FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search

Aditi Singh
t-adisin@microsoft.com
Microsoft Research India

Suhas Jayaram Subramanya*
suhasj@cs.cmu.edu
Carnegie Mellon University

Ravishankar Krishnaswamy†
Harsha Vardhan Simhadri†
{rakri,harshasi}@microsoft.com
Microsoft Research India

Abstract

Approximate nearest neighbor search (ANNS) is a fundamental building block in information retrieval with *graph-based indices* being the current state-of-the-art [7] and widely used in the industry. Recent advances [51] in graph-based indices have made it possible to index and search billion-point datasets with high recall and millisecond-level latency on a single commodity machine with an SSD.

However, existing graph algorithms for ANNS support only *static* indices that cannot reflect real-time changes to the corpus required by many key real-world scenarios (e.g. index of sentences in documents, email or a news index). To overcome this drawback, the current industry practice for manifesting updates into such indices is to *periodically re-build these indices*, which can be prohibitively expensive.

In this paper, we present the first graph-based ANNS index that reflects corpus updates into the index in real-time *without compromising on search performance*. Using update rules for this index, we design FreshDiskANN, a system that can index over a billion points on a workstation with an SSD and limited memory, and support thousands of concurrent real-time inserts, deletes and searches per second each, while retaining > 95% 5-recall@5. This represents a 5-10x reduction in the cost of maintaining *freshness* in indices when compared to existing methods.

1 Introduction

In the Nearest Neighbor Search problem, we are given a dataset P of points along with a pairwise distance function. The goal is to design a data structure that, given a target k and a query point q, efficiently retrieves the k closest neighbors for q in the dataset P according to the given distance function. This fundamental problem is well studied in the research community [6, 9, 11, 16, 32, 35, 38, 43, 59] and is a critical component for diverse applications in computer vision [57], data mining [19], information retrieval [44], classification [26], and recommendation systems [21], to name a few. As advances in deep learning have made embedding-based approaches the state-of-the-art in these applications, there has been renewed interest in the problem at scale. Several open-source inverted-index based search engines now support NNS [49, 50, 55], and new search engines based on NNS are being developed [45, 56]. In newer applications of this problem, the dataset to be indexed and the queries are the output of a deep learning model – objects such as sentences or images are mapped so that semantically similar objects are mapped to closer points [10, 23]. These points reside in a space of dimension d (typically 100-1000), and the distance function is the Euclidean distance (ℓ_2) or cosine similarity (which is identical to ℓ_2 when the data is normalized).

Since it is impossible to retrieve the exact nearest neighbors without a cost linear in the size of the dataset in the general case (see [32, 59]) due to a phenomenon known as the *curse of dimensionality* [20], one aims to find the *approximate nearest neighbors* (ANN) where the goal is to retrieve k neighbors that are close to being optimal. The quality of an ANN algorithm is judged by the *trade-off* it provides between accuracy and the hardware resources such as compute, memory and I/O consumed for the search.

Even though this abstraction of ANN search is widely studied, it does not capture many important real-world scenarios where user interactions with a system creates and destroys data, and results in updates to P (especially in the literature on graph-based ANNS indices [58]). For example, consider an enterprise-search scenario where the system indexes sentences in documents generated by users across an enterprise. Changes to sentences in a document would correspond to a set of new points inserted and previous points deleted. Another scenario is an email server where arrival and deletion of emails correspond to insertion and deletion of points into an ANNS index. ANNS systems for such applications would need to host indices containing trillions of points with real-time updates that can reflect changes to the corpus in user searches, ideally in real-time.

Motivated by such scenarios, we are interested in solving the **fresh-ANNS** problem, where the goal is to support ANNS on a *continually changing set of points*. Formally, we define the fresh-ANNS problem thus: given a time varying dataset P (with state P_t at time t), the goal is to maintain a dynamic index that computes the approximate nearest neighbors for any query q issued at time t only on the active dataset P_t. Such a system must support three operations (a) *insert* a new point, (b) *delete* an existing point, and (c) *search* for the nearest neighbors given a query point. The overall quality of a fresh-ANNS system is measured by:

*Work done while at Microsoft.
†Authors listed in alphabetical order.

- The recall-latency tradeoff for search queries, and its robustness over time as the dataset P evolves.
- Throughput and latency of insertions and deletions.
- Overall hardware cost (CPU, RAM and SSD footprint) to build and maintain such an index.

We are interested in *quiescent consistency* [22, 31], where the results of search operations executed at any time t are consistent with some total ordering of all insert and delete operations completed before t.

We use the following notion of recall in this paper.[1]

Definition 1.1 (k-**recall@**k**).** For a query vector q over dataset P, suppose that (a) $G \subseteq P$ is the set of actual k nearest neighbors in P, and (b) $X \subseteq P$ is the output of a k-ANNS query to an index. Then the k-recall@k for the index for query q is $\frac{|X \cap G|}{k}$. Recall for a set of queries refers to the average recall over all queries.

Goal. Motivated by real-world scenarios, we seek to build the most cost-effective system for the fresh-ANNS problem which can maintain a billion-point index using commodity machines with 128GB RAM and a 2TB SSD[2] and support *thousands* of real-time inserts and deletes per second, and also *thousands* of searches per second with high accuracy of 95+% 5-recall@5. Indeed, the current state-of-art system for fresh-ANNS which can support comparable update and search performance on a billion-point dataset is based on the classical LSH algorithm [54], and requires a hundred machines of 32GB RAM (translating to around 25 machines of our stated configuration). *In this work, we seek to reduce this deployment cost down to a single machine per billion points.* To handle trillion-point indices (as in web-search scenarios), one can employ a simple distributed approach wherein thousand machines host a billion points each – queries are broadcast and results aggregates while updates are routed to the appropriate nodes.

1.1 Shortcoming of existing algorithms

Of all the algorithms for static-ANNS, the ones most easily capable of supporting streaming support are the ones based on simple hashing algorithms such as LSH (locality sensitive hashing). However, these algorithms suffer from either being too memory intensive, needing to store hundreds of hash functions in main memory, or become extremely slow for query processing when the index is stored on secondary storage. For example, the state-of-art system for streaming similarity search (or fresh-ANNS), PLSH [54], is a parallel and distributed LSH-based mechanism. While it offers comparable update throughput and search performance as our system, it ends up needing 25X more machines due to the high RAM consumption. A similar issue can be seen with PM-LSH, another state-of-art system based on LSH [62], where the memory footprint is a bit lower than PLSH (due to the system using fewer LSH tables), but the query latencies are an order of magnitude slower than our system and PLSH. Alternately, disk-based LSH indices such as SRS [53] can host a billion-point index on a single machine, but the query latencies are extremely slow with the system fetching around 15% of the total index (running into GBs per query) from the disk to provide good accuracy. Another recent algorithm HD-Index [5] can serve a billion-point index with just a few megabytes of RAM footprint, but it suffers from search latencies of a few seconds to get accuracy of around 30%. Moreover, the algorithm only handles insertions, and simply performs a variant of blacklisting for deletions, and hence would need periodic rebuilding. Finally, there are other classes of ANNS algorithms such as kd-Tree [14], Cover Trees [17] which support reasonably efficient update policies, but these algorithms work well only when the data dimensionality is moderately small (under 20); their performance drops when the data dimensionality is 100 or more which is typical for points generated by deep-learning models..

