Cracking Vector Search Indexes

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

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

Gustavo Alonso
Systems Group, ETH Zurich, 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. The idea can be applied over data lakes, leading to the notion of embedding 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 dataset. Thus, we propose an adaptive, partition-based index, CrackIVF, that performs much better than up-front index building. CrackIVF starts answering as a small index, and only expands to improve performance 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 those built with conventional techniques. CrackIVF can often answer more than 1 million queries before other approaches have even built the index, achieving 10-1000x faster initialization times. This makes it ideal for cold or infrequently used data and as a way to bootstrap access to unseen datasets.

PVLDB Reference Format:
Vasilis Mageirakos, Bowen Wu, and Gustavo Alonso. Cracking Vector Search Indexes. PVLDB, 18(11): 3951 - 3964, 2025.
doi:10.14778/3749646.3749666

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/mageirakos/crack-ivf-vldb.

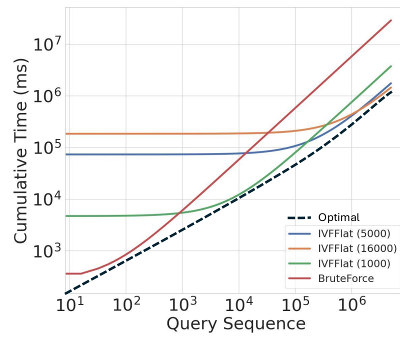


Figure 1: Total time to answer the queries submitted for different indexing strategies vs the number of queries submitted

1 INTRODUCTION

Large language models (LLMs) [12] can be complemented with retrieval-augmented generation (RAG) [46]. In RAG, external data is represented as *vector embeddings* (i.e., learned representations of data that results in a multidimensional vector). It is indexed and accessed through approximate nearest neighbor (ANN) search so that LLMs can answer queries on information they were not trained on. The effectiveness of RAG is tied to that of the ANN index structures, making them a critical component.

In practice, the vast majority of private, unstructured data remains untapped. It is estimated to be up to 80-90% of all data [53, 56], and is often stored in data lakes on open data formats [54]. As much as 70% of it remains unused [19], classified as "dark" data [33, 93]. While the datasets can be made discoverable [14], the actual underlying data in them often remains unindexed. As of March 2020, Google Research's Dataset Search [11] had indexed around 28 million structured and unstructured datasets [8] based on their metadata. Embedding-based vector search and RAG techniques could however help us go a step further by enabling direct retrieval and question answering on the underlying data itself. This requires embedding the data, as well as the creation of approximate nearest neighbor indexes so as to expose the data to RAG systems. We refer to such an approach as *embedding data lakes* (EDL), where unstructured data is stored alongside its vector representations and is queried by a pool of vector databases. Efforts in this direction are already ongoing. Databricks uses vector search [20] over their Lakehouse [3], and new custom storage formats [45] are being used to store embeddings on data lakes. Researchers are also exploring RAG-based techniques to query multi-modal data lakes [17, 79], as well as answering queries over unstructured data [1, 49].

The issue we address in this paper is ***index selection at scale***. Embedding data lakes, in principle, would require building millions of indexes across diverse datasets, modalities, workloads, and embedding models, which is neither feasible nor cost efficient. Some datasets may receive the majority 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 queried. On the other hand, skipping the indexing process and relying on a brute-force search is not scalable. Hence, data must be indexed, which raises two questions: **when and how to index the data**.

An optimal ANN index depends on the 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

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749666

cost (e.g., the large IVFlat configurations (5000, 16000 clusters) in the figure). Brute-force search, and smaller indexes (IVFlat 1000 clusters), minimize time-to-query for fast discovery but struggle at scale due to their higher query response times.

To tackle index selection at scale, we adopt a well-established strategy: carefully avoiding the need to build the full index upfront. We draw inspiration from deferred data structuring [41] and database cracking [32, 35, 73]. Similar approaches have been extended to multidimensional data [34, 62, 65, 92]. However, prior work does not scale to the embedding dimensionalities nor supports the k-Nearest Neighbour (k-NN) queries typical in RAG deployments. Recently, the AV-Tree was the first adaptive index for high-dimensional k-NN search [44]. 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 index structures to trade off a bit of accuracy for lower query response time and higher scalability. Nonetheless, the idea of dynamically building an index tailored to the workload is compelling. In this paper, we demonstrate how to do this efficiently in the context of RAGs by employing the same data structures [77] and systems [24] used for ANN search workloads [5, 76].

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 around which we can add a new partition, or crack. Likewise, its local region visited during search is a candidate for local refinement, allowing the index to grow and improve as more queries arrive. Efficiency is maintained through two control mechanisms: one deciding where to apply cracking and refinement in the search space, and another controlling when, balancing indexing and search times. In this work we focus on the static setting (i.e., no data or query distribution shifts) and evaluate CrackIVF on multiple standard open source datasets for ANN search [6, 10, 39, 66].

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. More partitions increase the upfront construction cost but improve query performance. This decision has to be made without knowing the future query workload. On the other hand, CrackIVF waits to build the index as 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. CrackIVF is an incrementally built index that asymptotically matches or surpasses the performance of pre-built indexes, while it has been able to answer queries along the way.

Across benchmarks, CrackIVF consistently outperforms prebuilt 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, 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.

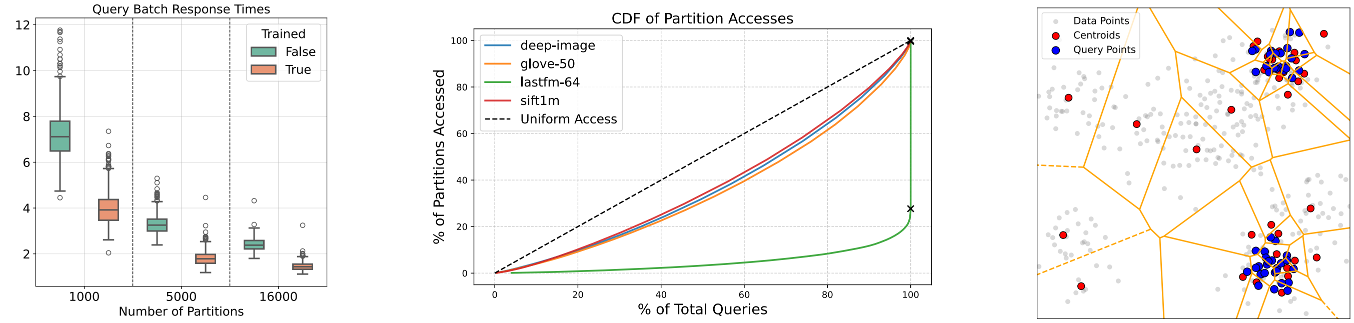
2 BACKGROUND

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)). A straightforward Exact Nearest Neighbor approach is a linear scan of every point, a.k.a *brute-force search*, incurring $O(|P| \cdot d)$ complexity. There exist more sophisticated *exact indexing* structures (e.g., KD-trees [9], R-trees [30]). However, due to the curse of dimensionality [89], performance degrades to near-linear-scan in higher dimensions, making them impractical when $|P|$ or d are large. With embeddings of hundreds to thousands of dimensions [63] and real-world datasets at the scale of billions of data points [76], RAG systems turn to Approximate Nearest Neighbor search techniques. They allow a small retrieval error ε in exchange for better runtime or memory usage. The approximation quality can 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, 77, 78]. 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} *representative vectors* for each partition, typically using k-means on a training subset $P_{\text{train}} \subset P$. Each data point is then assigned to the nearest representative. These assignments are typically stored in an *inverted file* (IVF) structure [77]. A widely adopted implementation of IVF indexes is the FAISS library from Meta [24], which we built CrackIVF on and use as a baseline. See Section 6 for other ANN index types.

