* (2024).

[60] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-throughput vector similarity search in knowledge graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[61] Javier Moya, Matthias Gabathuler, Mario Ruiz, and Gustavo Alonso. 2023. fpgasystems/hacc: ETHZ-HACC. Zenodo. https://doi.org/10.5281/zenodo.8340448 https://doi.org/10.5281/zenodo.8340448.

[62] Javier Moya, Mario Ruiz, and Gustavo Alonso. 2023. fpgasystems/sgrt: ETHZ-SGRT. Zenodo. https://doi.org/10.5281/zenodo.8346565 https://doi.org/10.5281/zenodo.8346565.

[63] Javier Moya, Mario Ruiz, and Gustavo Alonso. 2024. fpgasystems/hdev: HACC Development. Zenodo. https://doi.org/10.5281/zenodo.14202998 https://doi.org/10.5281/zenodo.14202998.

[64] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence* 36, 11 (2014), 2227–2240.

[65] Matheus Agio Nerone, Pedro Holanda, Eduardo C De Almeida, and Stefan Manegold. 2021. Multidimensional adaptive & progressive indexes. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 624–635.

[66] OpenAI. 2023. OpenAI Embeddings. Online. https://platform.openai.com/docs/guides/embeddings Accessed: 2023-XX-YY.

[67] OpenAI. 2024. Introducing ChatGPT Search. https://openai.com/index/introducing-chatgpt-search/ Accessed: 2025-02-16.

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

[69] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: query-aware spatial incremental index. In *21st International Conference on Extending Database Technology (EDBT)*.

[70] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[71] pgvector contributors. 2025. pgvector: Open-Source Vector Similarity Search for PostgreSQL. https://github.com/pgvector/pgvector. Accessed: 2025-03-01.

[72] Pinecone. 2025. Pinecone - Vector Database. http://pinecone.io Accessed: 2025-03-01.

[73] Usman Raza, Alessandro Camerra, Amy L Murphy, Themis Palpanas, and Gian Pietro Picco. 2015. Practical data prediction for real-world wireless sensor networks. *IEEE Transactions on Knowledge and Data Engineering* 27, 8 (2015), 2231–2244.

[74] Google Research. 2023. SOAR: New Algorithms for Even Faster Vector Search with ScaNN. Google Research Blog. https://research.google/blog/soar-new-algorithms-for-even-faster-vector-search-with-scann/ Accessed: February 22, 2025.

[75] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[76] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[77] Hanan Samet. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)* 16, 2 (1984), 187–260.

[78] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2016. An experimental evaluation and analysis of database cracking. *The VLDB Journal* 25 (2016), 27–52.

[79] Dennis Shasha. 1999. Tuning time series queries in finance: Case studies and recommendations. *IEEE Data Eng. Bull.* 22, 2 (1999), 40–46.

[80] Jin Shieh and Eamonn Keogh. 2008. i SAX: indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 623–631.

[81] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. 2022. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR, 177–189.

[82] Sivic and Zisserman. 2003. Video Google: A text retrieval approach to object matching in videos. In *Proceedings ninth IEEE international conference on computer vision*. IEEE, 1470–1477.

[83] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2024. SOAR: improved indexing for approximate nearest neighbor search. *Advances in Neural Information Processing Systems* 36 (2024).

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

[85] Nan Tang, Chenyu Yang, Zhengxuan Zhang, Yuyu Luo, Ju Fan, Lei Cao, Sam Madden, and Alon Halevy. [n.d.]. Symphony: Towards Trustworthy Question Answering and Verification using RAG over Multimodal Data Lakes. ([n. d.]).

[86] Christophe Van Gysel, Maarten de Rijke, and Evangelos Kanoulas. 2016. Learning latent vector spaces for product search. In *Proceedings of the 25th ACM international on conference on information and knowledge management*. 165–174.

[87] Vespa.ai. 2024. Hybrid Search. https://docs.vespa.ai/en/tutorials/hybrid-search.html Accessed: 2024-03-01.

[88] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. 2012. Scalable k-nn graph construction for visual descriptors. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 1106–1113.

[89] Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian-Sheng Hua. 2013. Trinary-projection trees for approximate nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 36, 2 (2013), 388–403.

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

[91] Jingdong Wang, Ting Zhang, Nicu Sebe, Heng Tao Shen, et al. 2017. A survey on learning to hash. *IEEE transactions on pattern analysis and machine intelligence* 40, 4 (2017), 769–790.

[92] Zeyu Wang, Qitong Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Dumpy: A compact and adaptive index for large data series collections. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[93] Zeyu Wang, Qitong Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2024. DumpyOS: A data-adaptive multi-ary index for scalable data series similarity search. *The VLDB Journal* 33, 6 (2024), 1887–1911.

[94] Weaviate. 2025. Weaviate - Vector Database. http://weaviate.io Accessed: 2025-03-01.

[95] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*.

[96] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 545–561.

[97] Peter N Yianilos. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Soda*, Vol. 93. 311–21.

[98] Fatemeh Zardbani, Nikos Mamoulis, Stratos Idreos, and Panagiotis Karras. 2023. Adaptive indexing of objects with spatial extent. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2248–2260.

[99] Ce Zhang, Jaeho Shin, Christopher Ré, Michael Cafarella, and Feng Niu. 2016. Extracting databases from dark data with deepdive. In *Proceedings of the 2016 International Conference on Management of Data*. 847–859.

[100] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. 2023. {VBASE}: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 377–395.

[101] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1555–1566.