At the other end of the spectrum of ANNS indices are *graph-based indexing algorithms* [28, 33, 34, 43, 51, 52]. Several comparative studies [7, 25, 41, 58] of ANNS algorithms have concluded that they significantly out-perform other techniques in terms of search throughput on a range of real-world static datasets. These algorithms are also widely used in the industry at scale. However, *all known graph indices are static and do not support updates, especially delete requests* [18], possibly due to the fact that simple graph modification rules for insertions and deletions do not retain the same graph quality over a stream of insertions and deletions.

As a result, the current practice in industry is to periodically re-build such indices from scratch [18] to manifest recent changes to the underlying dataset. However, this is a very expensive operation. It would take about 1.5-2 hours on a dedicated high-end 48-core machine to build a good quality HNSW index [47] over 100M points. So we would need *three dedicated machines* for constantly rebuilding indices to maintain even *six-hourly freshness guarantee over a billion-point index*. This is apart from the cost of actually serving the indices, which would again be anywhere between one for DRAM-SSD hybrid indices [51] to four for in-memory indices [47] depending on the exact algorithm being deployed. This paper aims to serve and update an index over a billion points with **real-time freshness** using just one machine. This represents a significant cost advantage for web and enterprise-scale search platforms that need to serve indices spanning trillions of points.

1.2 Our Contributions

In this paper, we present the **FreshDiskANN** system to solve the fresh-ANNS problem for points in Euclidean space with real-time freshness, and with 5-10x fewer machines than

[1] An index that provides good k-recall@k can be used to satisfy other notions of recall such as finding all neighbors within a certain radius.

[2] Henceforth, when we refer to "a machine", we implicitly refer to this configuration unless otherwise specified.

the current state-of-the-art. As part of this, we make several technical contributions:

1. We demonstrate how simple graph update rules result in degradation of index quality over a stream of insertions and deletions for popular graph-based algorithms such as HNSW [43] and NSG [28].
2. We develop FreshVamana, the first graph-based index that supports insertions and deletions, and empirically demonstrate its stability over long streams of updates.
3. In order to enable scale, our system stores the bulk of the graph-index on an SSD, with only the most recent updates stored in memory. To support this, we design a novel two-pass `StreamingMerge` algorithm which makes merges the in-memory index with the SSD-index in a very write-efficient manner (crucial since burdening the SSD would lead to worse search performance as well). Notably, the *time and space complexity of the merge procedure is proportional to the change set*, thereby making it possible to update large billion-point indices on a machine with limited RAM using an order of magnitude less compute and memory than re-building the large index from scratch.
4. Using these ideas, we design the FreshDiskANN system to consist of a long-term *SSD-resident index* over the majority of the points, and a short-term *in-memory index* to aggregate recent updates. Periodically, unbeknownst to the end user, FreshDiskANN consolidates the short-term index into the long-term index using our `StreamingMerge` process in the background to bound the memory footprint of the short-term index, and hence the overall system.

We conduct rigorous week-long experiments of this system on an (almost) billion point subset of the popular SIFT1B [36] dataset on a 48 core machine and 3.2TB SSD. We monitor recall stability, end-user latency and throughput for updates and searches. Some highlights are:

- The system uses less than 128GB of DRAM at all times.
- The `StreamingMerge` can merge a 10% change to the index (5% inserts + 5% deletes) to a billion-scale index in ∼10% of the time than it takes to rebuild the index.
- FreshDiskANN can support a steady-state throughput of 1800 inserts and 1800 deletes per second while retaining freshness and without backlogging background merge. The system can also support short bursts of much higher change rate, up to even 40,000 inserts/second.
- The user latency of insertion and deletion is under 1ms, even when a background merge is underway.
- FreshDiskANN supports 1000 searches/sec with 95+% 5-recall@5 over the latest content of the index, with mean search latency well under 20ms.

2 Related Work

ANNS is a classical problem with a large body of research work. Recent surveys and benchmarks [7, 25, 41] provide a great overview and comparison of the state-of-the-art ANN algorithms. This section focuses on the algorithms relevant for vectors in high-dimensional space with Euclidean metrics, and examines their suitability for the fresh-ANNS setting we consider in this paper. Beyond ANNS for points in Euclidean spaces, there has been work for tailored inputs and other notions of similarity such as those for time series data, e.g., [1, 19, 40]. The work [25] provides a comprehensive study of such algorithms and their applicability.

Trees. Some of the early research on ANNS focused on low-dimensional points (say, $d \leq 20$). For such points, spatial partitioning ideas such as R^*-trees [13], kd-trees [14] and Cover Trees [16] work well, but these typically do not scale well for high-dimensional data owing to the curse of dimensionality. There have been some recent advances in maintaining several trees and combining them with new ideas to develop good algorithms such as FLANN [46] and Annoy [15]. However, they are built for static indices, and moreover, even here, the graph-based algorithms outperform them [7] on most datasets.

Hashing. In a breakthrough result, Indyk and Motwani [32] show that a class of algorithms, known as *locality sensitive hashing* can yield provably approximate solutions to the ANNS problem with a polynomially-sized index and sublinear query time. Subsequent to this work, there has been a plethora of different LSH-based algorithms [3, 32, 62], including those which depend on the data [4], use spectral methods [61], distributed LSH [54], etc. While the advantage of the simpler data-independent hashing methods are that updates are almost trivial, the indices are often entirely resident in DRAM and hence do not scale very well. Implementations which make use of auxiliary storage such as SRS [53] typically have several orders of magnitude slower query latencies compared to the graph-based algorithms. Other hashing-based methods [37, 42, 48] learn an optimal hash family by exploiting the neighborhood graph. Updates to an index would require a full re-computation of the family and hashes for every database point, making them impractical for fresh-ANNS.