3 SOLUTION OVERVIEW

Embedding lake deployments involve constructing ANNS indexes for each data set of interest. The goal is to minimize index construction and memory overhead while also ensuring high queries per second (QPS). Although other index types excel in specific areas, IVF performs well in all three. For example, HNSW [52], a graph-based index, can achieve higher QPS, but at the cost of much higher memory. For both IVF and HSNW, the memory-QPS trade-offs, Figure 5 in [5], "perform nearly indistinguishably" and better than alternatives, but as mentioned IVF is a much smaller index. The build times of IVF indexes have been shown to be 100× lower than any other index and 8000× lower than HNSW (Figure 10 in [5]). Thus, looking at indexing speed versus QPS trade-offs [70], graph-based methods struggle, where IVFs are the best (faiss-ivfpqfs[24], and ScaNN[29]). Focusing on a single index type doesn't solve the index selection problem. While IVF indexes reduce compute and storage costs, they are still paid upfront before the query workload is known. We show how to extend IVF to enable incremental construction, scaling performance based on the query volume, and amortizing build and storage costs across time.



(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 the number and refinement of partitions; (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

3.1 Cracking in high dimensions

An approach to avoid the upfront cost for indexing is *cracking*. In the original database cracking [35], the final state is where a single column is sorted. Over time, fewer points are physically reorganized reducing the overhead. In high-dimensional spaces, the having a final sorted state no longer holds, and space filling curves, like Z-order [72], do not scale past a few dimensions. Additionally, in a naive implementation, the physical reorganization happens with every incoming query. In high-dimensions, this becomes increasingly difficult. High-dimensional vectors require moving multiple bytes per dimension, increasing data movement costs. In IVF, moving points between inverted lists incurs random access overhead, with no eventual locality benefits. Furthermore, inverted lists are often implemented as memory-aligned contiguous arrays to allow for efficient SIMD-based operations, and they may require linear-time compaction even for a single vector to move. These factors contribute to the latency overhead, making physical reorganization after every query difficult. The challenge 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

Index construction can be divided into two operations that independently improve index performance, *i*) increasing the number of partitions and *ii*) refining partitions by using K-means, which helps to distribute points more evenly (Figure 2a). The best results consistently come after reaching a maximum index size, even with randomly chosen 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 also refine their locations using K-means.

Observation 1: *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 partitions sampled uniformly and K-means refinements performed globally. While simple and intuitive, this approach assumes equal importance across the 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 [58, 59], up to 85% of partitions 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 [6, 10, 39, 66]. 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: *Just as index construction can be separated in time, it can also be* ***separated in space****. Index build operations do not have to be applied uniformly across the vector space. Instead, they can be local to regions that are accessed. Adaptive index construction that follows the query distribution is one strategy to optimize for efficiency.*

In partition-based IVF indexes, operations performed during search and those performed during index construction are very similar. During search the distances of the query to centroids and nearby points are computed, and during index construction the distances of points to centroids are computed. This means that by treating the query as a potential new partition representative "centroid", we can both identify good locations to add new cracks, and reuse the intermediate computations of search to perform incremental index construction. This naturally leads to a cracking-based approach for IVF, where index construction can occur as a side effect of query execution.

Observation 3: *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

We propose a partition-based crackable IVF index for ANN search. It is designed to remain near the minimum cumulative time across the number of queries submitted, be constructed as the query load increases, and avoid indexing unused parts of the vector space.

At initialization, CrackIVF starts as a small IVF index, with coarse partitioning of the entire dataset. As queries come in, more partitions are added in the accessed regions related to the queries (Figure 2c). The index employs two localized and independent build operations: *CRACK* and *REFINE*. The former introduces new partitions, while the latter applies a localized K-means to refine them. These operations run independently across both time and region of the vector space. Queries and their local visited regions act as *CRACK* and *REFINE* candidates, determining where new partitions should be introduced. 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. Thus, intermediate results from search operations are reused to amortize the cost of construction. Additionally, using the queries as candidates for *CRACK* and *REFINE* enables CrackIVF to follow the query distribution in terms of when and where to execute build operations. Partitions that receive a larger number of queries will be proportionally more likely to be selected for cracking and refinement.

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 only when it is beneficial, while *REFINE* operations are executed eagerly, but infrequently, to improve local regions that queries are visiting. We introduce two control mechanisms to make these decisions. The first is a set of heuristic rules that determine the decision boundaries for 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 improve as the query workload demands increase.

4 CRACK-IVF

CrackIVF, is implemented in FAISS [24]. We describe the core logic of the algorithm and operations in Section 4.1 and Algorithm 1; the control mechanisms, including heuristic rules and cost models, in Sections 4.2 and 4.3. These aid in making accurate decisions about where and when to perform the build operations, *CRACK* and *REFINE* (Algorithms 2 and 3). The notation used is in Table 1.

4.1 Algorithm and Operations

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

SEARCH: During the search procedure of IVF, the local region to the query is identified by finding the $nprobe$ nearest partitions to the query. Subsequently, the point assignments to these local partitions are scanned to find the k nearest neighbors. The result includes the nearest point ids I, their distances D to the query, and the $C_{visited}$ ids of the local region cracks that the query visited.

Table 1: Notation used in algorithms and cost model.

Symbol	Description
$\lvert X\rvert$	Cardinality, e.g., $\lvert X\rvert = N$ if $X \in \mathbb{R}^{N\times d}$
$metric(a,b)$	Similarity metric, where in our work $metric \in$ {Eucl. Distance (L2), Inner Product (IP)}
$P \in \mathbb{R}^{\lvert P\rvert \times d}$	Data points, d-dimensional
$C \in \mathbb{R}^{\lvert C\rvert \times d}$	Committed cracks
$C_{\text{local}} \in \mathbb{Z}^{nprobe}$	Local region crack IDs
P_{local}	Local region points
P_{train}	K-means training points, where $\lvert P_{\text{train}}\rvert = \min(\lvert P_{\text{local}}\rvert, \lvert C_{\text{local}}\rvert \times \max_{\text{points}})$
n_{iter}	K-means iterations
$\max_{\text{points}}$	K-means max training sample per crack
$nlist$	Total inverted lists storing P, where $nlist = \lvert C\rvert$
$Q \in \mathbb{R}^{bs\times d}$	Query set per search
$D \in \mathbb{R}^{bs\times\lvert r\rvert}$, $I \in \mathbb{Z}^{bs\times\lvert r\rvert}$	kNN distances and point indices from *SEARCH*, where $\lvert r\rvert \in [k, \lvert P_{\text{visited}}\rvert]$
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
$A_P^* \in \mathbb{Z}^{\lvert P\rvert}$	Point assignments to crack IDs
$D_P^* \in \mathbb{R}^{\lvert P\rvert}$	Point distances to crack representative
$H_C^* \in \mathbb{N}^{\lvert C\rvert}$	Histogram of points per crack

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

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 rules that define the decision boundaries which 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. If it is, then it's beneficial to reassign the point to the new crack, and this assignment is buffered, to be committed when *CRACK* is executed, as explained in Section 4.3. *CRACK* is the procedure that performs the physical reorganization, synchronizing the buffered state of the index with the currently working state. The core individual kernels of the procedure are shown in Algorithm 2. It adds a 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 procedure is only executed when enough budget for build operations has been accumulated throughout the usage of the index (Line 18 Algorithm 1). This budget 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. Since *CRACK* is applied lazily, the heuristics governing where a crack is added and the cost-based control mechanism for when it is added are independent.

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: $T_{\text{search}} \leftarrow T_{\text{search}} + (\text{CURRENTTIME}() - T_{\text{start\_search}})$
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: $I_{\text{steal}} \leftarrow \{p \in I_{\text{local}} \mid D_{\text{local}}(p) \text{ op } D_P^{\text{dyn}}(p)\}$ {op = < for L2, > for IP}
9: $good\_crack\_candidate$ = CrackHeuristic(I_{steal}, 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: $enough\_buffered \leftarrow (|C_{\text{buffered}}| > C_{\text{min}})$
20: **if** can_afford_crack **and** $enough\_buffered$ **then**
21: \\ UNDO if *REFINE* invalidated previously good cracks
22: $C_{\text{buffered}} \leftarrow \{c \in C_{\text{buffered}} \mid \text{CRACKHEURISTIC}(c)\}$
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: $T_{\text{build}} \leftarrow T_{\text{build}} + (\text{CURRENTTIME}() - T_{\text{start\_crack}})$
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}, State$_{\text{true}}$)
36: **if** $good\_refine\_candidate$ **then**
37: $T_{\text{start\_refine}} \leftarrow$ CURRENTTIME() {Start timing REFINE}
38: REFINE(...)
39: $T_{\text{build}} \leftarrow T_{\text{build}} + (\text{CURRENTTIME}() - T_{\text{start\_refine}})$
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/ 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.

REFINE: Refinement is the second operation that improves the index. It is performed eagerly, focusing on the local region visited by the query. A local region is defined by the *nprobe* nearest partitions that the query visits during search. Algorithm 1 Line 7, is where the local region C_{local} for a query is extracted from the *SEARCH* result $C_{visited}$ of a query batch. The goal of REFINE is to immediately correct suboptimal assignments in the regions of the current index. Thus, any buffered cracks and assignments within the local region are not considered, as they are not part of the current index state. The decision to refine a region is based on heuristics evaluating the imbalance of point assignments in local cracks. 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 takes place when both the currently visited local region is a good candidate for refinement and there is sufficient computational budget to execute the operation. If the "where" and "when" heuristics for *REFINE* do not simultaneously agree, the next opportunity is when another query accesses the same region. *REFINE* is shown in Algorithm 3). It 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. *REFINE* has higher computational costs as it does not rely on pre-computed buffered assignments, and assignments are determined dynamically upon completion of the local K-means process. *CRACK* has a higher data-movement cost, since typically a larger region of the vector space is affected when cracks are added throughout.

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}.
3: **Commit Reorg (w/o 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.

Index State: Since *CRACK* is a lazy operation, the index has two concurrent states, the true current state and a dynamic buffered 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. *REFINE* operations directly alter the current index state, while cracking decisions are made based on the dynamic state. This can lead to scenarios that require us to undo buffering decisions. This is done to avoid 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 index state, we define:

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

where $* \in \{\text{dyn}, \text{true}\}$ denotes either the *dynamic* uncommitted or *true* committed state. The *dynamic state* consists of *dynamic assignments* A_P^{dyn}, *dynamic point distances* D_P^{dyn}, and a *histogram* H_C^{dyn} of both the true and buffered crack sizes. Similarly, the *true state* represents the true, committed state of assignments, distances, and crack sizes in the index. We extend the FAISS-IVF "`.add()`" operation to return the assignments and distances computed, which are an intermediate result, in order to initialize the state tracking. 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 at time t_0, so $State_{true} = State_{dyn}$. After a few queries, at t_1, assume that 2 good crack candidates have been buffered, both in the same 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} changing the assignments to C'_{local}, affecting S_{true}. Since *REFINE* improves the current state eagerly, the decision that the buffered cracks are good was taken at a prior time t_1, on a prior state C_{local}. Thus, at t_2, during *REFINE* the buffered cracks may now have had their points stolen back, and assigned to other refined cracks in C'_{local}. Now, while still $|C_{buffered}| = 2$, it may now contain bad cracks. Finally, when at t_3, it is a good time to execute a *CRACK* operation again, all buffered cracks are re-evaluated, as shown in lines 22-24 Algorithm 1, and only the good cracks that remain are committed.

Memory Overhead and Optimizations: The additional memory cost of maintaining $State_{\text{dyn}}$ and $State_{\text{true}}$ is $O(4|P| + 2|C|)$. It is negligible since $d \gg 4$ and the in-memory data alone are $O(|P| \times d)$. 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 many optimizations that a full-fledged implementation could utilize. 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 $State_{\text{true}}$ recomputed as needed. Full memory overhead is only necessary during bursty query loads with limited hardware resources, 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 [91] demonstrated how to concurrently execute queries by dispatching such tasks to background threads. In our work, the entire thread pool is dedicated either to search or cracking operations. We also assume a continuous query load at system capacity. As a result, the memory overhead for index state tracking is unavoidable. We further cover index maintenance prior work in Section 6.

4.2 Where to apply build operations?

Each arriving query is a crack candidate, and the visited region during index traversal is a candidate for refinement. This results in a continuous stream of candidates of varying quality. As discussed, not all vector space regions are equally important, and not all operations lead to performance gains. The **goal** is *to maintain a stream of high-quality candidates*, classified from incoming queries, enabling the index to grow by adding cracks in frequently accessed regions that need further partitioning, and refining poorly clustered regions with K-means. CrackIVF employs two heuristics, acting as a binary classification model for candidates. Each uses a set of decision rules that are distilled from our empirical observations of IVF index behavior. We do not claim to have defined an exhaustive list of decision rules for the model. Changing or learning the model and the parameters while still achieving the stated goal above, would not change the overall design of the index (Algorithm 1).

Where to CRACK (CrackHeuristic): We observe that the optimal performance is achieved at some maximum number of partitions (Figure 2a). Thus, for it to grow, we treat each incoming query as a *good crack candidate by default* and define a set of decision rules to classify when a query is a *bad* candidate. Query candidates classified as *good*, are buffered until a future *CRACK* operation. A *bad* candidate (returning *False*) is if *any* of the following hold:

Don't Crack: Rule 1 rejects queries that steal too few points:

$$|I_{\text{steal}}| < \text{min\_pts} \tag{2}$$

Don't Crack: Rule 2 prevents excessive partitioning, constraining local partition count based on the local number of points:

$$\frac{\text{total points in local region}}{\text{total cracks in local region}} > \text{pts\_crack\_thr} \tag{3}$$

We set min_pts = 2 in order to avoid adding empty cracks, and pts_crack_thr = 64 to ensure an average of at least 64 points per crack locally. Rule 2 is especially important in high skew datasets, where most crack candidates are for the same region.