Data quantization and **Inverted indices** based algorithms have seen success w.r.t the goal of scaling to large datasets with low memory footprint. These algorithms effectively reduce the dimensionality of the ANNS problem by *quantizing* vectors into a compressed representation so that they may be stored using smaller amount of DRAM. Some choices of quantizers [38] can support GPU-*accelerated* search on billion-scale datasets. Popular methods like IVFADC [35], OPQ [29], LOPQ [39], FAISS [38], IVFOADC+G+P [12] and IMI [8] exploit the data distribution to produce low

memory-footprint indices with reasonable search performance when querying for a large number of neighbors. While most methods[9, 29, 35, 38] minimize the vector *reconstruction error* $||x - x^{\dagger}||^2$, where x is a database vector and $x^{\dagger}$ is its reconstruction from the quantized representation, Anisotropic Vector Quantization [30] optimizes for error for maximum inner-product search. Some of these systems such as FAISS [38] support insert and delete operations on an existing index under reasonable conditions like stationary data distributions. However, due to the irreversible loss due to the compression/quantization, these methods fail to achieve even moderate values of 1-recall@1, sometimes plateauing at 50% recall. These methods offer good guarantees on weaker notions such as 1-recall@100, which is the likelihood that the true nearest neighbor for a query appears in a list of 100 candidates output by the algorithm. Hence they are not the methods of choice for high-recall high-throughput scenarios.

A recent work, ADBV [60], proposes a hybrid model for supporting **streaming inserts and deletes**. New points are inserted into an in-memory HNSW [43] index while the main on-disk index utilises a new PQ-based indexing algorithm called VGPQ. In order to mitigate the accuracy loss due to PQ, VGPQ search performs a large number of distance computations and incurs high search latencies. As distributed system over several powerful nodes, the model has low search throughput even when no inserts and deletes are going on. Hence, such a system cannot be used in high-throughput scenarios.

A recent work, ADBV [60], proposes a hybrid SQL-vector search model. New vectors are inserted into an in-memory HNSW index while the main on-disk index spanning upto a billion points is spread across multiple machines. The on-disk index is an extension of IVF-clustering [35] which is far less efficient for search compared to graph indices in terms of the number of distance comparisons and I/O. As a result, their aggregate search throughput on a billion point index spread across disks on 16 machines is lesser than the throughput of FreshDiskANN with one machine. Our work achieves this by designing an on-SSD updatable graph index which is far more efficient for search. Their insertion throughput on an index spread across 70 machines is also much lesser than that of FreshDiskANN on one machine.

3 Graph-based ANNS indices

In this section, we recap how most state-of-the-art graph-based indices work for static-ANNS and also highlight the issues they face with supporting deletions.

3.1 Notation

The primary data structure in graph indices is a directed graph with vertices corresponding to points in P, the dataset that is to be indexed, and edges between them. With slight notation overload, we denote the graph $G = (P, E)$ by letting P also denote the vertex set. Given a node p in this directed graph, we let $N_{\text{out}}(p)$ and $N_{\text{in}}(p)$ denote the set of out- and in-edges of p. We denote the number of points by $n = |P|$. Finally, we let x_p denote the database vector corresponding to p, and let $d(p, q) = ||\mathrm{x}_p - \mathrm{x}_q||$ denote the ℓ_2 distance between two points p and q. We now describe how graph-based ANNS indices are built and used for search.

Algorithm 1: GreedySearch(s, x_q, k, L)

Data: Graph G with start node s, query x_q, result size k, search list size $L \geq k$
Result: Result set $\mathcal{L}$ containing k-approx NNs, and a set $\mathcal{V}$ containing all the visited nodes

```
begin
    initialize sets L ← {s} and V ← ∅
    while L \ V ≠ ∅ do
        let p* ← arg min_{p∈L\V} ||x_p − x_q||
        update L ← L ∪ N_out(p*) and V ← V ∪ {p*}
        if |L| > L then
            update L to retain closest L points to x_q
    return [closest k points from V; V]
```

3.2 Navigability and Index Search

Roughly speaking, navigability of a directed graph is the property that ensures that the index can be queried for nearest neighbors using a *greedy* search algorithm. The greedy search algorithm traverses the graph starting at a designated *navigating* or start node $s \in P$. The search iterates by greedily walking from the current node u to a node $v \in N_{\text{out}}(u)$ that minimizes the distance to the query, and terminates when it reaches a *locally-optimal* node, say p^*, that has the property $d(p^*, q) \leq d(p, q) \, \forall p \in N_{\text{out}}(p^*)$. Greedy search cannot improve distance to the query point by navigating *out* of p^* and returns it as the candidate nearest neighbor for query q. Algorithm 1 describes a variant of this greedy search algorithm that returns k nearest neighbor candidates.

Index Build consists of constructing a navigable graph. The graph is typically built to achieve two contrasting objectives to minimize search complexity: (i) make the greedy search algorithm applied to each base point $p \in P$ in the vertex set converge to p in the fewest iterations (intuitively, this would ensure that Algorithm 1 converges to p when searching for a query x_q if p is the nearest-neighbor for x_q), and (ii) have a maximum out-degree of at most R for all $p \in P$, a parameter typically between $16 - 128$.

Algorithms like NN-*Descent* [24] use gradient descent techniques to determine G. Others start with a specific type of graph — an empty graph with no edges [43, 51] or an approximate k–NN graph [27, 28] — and iteratively refine G using the following two-step construction algorithm to improve navigability:

- **Candidate Generation** - For each base point x_p, run Algorithm 1 on G to obtain $\mathcal{V}, \mathcal{L}$. $\mathcal{V} \cup \mathcal{L}$ contains nodes *visited* and/or closest to p in G during the search in the current graph G, making them good candidates for adding to $N_{\text{out}}(p)$ and $N_{\text{in}}(p)$, thereby improving the navigability to p in the updated graph G.
- **Edge Pruning** – When the out-degree of a node p exceeds R, a pruning algorithm (like Algorithm 3 with α set to 1) filters out similar kinds of (or redundant) edges from the adjacency list to ensure $|N_{\text{out}}(p)| \leq R$. Intuitively, the procedure sorts the neighbors of p in increasing order of distance from p, and only retains an edge (p, p'') if there is no edge (p, p') which has been retained and p' is closer to p'' than p (i.e., if Algorithm 1 can reach p'' from p through p', then we can safely remove the edge (p, p'')).

3.3 Why are Deletions Hard?

While graph-indices offer state-of-the-art search performance, all known algorithms apply for the static-ANNS problem. In particular, deletions pose a big challenge for all these algorithms – e.g., see this discussion [18] on HNSW supporting delete requests by adding them to a blacklist and omitting from search results. Arguably, this is due to the lack of methods which modify the navigable graphs while retaining the original search quality. To further examine this phenomenon, we considered three popular static-ANNS algorithms, namely HNSW, NSG, and Vamana and tried the following natural update policies when faced with insertions and deletions.