Where to REFINE (RefineHeuristic): We observe that refining an index improves performance at each index size (Figure 2a), by better distributing points across partitions and repositioning centroids to better fit the data. Optimal performance is reached at the maximum index size, so we want to avoid getting stuck refining a small index. Unlike *CRACK*, where latency is amortized by buffering, *REFINE* is eagerly applied to immediately improve the current index, but it is also a costly operation. Thus, we treat *each incoming refine candidate as bad*, leaving more of the cost budget for cracks so that the index can grow. A refine candidate is the local visited region of the query, visited during search. It is a *good* candidate for refinement (returning *True*) if *any* of the following hold:

Refine: Rule 1 captures local imbalance:

$$\text{CV}_{\text{local}} = \frac{\sigma_{\text{local}}}{\mu_{\text{local}}} > \text{cv\_max} \tag{4}$$

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 imbalance using the global distribution:

$$\begin{aligned} \exists c_1, c_2 \quad \text{s.t.} \quad & S(c_1) \leq \text{cutoff}_{\text{low}}, \\ & S(c_2) \geq \text{cutoff}_{\text{high}} \end{aligned} \tag{5}$$

where c_1, c_2 are cracks in the local region, and $S(c) = H_C^{\text{true}}[c]$ represents the size (i.e., number of assignments) of crack c. The $\text{cutoff}_{\text{low}}$ and $\text{cutoff}_{\text{high}}$ correspond to cluster sizes at the *size_prctl* and 100 − *size_prctl* percentiles of the global partition sizes. This rule captures the unlikely scenario that a large and small crack globally are next to each other locally, indicating points can be better distributed.

We set cv_max = 2, meaning refinement is triggered if cluster sizes vary by at least twice the local mean, and size_prctl = 10, defining small and large cracks based on the 10th and 90th percentiles of global cluster sizes.

4.3 When to apply build operations?

To mitigate the overhead of constant physical reorganization, we use a build budget to restrict the fraction of time dedicated to *CRACK* and *REFINE* operations. We define T_{build} as the total measured time spent on previous build operations. 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 a *CRACK* operation in the current dynamic state or a *REFINE* operation in the true state. The build operation considered is specified by $f \in \{\text{CRACK}, \text{REFINE}\}$. To enforce the budget constraint, the total time spent on past build operations in addition to the estimated time if the current one is executed must remain within a fraction α of the system's total runtime:

$$T_{\text{build}} + \hat{T}_f \leq \alpha \cdot (T_{\text{build}} + T_{\text{search}} + \hat{T}_f). \quad (6)$$

T_{build} and T_{search} are directly measured from the execution time, while $\hat{T}_f$ is estimated using a predictive cost model. The parameter α defines the proportion of total execution time that can be allocated to indexing operations. We set $\alpha = 0.5$, which ensures a maximum of 50% of the total time at any given point is spent on build operations. 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. The idea comes from Figure 2a, where if a REFINE can achieve similar performance improvement at the current index size, it is better to wait until more cracks have been buffered, so that the performance improvement from index growth is substantial. Specifically, in our implementation, we set that minimum to 20% of the current index size, $C_{min} = 0.2 \times |C|$ (Line 19 Algorithm 1).

Predictive Cost Model: Estimating the execution time of the *CRACK* and *REFINE* operations happens after every incoming query. Thus, we must maintain low latency when making a prediction, while have accurate enough cost estimates to correctly use the budget constraint Eq. 6. For this, we use a simple first-order linear model fitted using multivariate regression [38]. 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. The execution time is dominated by the following kernels:

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}).

$$T_f = \sum_i T_{f,i}, \quad \forall f \in \{\textit{CRACK}, \textit{REFINE}\}. \quad (7)$$

These kernels are executed sequentially so the total execution time T_f is simply the sum of kernel execution times. 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. For each, we learn the following predictive model:

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

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	$n_{\text{iter}} \times \lvert P_{train}\rvert \times \lvert C_{\text{local}}\rvert \times d$	$\lvert C\rvert \times d$

$$T_{f,i} = w_{f,i}^{(1)} \cdot C_{f,i} + w_{f,i}^{(2)} \cdot D_{f,i} + b_{f,i}, \quad (8)$$

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 for computation and data movement and $b_{f,i}$ is a learned parameter that accounts for fixed overheads. We gather measurement with micro-benchmarks for each kernel, by varying input parameters and measurig execution latency to fit a regression model, learning $w_{f,i}^{(1)}, w_{f,i}^{(2)}, b_{f,i}$.

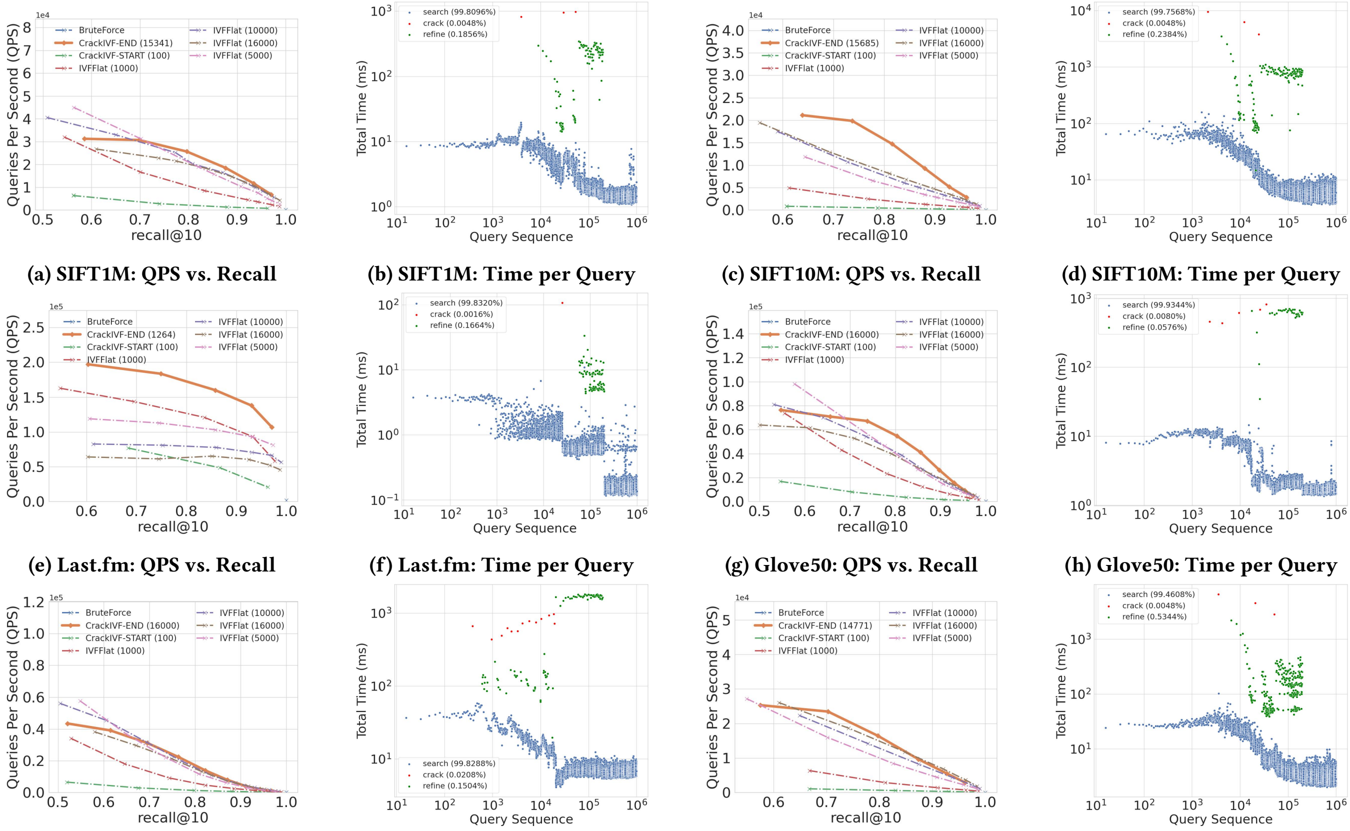
Our approach removes the need for exact analytical models coupled to specific implementations and hardware. Switching hardware requires re-running micro-benchmarks and refitting the linear model. The main limitation is that execution time is modeled as a linear function. While this captures dominant compute and memory trends, it overlooks system-level nonlinearities such as caching effects, NUMA constraints, and SIMD optimizations. Despite this, 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 the budget constraint (Equation 6), which help correct over and under estimations. In our tests, the model achieved moderate to high R^2 scores, explaining 40–95% of the total variance, with RMSE ranging from milliseconds to a few seconds.