Insertion Policy. For insertion of a new point p, we run the candidate generation algorithm as used by the respective algorithms and add the chosen in- and out-edges, and if necessary, whenever the degree of any vertex exceeds the budget, run the corresponding pruning procedure.

Delete Policy A. When a point p is deleted, we simply remove all in- and out-edges incident to p, without adding any newer edges to compensate for potential loss of navigability. Indeed, note that p might have been on several navigating paths to other points in the graph.

Delete Policy B. When a point p is deleted, we remove all in- and out-edges incident to p, and add edges in the local neighborhood of p as follows: for any pair of directed edges (p_{in}, p) and (p, p_{out}) in the graph, add the edge $(p_{\text{in}}, p_{\text{out}})$ in the updated graph. If the degree bound of any vertex is violated, we run the pruning procedure associated with the respective algorithm to control the degrees.

Figure 1 shows that both of these delete policies are not effective. In this experiment, we consider the SIFT1M dataset [2] comprising of a million points in 128 dimensions, and start with the static-ANNS index for each of the algorithms. We then compose an update stream by selecting 5% of the points at random and deleting them, followed by presenting them again as insertions. We then repeat this process over multiple

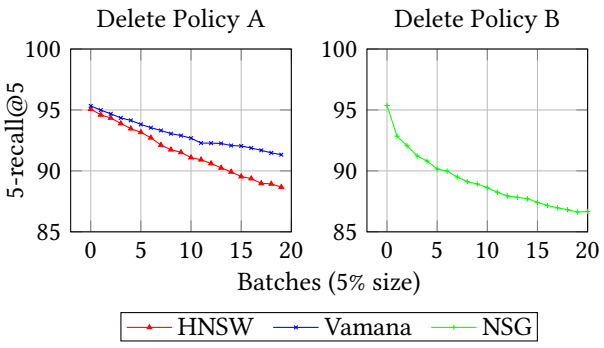

Figure 1. Search recall over 20 cycles of deleting and re-inserting 5% of SIFT1M dataset with statically built HNSW, Vamana, and NSG indices with L_s = 44, 20, 27, respectively.

cycles. A stable update policy should result in similar search performance after each cycle since the index is over the same dataset. However, all of the algorithms show a consistently deteriorating trend in search performance (the recall drops for a fixed candidate list size). The left plot in Figure 1 shows the trend for HNSW and Vamana indices with Delete Policy A, while the other considers the Delete Policy B for the NSG index. Other combinations show similar trends but we omit them due to lack of space.

4 The FreshVamana algorithm

Following the experiments in Section 3.3, we investigated the reason that the recall drops over multiple cycles of updates for deleting and re-inserting the same set of points. It turns out that the graph becomes sparse (lesser average degree) as we update it, and hence it becomes less navigable. We suspect that this is due to the very aggressive pruning policies of existing algorithms such as HNSW and NSG use to favor highly sparse graphs.

Fortunately, the sparsity-vs-navigability issue has recently been studied from a different perspective in [51], where the authors seek to build denser graphs to ensure the navigating paths converge much quicker. This in turn enables them to store such graphs on the SSD and retrieve the neighborhood information required by Algorithm 1 as required from the SSD without incurring large SSD latencies.

α-RNG Property. The crucial idea in the graphs constructed in [51] is a more relaxed pruning procedure, which removes an edge (p, p'') only if there is an edge (p, p') and p' must be significantly closer to p'' than p, i.e., $d(p', p'') < \frac{d(p,p'')}{\alpha}$ for some $\alpha > 1$. Generating such a graph using $\alpha > 1$ intuitively ensures that the distance to the query vector progressively decreases geometrically in α in Algorithm 1 since we remove edges only if there is a detour edge which makes significant progress towards the destination. Consequently, the graphs become denser as α increases.

We now present one of our crucial findings and contributions – graph index update rules for insertions and deletions that

Algorithm 2: Insert(x_p, s, L, α, R)

Data: Graph $G(P, E)$ with start node s, new point to be added with vector x_p, distance threshold $\alpha > 1$, out degree bound R, search list size L
Result: Graph $G'(P', E')$ where $P' = P \cup \{p\}$
begin
  initialize set of expanded nodes $\mathcal{V} \leftarrow \emptyset$
  initialize candidate list $\mathcal{L} \leftarrow \emptyset$
  $[\mathcal{L}, \mathcal{V}] \leftarrow$ GreedySearch$(s, p, 1, L)$
  set p's out-neighbors to be $N_{\text{out}}(p) \leftarrow$ RobustPrune$(p, \mathcal{V}, \alpha, R)$ (Algorithm 3)
  foreach $j \in N_{\text{out}}(p)$ **do**
    if $|N_{\text{out}}(j) \cup \{p\}| > R$ **then**
      $N_{\text{out}}(j) \leftarrow$ RobustPrune$(j, N_{\text{out}}(j) \cup \{p\}, \alpha, R)$
    else
      update $N_{\text{out}}(j) \leftarrow N_{\text{out}}(j) \cup \{p\}$

Algorithm 3: RobustPrune($p, \mathcal{V}, \alpha, R$)

Data: Graph G, point $p \in P$, candidate set $\mathcal{V}$, distance threshold $\alpha \geq 1$, degree bound R
Result: G is modified by setting at most R new out-neighbors for p
begin
  $\mathcal{V} \leftarrow (\mathcal{V} \cup N_{\text{out}}(p)) \setminus \{p\}$
  $N_{\text{out}}(p) \leftarrow \emptyset$
  while $\mathcal{V} \neq \emptyset$ **do**
    $p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$
    $N_{\text{out}}(p) \leftarrow N_{\text{out}}(p) \cup \{p^*\}$
    if $|N_{\text{out}}(p)| = R$ **then**
      break
    for $p' \in \mathcal{V}$ **do**
      if $\alpha \cdot d(p^*, p') \leq d(p, p')$ **then**
        remove p' from $\mathcal{V}$

exploit the α-RNG property to ensure continued navigability of the graph and retain stable recall over multiple modifications.

4.1 Insertion

A new point x_p is inserted into a FreshVamana index using Algorithm 2. Intuitively, it queries the current index for nearest neighbors of p to obtain the visited set $\mathcal{V}$, generates candidate out-neighbors for x_p using pruning procedure in Algorithm 3 on $\mathcal{V}$, and adds bi-directed edges between p and the pruned candidates. If out-degree of any vertex exceeds R, Algorithm 3 can be used to prune it to R.