5 EXPERIMENTS

We run on a dual-socket AMD EPYC 7V13 system (128 cores, 3.7 GHz, 512GB RAM). All experiments, except for Table 3, use 16 cores and a batch size of 16, the point at which memory bandwidth saturates. IVF and CrackIVF share the same SEARCH operation in FAISS and benefit equally from inter-query parallelism.

We evaluate on standard ANN search benchmarks, GloVe[66], SIFT[39], DEEP[6], and Last.fm[10]. SIFT1M and SIFT10M contain 128-dimensional SIFT descriptors; DEEP10M consists of 96-dimensional embeddings, with 1M and 10M being the number of data points in each slice. Both use L2 distance as a metric. GloVe contains 1.18M varying dimensional embeddings and uses cosine similarity, which we implement by *l2-normalizing* and computing the inner product (IP). Last.fm, contains 64-dimensional vectors and IP metric. The Last.fm query set is highly skewed, allowing us to test CrackIVF in such cases. Last.fm provides 50k unique queries, while the other datasets provide 10K unique queries. We replicate them until we match our target scales (up to 1M).

For ENN, we compare against Brute Force, and AV-Tree [44], which is a cracking baseline. For ANN search, we compare against FAISS IVFFlat indexes with the number of partitions chosen shown



(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.

in parentheses, e.g., IVFFlat (1000). We vary the number of partitions following FAISS guidelines [25], for our dataset scales. For CrackIVF, we use fixed parameters values for the control mechanisms, set as explained in (Sections 4.2, 4.3). We consider the index converged when the crack decision rules stop triggering and refines keep targeting the same region without effect, thus the index can no longer grow in size, nor improve at the current size. For CrackIVF and IVFFlat, *nprobe* is set to always achieve 90–95% Recall@10 in ANN. CrackIVF is initialized with 100 partitions on all experiments. We set this parameter by choosing the largest possible starting partitions that still give minimal startup cost. Because of FAISS parallelism, at 16 threads, the startup time for 100 partitions is identical or faster than starting with 1 partition. On a single thread for Table 3, the difference between them is only ≈120 ms.

5.1 Comparison with AV-Tree:

To the best of our knowledge, AV-tree [44] is the only similarity search index, whose underlying idea is based on cracking. It is a tree-based ENN cracking index targeting short-lived data up to one thousand queries. Unlike AV-Tree, CrackIVF is an IVF-based ANN cracking index, implemented around FAISS, offering parallelism, and targeting high dimensional RAG applications which could reach much larger query volumes. As AV-Tree is single-threaded and an ENN index, to compare fairly, we set CrackIVF and the IVF baselines 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 1M queries over the SIFT 1M dataset in Table 3. At the start, the 10^0-th query time includes the index construction costs of CrackIVF and IVF baselines. CrackIVF is initialized as a small 100 partition index which grows as more queries arrive, whereas AV-Tree does not pre-build any index. In this experiment, AV-Tree has an advantage over CrackIVF for the first 10 queries, as it can start answering immediately. However, CrackIVF processes each query faster and outperforms AV-Tree by a large margin after the

Table 3: AV-Tree vs. CrackIVF. † Cracking-based baselines.

	Cumulative Time at i-th Query (seconds)						
	10^0	10^1	10^2	10^3	10^4	10^5	10^6
IVF-1K	12.57	12.59	12.76	14.48	32.10	205.40	1938.60
IVF-5K	178.17	178.18	178.35	179.93	195.94	354.63	1943.02
AV-Tree†	0.08	0.75	7.48	75.43	830.0	1274.85	5721.29
CrackIVF†	1.10	1.16	1.73	13.18	55.33	233.11	1531.40

first few queries. The cumulative time of AV-Tree increases at a slower rate with more queries, showing AV-Tree is improving over time. Our CrackIVF is 3.7x faster than AV-Tree in the end of the trace. As cracking baselines, both CrackIVF and AV-Tree, vastly outperform IVF indexes on the low query scale. AV-Tree continues to outperform up to 1000 queries, matching original claims [44]. CrackIVF performance benefits continue even at 1 million queries. In the following experiments we focus on ANN.

5.2 Does CrackIVF improve over time?

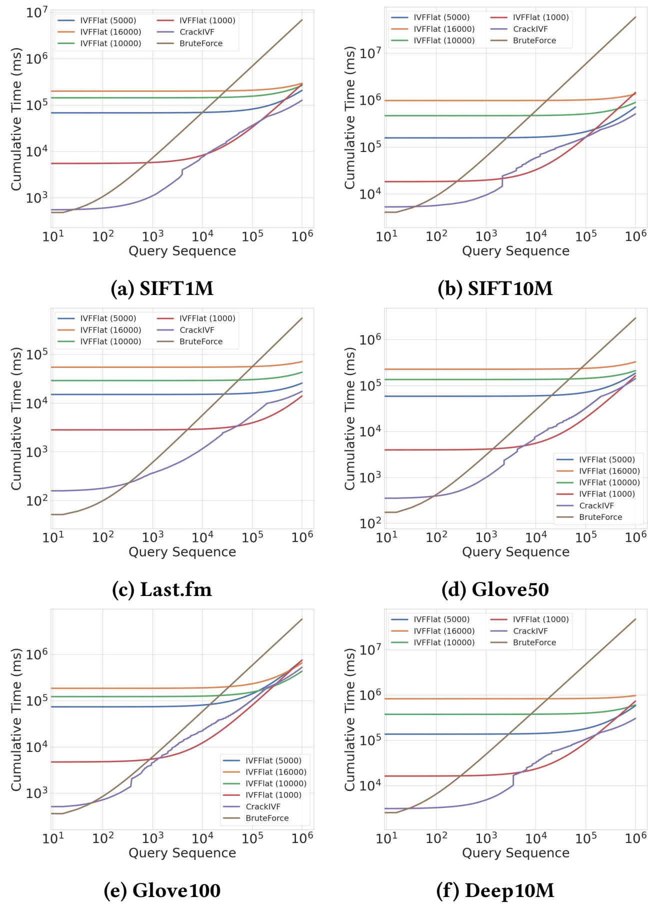
We measure how CrackIVF improves over time in Figure 3. QPS vs. Recall plots show the search performance trade-off in which ANN search operates under. In our experiments, by the end of the query trace, CrackIVF-END is consistently at or near the Pareto frontier across all tested datasets. On SIFT10M and Last.fm, CrackIVF is by far the best-performing index. Specifically on Last.fm, CrackIVF with 1264 final partitions, achieves ≈50% higher QPS for Recall@10 = 0.9, than any other index. We attribute this to CrackIVF's approach of allocating new cracks following the query distribution. Since Last.fm is the dataset with the highest skew (Figure 2b), we can fully partition the accessed space with a small number of cracks, achieving the best performance. In Figure 3, we also plot the response time per query batch across the entire query trace. CrackIVF search performance improves over time after incremental reorganization operations. The vast majority of outliers are due to the individual *CRACK* and *REFINE* reorganization operations. In our experiments, *CRACK* operations are less than 0.03% and *REFINE* less than 0.6% of the query trace.

5.3 Does CrackIVF Minimize Cumulative Time?

We run a trace of 1 million queries for each dataset and include the upfront build cost for all baselines as the time before they answer the first query (Figure 4). The smaller 1000 partition IVF indexes have relatively low startup costs but do not scale well past ≈10k queries. Larger indexes have a huge upfront build cost, which only pays off if a dataset receives >1 million queries. Brute-force has the minimum initialization time, making it the better choice when the number of queries is low (<100). In our experiments, CrackIVF achieves near-minimum cumulative time across all query scales. It achieves almost 1000x lower initialization time than larger indexes (Figure 4d) and can answer 100K to 1M queries before comparable indexes finish building. For example on SIFT10M (Figure 4b), the initialization time cost is ≈100x lower than IndexFlat(5000) and almost 1000x lower than IndexFlat(16000), yet CrackIVF answers 100k and 1M queries respectively, before the indexes finish building.

5.4 Does CrackIVF Reduce Total Distance Computations for Build Operations?

IVF indexes incur full indexing cost upfront over the entire vector space. Indexing large datasets (100M–1B+) in reasonable time requires GPUs, but even then, K-means training can take hours due to the increasing number of training points and partitions, which have a multiplicative effect in the number of pairwise distance computations (see Table 2 *K-means Compute*). CrackIVF amortizes this cost over time using *REFINE* operations, which apply K-means locally on small subsets of points and partitions. As a result, total

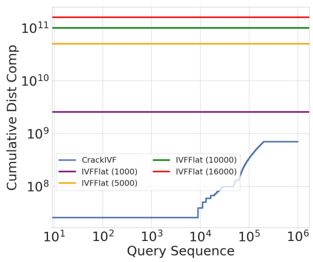


(a) SIFT1M (b) SIFT10M (c) Last.fm (d) Glove50 (e) Glove100 (f) Deep10M

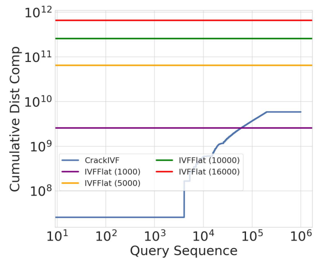
Figure 4: Cumulative time plots across datasets.

distance computations are spread across time and space and can potentially be handled on cheaper hardware (e.g., CPUs).

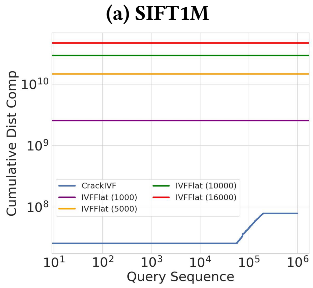
Figure 5 shows the cumulative distance computations during index build, excluding search operations. CrackIVF pays this cost gradually, and after the index converges, the final distance computations can be orders of magnitude lower than larger IVF indexes (Figure 5a, 5f). Despite having comparable or better performance (Figure 3a, 3k). Convergence occurs when the arrival of new crack candidates stops and refines do not trigger on new regions, example is shown in ablation. On Glove, distance computations increase faster due to *REFINE* regions containing more points and more cracks. The median *REFINE* in Glove100 covers 4.78% of the dataset with 512 cracks, while the median *REFINE* in SIFT1M is only 0.63% of the total points across 64 cracks. Nonetheless, the total cost remains amortized over time and is still almost an order of magnitude lower than that of the largest IVF index, while achieving similar final QPS-Recall performance. Finally, *CRACK* is dominated by data movement, while *REFINE* is mostly compute (Table 2). Even though *CRACK* is infrequent (< 0.03% of trace Figure 3), its overhead is not fully captured by distance computations alone. This is why we use fixed hardware and cumulative time plots, to compare the total time across all operations, including search, in Figure 4.



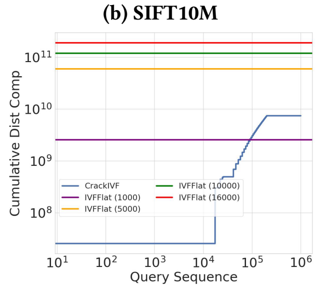
(a) SIFT1M



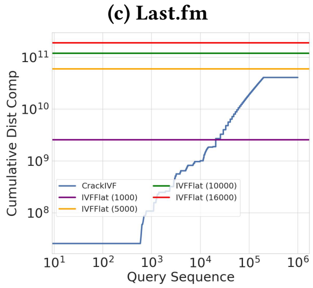
(b) SIFT10M



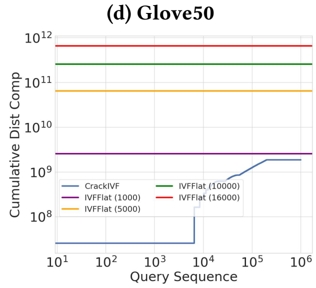
(c) Last.fm



(d) Glove50

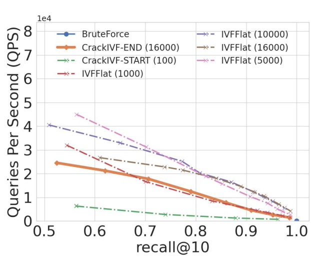


(e) Glove100

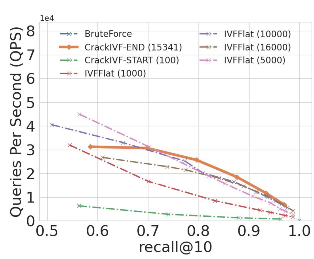


(f) Deep10M

Figure 5: Cumulative distance computations for index build.



(a) WHERE=OFF



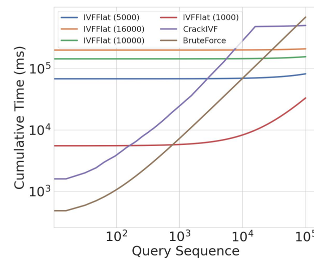
(b) WHERE=ON

Figure 6: Effect of removing the "where" control mechanism

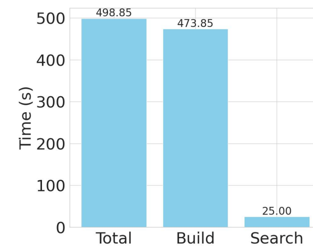
5.5 Control Mechanisms Ablation Study

This section is an ablation study exploring the control mechanisms "where" and "when" to apply *CRACK* and *REFINE*.

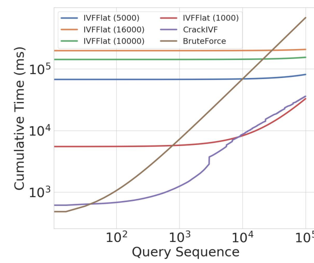
Turning off the control mechanism for "where": To illustrate the importance and accuracy of the heuristic rules for where to *CRACK* and *REFINE*, we switch them OFF and see the effect (Figure 6). Specifically, when "WHERE=OFF", and no heuristic rule is used to decide where a crack or refine should happen, the index defaults to trying to apply these operations whenever there



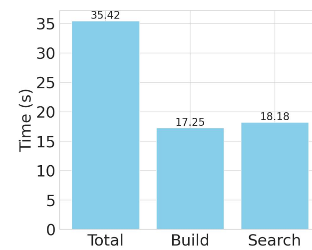
(a) Cumulative Time (OFF)



(b) Time Breakdown (OFF)



(c) Cumulative Time (ON)



(d) Time Breakdown (ON)