We use lock-based concurrency control to guard access to $N_{\text{out}}(p)$ for a node p, allowing for high insertion throughput using multiple threads. Due to the fine granularity of locking and the short duration for which the locks are held, insertion throughput scales near-linearly with threads (see Appendix).

Algorithm 4: Delete(L_D, R, α)

Data: Graph $G(P, E)$ with $|P| = n$, set of points to be deleted L_D
Result: Graph on nodes P' where $P' = P \setminus L_D$
begin
  foreach $p \in P \setminus L_D$ *s.t.* $N_{\text{out}}(p) \cap L_D \neq \emptyset$ **do**
    $\mathcal{D} \leftarrow N_{\text{out}}(p) \cap L_D$
    $C \leftarrow N_{\text{out}}(p) \setminus \mathcal{D}$ //initialize candidate list
    foreach $v \in \mathcal{D}$ **do**
      $C \leftarrow C \cup N_{\text{out}}(v)$
    $C \leftarrow C \setminus \mathcal{D}$
    $N_{\text{out}}(p) \leftarrow$ RobustPrune(p, C, α, R)

4.2 Deletion

Our deletion algorithm Algorithm 4 is along the lines of Delete Policy B in Section 3.3, with the crucial feature being using the relaxed α-pruning algorithm to retain density of the modified graph. Specifically, if p is deleted, we add edges (p', p'') whenever (p', p) and (p, p'') are directed edges in the current graph. In this process, if $|N_{\text{out}}(p')|$ exceeds the maximum out-degree R, we prune it using Algorithm 3, preserving the $\alpha-$RNG property.

However, since this operation involves editing the neighborhood for all the in-neighbors of p, it could result be expensive to do *eagerly*, i.e., processing deletes as they arrive. FreshVamana employs a *lazy* deletion strategy – when a point p is deleted, we add p to a DeleteList without changing the graph. DeleteList contains all the points that have been deleted but are still present in the graph. At search time, a modified Algorithm 1 uses nodes in the DeleteList for navigation, but filters them out from the result set.

Delete Consolidation. After accumulating a non-trivial number of deletions (say 1-10% of the the index size), we batch-update the graph using Algorithm 4 to update the neighborhoods of points with out-edges to these deleted nodes. This operation is trivially parallelized using prefix sums to consolidate the vertex list, and a parallel map operation to locally update the graph around the deleted nodes.

4.3 Recall stability of FreshVamana

We now demonstrate how using our insert and delete algorithms (along with a choice of $\alpha > 1$) ensures that the resulting index is stable over a long stream of updates.

We start with a statically built Vamana index and subject it to multiple cycles of insertions and deletions using the FreshVamana update rules described in Section 4. In each cycle, we delete 5%, 10% and 50% of randomly chosen points from the existing index, and re-insert the same points. We then choose appropriate L_s (the candidate list size during

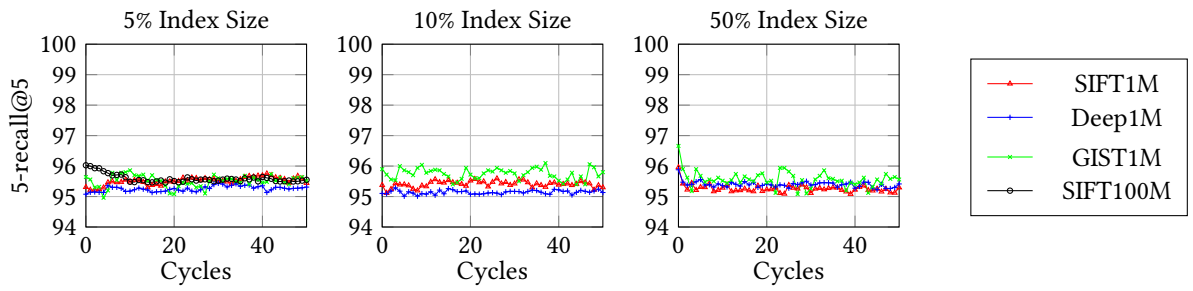

Figure 2. 5-recall@5 for FreshVamana indices for 50 cycles of deletion and re-insertion of 5%, 10%, and 50% of index size on the million-point and 5% of SIFT100M datasets. L_s is chosen to obtain 5-recall@5≈ 95% on Cycle 0 index.

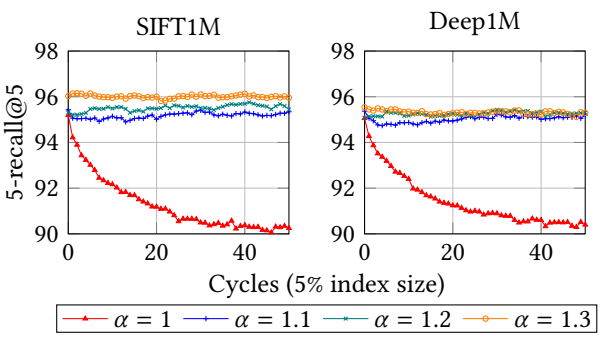

Figure 3. Recall trends for FreshVamana indices on SIFT1M and Deep1M over multiple cycles of inserting and deleting 5% of points using different values of α for building and updating the index. L_s is chosen to obtain $5-recall@5 \approx 95\%$ for Cycle 0 index.

search) for 95% 5-recall@5 and plot the search recall as the index is updated. Since both the index contents and L_s are the same after each cycle, a good set of update rules would keep the recall stable over these cycles. Figure 2 confirms that is indeed the case, for the million point datasets and the 100 million point SIFT100M dataset. In all these experiments, we use an identical set of parameters L, α, R for the static Vamana index we begin with as well as our FreshVamana updates. Note that in some of these plots, there is a small initial drop in recall; this is possibly due to the fact that the static Vamana indices which we are starting from are built by making two passes of refinement over the dataset and hence might have slightly better quality than the streaming FreshVamana algorithm.

Effect of α. Finally we study the effect of α on recall stability. In Figure 3, we run the FreshVamana update rules for a stream of deletions and insertions with different α values, and track how the recall changes as we perform our updates. Note that recall is stable for all indices except for the one with $\alpha = 1$, validating the importance of using $\alpha > 1$.