Figure 7: Effect of removing the "when" control mechanism

is enough budget for them, since the "when" mechanism is still ON. This seemingly random addition of new cracks leads to a final performance that is only slightly better than the initial state of CrackIVF, before any query has been received (Figure 6a). 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 6b). In both examples, we test an equal number of queries 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 $\alpha = 0.5$, 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 7a and Figure 7b. 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 7c and Figure 7d). These runs are for the same 100k query trace. The parameter controlling the "when" mechanism is α. The effect of varying α is shown in Figures 8a and 8b. If build is 25% of the total time then it is too restrictive. On the other hand, if 75% of the total time can go to build operations, a very large number of refines keep triggering. Figure 8b illustrates the importance of monitoring the new buffered cracks rate and the refine regions in order to converge the index. After ≈50k queries, no new cracks occur, preventing further index growth. Refines also repeatedly trigger in the same regions, as shown by the striated

Table 4: **Parameters defining the Decision Rule Boundaries.**

Parameter	Restrict "good candidates"	Tested Values
min_pts	↑	1, **2**, 32
pts_crack_thr	↑	16, **64**, 128
cv_max	↑	1, **2**, 8
size_prctl	↓	1, **10**, 25

pattern, offering no additional benefit, since the median response time does not improve after the refines. This indicates the index has converged. We monitor when this state is achieved and then converge the index, avoiding further build operations.

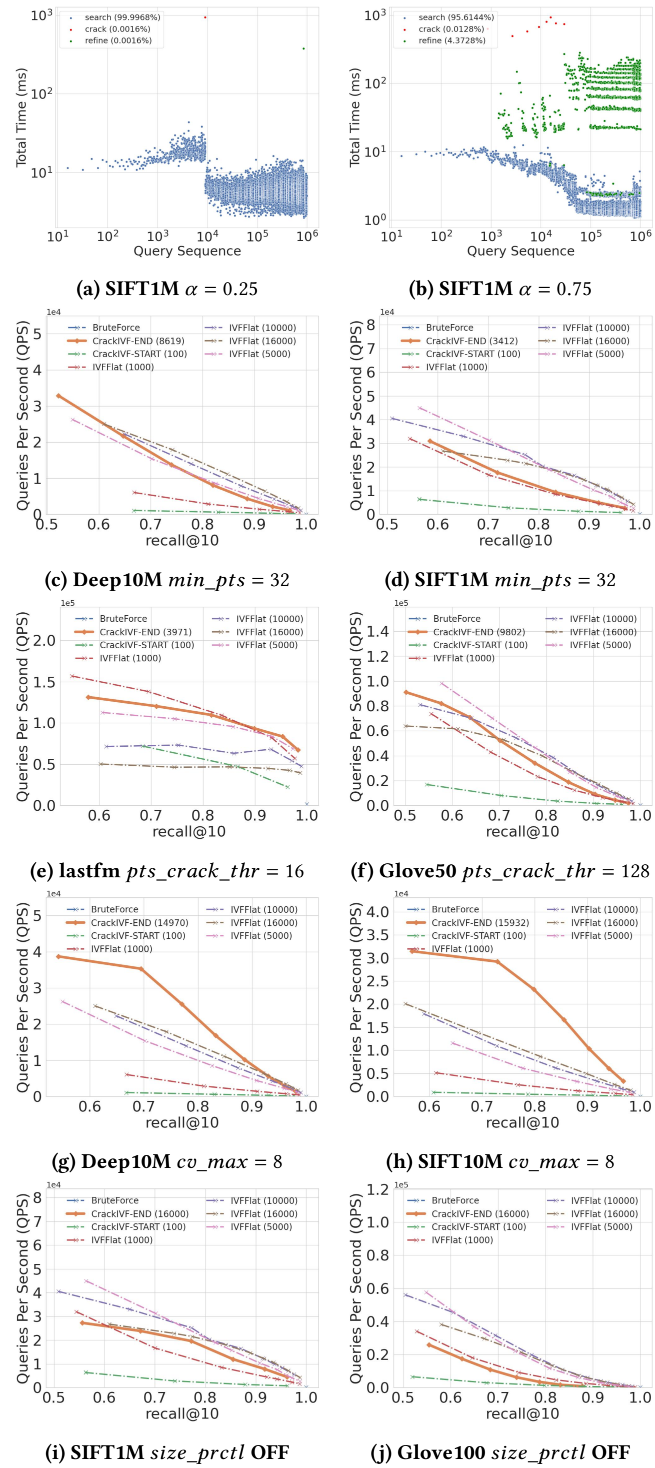
Ablation on decision boundary parameters: Each parameter defines a decision boundary (Section 4.2). Changing it in one direction, makes the model more restrictive, classifying fewer candidates as good, while in the opposite direction it is more permissive. Table 4 indicates the restrictive direction and tested values (default in **bold**). We did multiple runs for each dataset and parameter combination. For each parameter we pick the most interesting results, showcased in Figure 8. We only vary one parameter at a time, the others are fixed to their default value. We found the model to be robust to changing a single parameter. Out of the 48 edge-case parameter-dataset combinations, 15 had a noticeable effect on the final index. We attribute the robustness to each heuristic involving two decision rules (Section 4.2), thus two parameters. When one is changed, the other still provides a reasonable boundary. Setting both parameters to restrictive extremes prevents any cracking, while setting both to relaxed extremes is equivalent to `WHERE=OFF` (Figure 6a), where every query is classified as good and the operations occur when the cost budget allows.

For *min_pts*, we have the most consistent effect across all our observations. When we set it to a very restrictive value (32), the number of good cracks drops significantly, and thus the index converges to a smaller size, leading to lower performance (Figure 8d). This negatively affected all 6 datasets.

The *pts_crack_thr*, mostly affects the highest skew datasets. For example, on Last.fm performance worsens when the parameter is set to a permissive value, as this leads to over-partitioning of the region, as more cracks are allowed in a region (Figure 8e). But, the above value did not affect any other dataset. On the other hand, restricting it negatively affected 2 out of 6 datasets (Figure 8f). This was the same effect as that of *min_pts*. When the rule is too restrictive, not enough cracks pass it so index can not fully grow.

Restricting *cv_max* can have a positive effect, e.g. on SIFT10M (Figure 8h) and on DEEP10M (Figure 8g). This leads us to believe that in some datasets allowing a higher local imbalance, could have beneficial overfitting effects to the query distribution. The optimal value for *cv_max* is not consistent across datasets, two other datasets had worse results at a high value, and our default values provide the most robust results.

All the values tested for *size_prctl* had a negligible effect. Thus, to show the benefit of including the decision rule which *size_prctl* controls, we switch it off entirely in Figures 8i and 8j. Disabling it greatly affects final index. While the index grows, and it does not converge to the optimal state, as fewer refines occur.



(a) **SIFT1M** $\alpha = 0.25$ (b) **SIFT1M** $\alpha = 0.75$

(c) **Deep10M** *min_pts* = 32 (d) **SIFT1M** *min_pts* = 32

(e) **lastfm** *pts_crack_thr* = 16 (f) **Glove50** *pts_crack_thr* = 128

(g) **Deep10M** *cv_max* = 8 (h) **SIFT10M** *cv_max* = 8

(i) **SIFT1M** *size_prctl* **OFF** (j) **Glove100** *size_prctl* **OFF**

Figure 8: Ablation study of parameters.

Overall, we clearly see dataset-dependent effects which can change the optimal value of each parameter per dataset. This leaves room for further tuning and "learned" decision boundaries instead of setting them manually. Tuning of parameters exists in any indexing technique, the more you know about your dataset the better you can tune the index for it. We did not pursue tuning in this work, as the default values empirically provide very robust performance.

From our experiments, we make the following suggestion: BruteForce 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 minimal initialization cost, it is more complex and has a slight memory and runtime overhead compared to BruteForce due to the setup time and storage of internal object structures and keeping track of the first few partition representatives. At scale, in embedding lakes even a small overhead can accumulate. We recommend using CrackIVF once a dataset exceeds 100 queries and is likely to receive more.

6 RELATED WORK

Vector Databases: Vector Data Management Systems (VDBMS) [13, 15, 22, 28, 43, 68, 84, 88, 90, 94] support efficient ANN search, and have gained significant popularity. This coincides with the increase in use of Retrieval-Augmented Generation (RAG) [4, 46], to extend the capabilities of LLMs beyond what they have been trained on. VDBMS applications also include semantic search [57, 71], recommendation systems [47] and product search [80]. Many relational engines are incorporating vector search [18, 67]. Many of these systems support extensions to dense vector ANN search, allowing for specific query types such as filtered search [27], sparse vector search [26], and hybrid search [81]. A recent survey covering vector database management systems can be found in [64].

Vector Search Indexes: Approximate Nearest Neighbor (ANN) search has been well-studied. The two dominant categories of ANN indexes are partition-based and graph-based indices. Inverted file index (IVF) is a partition-based index and includes many different implementations. ScaNN [29] and SOAR [78] identify key mathematical properties of ANNS and use them to improve index efficiency. SPANN [16] uses disk-based storage to search over billion-scale datasets and supports updates [91]. FAISS-IVF [24] provides an in-memory implementation IVF which can utilize GPUs [40]. In this work we show how IVF indexes can be extended with cracking, and our approach is directly applicable to the above. We focus on IVF indexes as they offer the best trade-off between index build time, search performance, and memory overhead, which is critical to minimize overhead in large scale deployments on embedding data lakes, as covered in Section 3. The cracking paradigm can be generalized to other ANN indexes as well. With AV-tree[44] the authors show how to extend a tree-based index with cracking to supports k-NN queries targeting ENN search. Hash-based methods [2, 21, 36, 37, 85] and tree-based methods [50, 61, 83] also broadly fall in the partition-based category. Each index has a unique construction method and data structures. Adopting cracking ideas for them would require solving unique challenges. Finally, graph-based methods [23, 31, 51, 52, 82, 83] 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 ill-suited when the goal is minimizing the index construction cost and memory overhead. Cracking is applicable here and could push search performance even further. For example, future research can show how to dynamically change the connections in the graphs based on the arriving query distribution.

Adaptive indexes for specialized data types: Adaptive indexing techniques, such as Dumpy [86, 87] and ADS [95] are effective in data series similarity search. They are specifically designed for time-series data and include methods like Symbolic Aggregate Approximation (SAX) [48] and indexable SAX (iSAX) [75]. 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. Specialized, adaptive and incrementally constructed indexes are relevant candidates for embedding lake deployments, particularly for the subset of the datasets with inherent sequential dependencies such as energy, weather, financial data, audio, and video [42, 55, 69, 74].

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 [7] handles data or query distribution shifts keeping the total number of partitions fixed. LIRE [91] performs localized split and merge operations when partitions change after insertions/deletions. Ada-IVF [58] monitors partition size changes from updates and the partition access distribution from the queries. It combines this information to prioritize the maintenance operations. Finally, Quake [60] is an adaptive index for dynamic workloads with changing access patterns and heavy updates. They use a cost model to decide on maintenance operations, and dynamically set the nprobe parameter. Similar to us, these approaches demonstrate the effectiveness of localized operations for IVF indexes and the advantages of adapting to query distribution. However, they cannot be applied directly for our setting of incremental index construction to minimize cumulative time as the number of queries increases. These works trigger maintenance operations from incoming updates, or focus on distribution shits. Our focus is on the static setting, and we follow the database cracking paradigm: "index maintenance should be a byproduct of query processing, not of updates" [35]. Thus, CrackIVF operations are triggered solely from the queries.

7 CONCLUSION

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.

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] 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).

[2] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1225–1233.

[3] 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

[4] 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.

[5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374. https://doi.org/10.1016/j.is.2019.02.006

[6] 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.

[7] 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.

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

[9] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517. https://doi.org/10.1145/361002.361007

[10] 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)*.

[11] 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.

[12] 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.

[13] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6, Article 246 (Dec. 2024), 27 pages. https://doi.org/10.1145/3698822

[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] Yangshen Deng, Zhengxin You, Long Xiang, Qilong Li, Peiqi Yuan, Zhaoyang Hong, Yitao Zheng, Wanting Li, Runzhong Li, Haotian Liu, Kyriakos Mouratidis, Man Lung Yiu, Huan Li, Qiaomu Shen, Rui Mao, and Bo Tang. 2025. AlayaDB: The Data Foundation for Efficient and Effective Long-context LLM Inference. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 364–377. https://doi.org/10.1145/3722212.3724428

[23] 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.

[24] 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).

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

[26] 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.

[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* (Boston, Massachusetts) *(SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

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

[34] 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.

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

[36] 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.

[37] 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.

[38] R. Jain. 1991. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley New York.

[39] 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.

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

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

[42] 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.

[43] Leonardo Kuffo, Elena Krippner, and Peter Boncz. 2025. PDX: A Data Layout for Vector Similarity Search. *Proc. ACM Manag. Data* 3, 3, Article 196 (June 2025), 26 pages. https://doi.org/10.1145/3725333

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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).

[50] Ting Liu, Andrew W. Moore, Alexander Gray, and Ke Yang. 2004. An investigation of practical approximate nearest neighbor algorithms. In *Proceedings of the 18th International Conference on Neural Information Processing Systems* (Vancouver, British Columbia, Canada) *(NIPS'04)*. MIT Press, Cambridge, MA, USA, 825–832.

[51] 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.

[52] 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.

[53] 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.

[54] Renée J. Miller. 2018. Open data integration. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 2130–2139. https://doi.org/10.14778/3229863.3240491

[55] 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)*.

[56] 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.

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

[58] 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).

[59] 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.

[60] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. 2025. Quake: Adaptive Indexing for Vector Search. *arXiv preprint arXiv:2506.03437* (2025).

[61] 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.

[62] 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.

[63] OpenAI. 2025. OpenAI Embeddings. Online. https://platform.openai.com/docs/guides/embeddings Accessed: 2025-07-01.

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

[65] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: QUery-Aware Spatial Incremental Index. https://doi.org/10.5441/002/edbt.2018.29

[66] 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.

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

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

[69] 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.

[70] 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.

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

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

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

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

[75] 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.

[76] 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.

[77] 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.

[78] 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).

[79] Nan Tang, Chenyu Yang, Zhengxuan Zhang, Yuyu Luo, Ju Fan, Lei Cao, Sam Madden, and Alon Halevy. 2024. Symphony: Towards trustworthy question answering and verification using RAG over multimodal data lakes. *IEEE Data Eng. Bull* 48, 4 (2024), 135–146.

[80] 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.

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

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

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

[89] 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*.

[90] Qian Xu, Juan Yang, Feng Zhang, Junda Pan, Kang Chen, Youren Shen, Amelie Chi Zhou, and Xiaoyong Du. 2025. Tribase: A Vector Data Query Engine for Reliable and Lossless Pruning Compression using Triangle Inequalities. *Proc. ACM Manag. Data* 3, 1, Article 82 (Feb. 2025), 28 pages. https://doi.org/10.1145/3709743

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.