5 The FreshDiskANN system

While FreshVamana can support fast concurrent inserts, deletes and searches with an in-memory index, it will not scale to a billion-points per machine due to the large memory footprint of storing the graph and data in RAM. The main idea of overall system FreshDiskANN is to store a bulk of the graph-index on an SSD, and store only the recent changes in RAM.[3] To further reduce the memory footprint, we can simply store *compressed vector* representation (using an idea such as Product Quantization (PQ) [35]) of all the data vectors. In fact, these ideas of using α-RNG graphs and storing only compressed vectors formed the crux of the SSD-based DiskANN static-ANNS index [51].

While this will reduce the memory footprint of our index, and will also ensure reasonable search latencies, we cannot immediately run our insert and delete Algorithms 2 and 4 on to a SSD-resident FreshVamana index. Indeed, the insertion of a new point x_p has to update the neighborhoods of as many as R (the parameter controlling the degree bound) many points to add edges to p, which would trigger up to R random writes to the SSD. For typical indices, R would be as large as 64 or 128, requiring as many random SSD writes per insert. This would severely limit the insertion throughput and also reduce the search throughput as a high write load on the SSD also affects its read performance, which is critical to search latency. Similarly, each delete operation, if applied eagerly, would result in R_{in} writes, where R_{in} is the in-degree of the deleted point, which can be very large.

The FreshDiskANN system circumvents these issues and brings together the efficiency of a SSD-based system and the interactive latency of an in-memory system by *splitting* the index into two parts: (i) an in-memory FreshVamana component comprising of recent updates, and (ii) a larger SSD-resident index with longer term data.

[3] As FreshVamana graphs are constructed using the α-RNG property (Section 4), the number of steps that the greedy search algorithm takes to converge to a locally optima is much smaller than other graph algorithms. Hence the total search latency to fetch the graph neighborhoods from SSD is small. So the α-RNG property helps us with both ensuring recall stability as well as obtaining tolerable search latencies for SSD-based indices.

5.1 Components

The overall system maintains two types of indices: one *Long-Term Index* (aka LTI) and one or more instances of *Temporary Index* (a.k.a TempIndex), along with a DeleteList.

- LTI is an SSD-resident index that supports search requests. Its memory footprint is small, and consists only of about 25-32 bytes of compressed representations for each point. The associated graph index and full-precision data is stored on the SSD like [51]. *Insertions and deletions do not affect the* LTI *in real-time.*
- One or more TempIndex objects, which are instances of the FreshVamana index stored entirely in DRAM (both the data and the associated graph). By design, they contain points that have been recently inserted to P. As a result, their memory footprint is a small fraction of the entire index.
- DeleteList is the list of points that are present either in the LTI or the TempIndex, but have been requested for deletion by the user. This list is used to filter out the deleted points returned in the search results.

RO- and RW-TempIndex: To aid with crash recovery, FreshDiskANN uses two types of TempIndex. At all times, FreshDiskANN will maintain one mutable read-write TempIndex (called RW-TempIndex) which can accept insert requests. We periodically convert the RW-TempIndex into a *read-only in-memory index* called RO-TempIndex, and also snapshot it to persistent storage. We then create a new empty RW-TempIndex to ingest new points.

5.2 FreshDiskANN API

The following three operations are supported:

- Insert(x_p) to insert a new point to the index is routed to the sole instance of RW-TempIndex, which ingests the point using in Algorithm 2.
- Delete(p) request to delete an existing point p is added to the DeleteList.
- Search(x_q, K, L) to search for the K nearest candidates using a candidate list of size L is served by querying LTI, RW-TempIndex, and all instances of RO-TempIndex with parameters K and L, aggregating the results and removing deleted entries from DeleteList.

5.3 The `StreamingMerge` Procedure

Finally, to complete the system design, we now present details of the `StreamingMerge` procedure. Whenever the total memory footprint of the various RO-TempIndex exceeds a pre-specified threshold, the system invokes a background merge procedure serves to change the SSD-resident LTI to reflect the inserts from the various instances of the RO-TempIndex and also the deletes from the DeleteList. To this end, for notational convenience, let dataset P reflect the points in the LTI, and N denote points currently staged in the different RO-TempIndex instances, and D denote the points marked for deletion in DeleteList. Then the desired end-result of the `StreamingMerge` is an SSD-resident LTI over the dataset $(P \cup N) \setminus D$. Following the successful completion of the merge process, the system clears out the RO-TempIndex instances thereby keeping the total memory footprint under control. There are two important constraints that the procedure must follow:

- Have a memory footprint proportional to size of the changes $|D|$ and $|N|$, and not the size of overall index $|P|$. This is critical since the LTI can be much larger than the memory of the machine.
- Use SSD I/Os efficiently so that searches can still be served while a merge runs in the background, and so that the merge itself can complete fast.

At a high level, `StreamingMerge` first runs Algorithm 4 to process the deletes from D to obtain an intermediate-LTI index over the points $P \setminus D$. Then `StreamingMerge` runs Algorithm 2 to insert each of the points in N into the intermediate-LTI to obtain the resultant LTI. However, Algorithms 2 and 4 assume that both the LTI graph, as well as the full-precision vectors all the datapoints are stored in memory. The crucial challenges in `StreamingMerge` is to simulate these algorithm invocations in a memory and SSD-efficient manner. This is done in three phases outlined below.

1. `Delete Phase:` This phase works on the input LTI instance and produces an *intermediate*-LTI by running Algorithm 4 to process the deletions D. To do this in a memory-efficient manner, we load the points in LTI and their neighborhoods in the LTI *block-by-block* from the SSD, and execute Algorithm 4 for the nodes in the block using multiple threads, and write the modified block back to SSD on the intermediate-LTI. Furthermore, whenever Algorithm 4 or Algorithm 3 make any distance comparisons, *we use the compressed PQ vectors which are already stored on behalf of the* LTI *to calculate the approximate distances.* Note that this idea of replacing any exact distance computations with approximate distances using the compressed vectors will be used in the subsequent phases of the `StreamingMerge` also.

2. `Insert Phase:` This phase adds all the new points in N to the intermediate-LTI by trying to simulate Algorithm 2. As a first step, we run the GreedySearch$(s, p, 1, L)$ on the SSD-resident intermediate-LTI to get the set $\mathcal{V}$ of vertices visited on the search path. Since the graph is stored on the SSD, any requested neighborhood $N_{\text{out}}(p')$ by the search algorithm is fetched from the SSD. The α-RNG property ensures that the number of such neighborhood requests is small, and hence the overall latency per point is bounded. We then run the RobustPrune$(p, \mathcal{V}, \alpha, R)$ procedure to determine the candidate set of neighbors for p. However, unlike Algorithm 2, we do not immediately attempt to insert p into $N_{out}(p')$ for $p' \in N_{out}(p)$ (the backward edges) since this could result in an impractical number of random reads and writes to

the SSD. Instead, we *maintain an in-memory data-structure* $\Delta(p')$ and add p to that.

3. `Patch Phase:` After processing all the inserts, we patch the Δ data-structure into the output SSD-resident LTI index. For this, we fetch all points p in the intermediate-LTI block-by-block from the SSD, add the relevant out-edges for each node p from Δ, and check the new degree $|N_{\text{out}}(p) \cup \Delta(p)|$ exceeds R. If so, prune the neighborhood by setting $N_{out}(p) = \text{RobustPrune}(p, N_{out}(p) \cup \Delta(p), \cdot, \cdot)$. Within each block read from the SSD, this operation can be applied to each vertex in a data-parallel manner. Subsequently, the updated block is written back to SSD before loading a new block.

5.4 Complexity of `StreamingMerge`

I/O cost. The procedure does exactly two sequential passes over the SSD-resident data structure in the Delete and Patch Phases. Due to the α-RNG property of the intermediate-LTI, the insertion algorithm performs a small number of random 4KB reads per inserted point (about 100 disk reads, a little more than the candidate list size parameter, which we typically set to 75). Note that this number would be much larger without the α-RNG property due to the possibility of very long navigation paths.

Memory footprint: Throughout the `StreamingMerge` process, Δ data structure has size $O(|N|R)$ where R is the max-degree parameter of the index which is typically a small constant. For example, if $|N| = 30M$ and $R = 64$, this footprint will be ~7GB. In addition, for approximate distances, recall that we keep a copy of PQ coordinates for all points in the index ($\sim 32GB$ for a billion-point index).

Compute requirement: The complexity of the insert phase and the patch phase is essentially linear in the size of the new points N to insert, since the insert phase simply runs a search using Algorithm 1 for new point in N and updates the Δ data structure, and the patch phase adds the backward edges in a block-by-block manner.

The delete phase has a small fixed cost to scan $N_{\text{out}}(p)$ of each point $p \in P$ and check if there any deleted points and a larger variable cost, linear in the delete set size $|D|$ that we will bound by $O(|D|R^2)$ (in expectation over random deletes). We detail this calculation in Appendix D.

5.5 Recall Stability of `StreamingMerge`

While we have already demonstrated that our update algorithms Algorithms 2 and 4 ensure recall stability over long streams of updates in Section 4.3, the actual form in which these algorithms are implemented in our `StreamingMerge` procedure is different, especially with the use of approximate compressed vectors for distance computations. Indeed, as we process more cycles of the `StreamingMerge` procedure, we expect the initial graph to be replaced by a graph entirely built based on approximate distances. Hence, we expect a small drop in recall in the initial cycles, following which we expect the recall to stabilize.

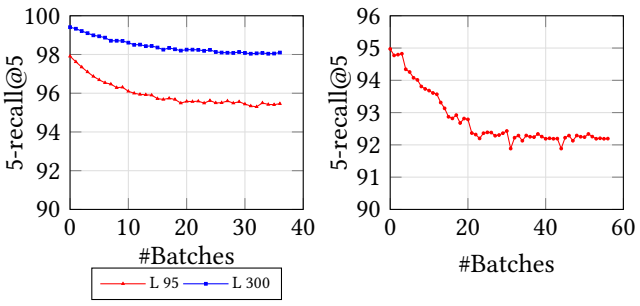

Figure 4. Recall evolution over multiple cycles of `StreamingMerge` in *steady-state* over (left) 80M point index with 10% deletes and inserts and (right) 800M point index with 30M insertions and deletions.

In the experiment in Figure 4, we start with a statically built SSD-index built on 80M points randomly sampled from the SIFT100M dataset. Then, in each cycle, we update the index to reflect 8M deletions and an equal number of insertions from the spare pool of 20M points using `StreamingMerge`. We run this experiment for a total of 40 cycles and trace recall for the index after each cycle in Figure 4. Note that the index stabilizes at a lower recall value compared to the static index it starts out with, due to the use of approximate distances in the `StreamingMerge` process. We observe recall stabilization after $\approx$ 20 cycles of deletion and insertion of 10% of the index size, at which point we expect most of the graph to be determined using approximate distances. Figure 4 (right) shows a similar plot for the 800M point subset of SIFT1B. We have thus empirically demonstrated that the FreshDiskANN index has stable recall over a stream of updates at steady-state.

5.6 Crash Recovery

To support crash recover, all index update operations are written into a redo-log. When a crash leads to the loss of the single RW-TempIndex instance and the DeleteList, they are rebuilt by replaying updates from the redo-log since the most recent snapshot. Since RO-TempIndex and LTI instances are read-only and periodically snapshotted to disk, they can be simply reloaded from disk.

The frequency at which RW-TempIndex is snapshotted to a RO-TempIndex depends on the intended recovery time. More frequent snapshots lead to small reconstruction times for RW-TempIndex but create many instances of RO-TempIndex all of which have to be searched for each query. While searching a few additional small in-memory indices is not the rate limiting step for answering the query (searching the large LTI is), creating too many could can lead to inefficient search. A typical set up for a billion-point index would hold up to 30M points in the TempIndex between merges to the LTI. Limiting each in-memory index to 5M points results in at

most 6 instances TempIndex which can each be searched in 0.77ms, compared to 0.89ms needed to search a single 30M size index, for $Ls = 100$. On the flip side, reconstructing the RW-TempIndex from the log using a 48 core machine takes just about 2.5 minutes if it has size 5M points as opposed to 16 minutes for a size of 30M points.

6 Evaluation

We now study the FreshDiskANN system on billion-scale datasets. We first describe the datasets and the machines used for all experiments reported in this paper. We defer presentation of recall-vs-latency curves for FreshVamana and FreshDiskANN at $k = 1, 10, 100$ to Appendix E.

6.1 Experimental Setup

Hardware. All experiments are run on one of two machines:

- (`mem-mc`) – a 64-*vcore* `E64d_v4` Azure virtual machine instance used to measure latencies and recall for in-memory indices and the FreshVamana update rules.
- (`ssd-mc`) – a bare-metal server with 2x Xeon 8160 CPUs (48 cores, 96 threads) and a 3.2TB Samsung PM1725a PCIe SSD to evaluate SSD-based indices and the overall FreshDiskANN system.

Datasets. We evaluate our algorithms and systems on the following widely-used public benchmark datasets.

- 1 million point image descriptor datasets SIFT1M[2], GIST1M[2], and DEEP1M[10] in 128, 960 and 98 dimensions respectively. They are all in `float32`. DEEP1M is generated by convolutional neural networks.
- 1 billion point SIFT1B[2] image descriptors in 128 dimensions. It is the largest publicly available dataset and is in `uint8` precision (total data size 128GB). We take a random 100M point subset of this dataset, represented in `float32` format and call it the SIFT100M dataset. We think that this smaller dataset captures many realistic medium-scale scenarios for ANNS.

6.2 Billion-Scale FreshDiskANN Evaluation

We now study the complete FreshDiskANN system in a realistic scenario – maintaining a large scale billion-scale index on the `ssd` machine and serving thousands of inserts, deletes and searches per second concurrently over multiple days. For this experiment, we use the SIFT1B dataset, but limit the size of our indices to around 800M points, so that we have a sufficiently big spare pool of 200M points for insertions at all times.

Parameters. We use $R = 64$, $L_c = 75$ and $\alpha = 1.2$ for all the system. Recall that R is the maximum degree of the graph, L_c is the list size used during the candidate generation phase of the algorithms (the parameter is used in Algorithm 2), and α is used in the pruning phase for ensuring the α-RNG property. We also use $B = 32$ bytes per data vector as the compression target in PQ (each data vector is compressed down to 32 bytes) for the SSD-based LTI indices. We also set a limit M of 30M points on the total size of the TempIndex so that the memory footprint of the TempIndex is bounded by around 13GB (128 bytes per point for the vector data, 256 bytes per point for the neighborhood information with $R = 64$, and some locks and other auxiliary data structures accounting for another 100 bytes per point). Finally, we use a maximum of $T = 40$ threads for the `StreamingMerge` process which runs in the background.

Memory Footprint of FreshDiskANN Deployment. As mentioned above, the memory footprint of the TempIndex is around 13 GB for 30M points, and our index will at any time store at most TempIndex instances totaling 60M points, contributing a total of ~26GB. The memory footprint index of the LTI for 800M points is essentially only the space needed to store the compressed vectors, which is around 24 GB. The space requirement for the background `StreamingMerge` process is again at most 50 GB (to store the compressed vectors of the 800M points of the LTI index and around $2 \cdot R \cdot 4$ bytes per inserted point for forward and backward edges in the Δ data structure), giving us a peak memory footprint of around 100GB. Since our index operated with a steady-state size of 800M points, this will roughly correspond to around 125GB for a billion-point index.

Our experiment can be divided into two phases: in the first phase, starting with a statically built index on a random 100M subset of SIFT1B, we define our *update stream to comprise only of inserts* until the total number of points in the index reaches around 800M points. We call this the ramp-up phase. We then transition into what we call a steady-state phase, where we update the index by deleting and inserting points at the same rate. We delete existing points and insert points from the spare pool of 200M points from the SIFT1B dataset. We then continue this for several days and observe the behaviour of the system in terms of latencies and recall.

How fast can we feed inserts into the system in these phases, i.e., how many threads can we use to concurrently insert into the FreshDiskANN system? If we use too many threads for insertion, the TempIndex will reach the limit M of 30M points before the `StreamingMerge` process has completed. This would result in a backlog of inserts not consolidate to LTI on SSD. With the benefit of some prior experiments (of how long each cycle of the `StreamingMerge` takes), we arrive at the number of threads which concurrently feed inserts into the FreshDiskANN system in each of the phases and describe them below.

Stage 1: Ramp Up. In the first stage of the experiment, we use the FreshDiskANN system to start with an index of 100M points randomly chosen from the SIFT1B dataset, and constantly feed inserts. 3 threads were used for concurrently inserting points from the spare pool of points from SIFT1B, and 10 threads for issuing concurrent search requests from the query set (with search parameters set to provide > 92%

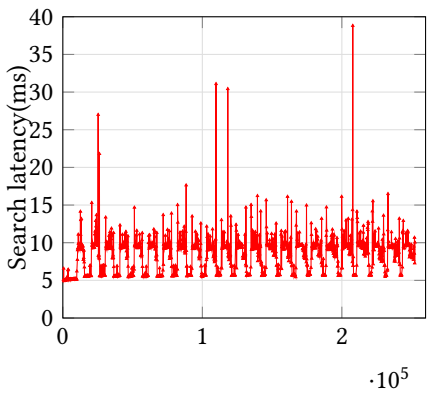

Figure 5. Search latencies for $Ls = 100$ (always > 92% 5-recall@5) over the course of ramping up an index to size 800M. Each point is mean latency over a 10000-query batch.

5-recall@5 at all times). We chose 3 threads for inserts so that the merge process does not get backlogged, i.e., in the time taken by `StreamingMerge` to merge the previous batch of 30M inserts to LTI, the TempIndex does not accumulate more than 30M points. The insertions continued until the index grew to a size of 800M points, which took around 3 days. User-perceived mean search latency over the course of the ramp-up fluctuates mostly between 5ms, when no merge is happening, and 15ms when `StreamingMerge` is running in the background and is presented in Figure 5.

Stage 2: Steady State. In the second stage of the experiment, we maintain an index size of around 800M while supporting a large number of equal inserts and deletes. Externally, 2 threads insert points into the index, 1 thread issues deletes, and 10 threads concurrently search it. Since the deletes happen near instantly, we added a sleep timer between the delete requests to ensure that the rate of deletions is similar to that of insertions. Note that we reduced the number of insert threads from 3 to 2 to slow down the insertion rate to accommodate the longer merge times compared to the ramp-up experiment – the `StreamingMerge` process now processes 30M deletes in addition to 30M inserts. We present user-perceived latencies for search and insertions in Figure 6.

Variations in Search Latency During `StreamingMerge`. The middle plot in Figure 6 shows that the user-perceived search latencies varies across based on the phase of the `StreamingMerge` process in progress. Since the `Insert` phase generates a significant number of random reads to the LTI index which interfere with the random read requests issued by the search threads, it results in slightly higher latencies. On the other hand, while the typical latencies are smaller during the `Delete` and `Patch` phases of `StreamingMerge`, the latencies occasionally spike as high as 40ms, which we think is likely due to head-of-line blocking by the large sequential read and write operations that copy the LTI index to and from the main memory.

Update Throughput of System. While FreshDiskANN provides latencies of about 1ms for insert (Figure 6) and 0.1μ for delete (since they are simply added to a DeleteList), in practice they need to be throttled so that the in-memory TempIndex do not grow too large before the ongoing background merge completes. As a result, the speed of the merge operation dictates the update rates the system can sustain over long periods of time. The threads allocation described above helps us control the rate of external insert and delete operations to what the `StreamingMerge` procedure can complete before the TempIndex grows to 30M points.

To better understand the thread allocation, we record the time taken for the `StreamingMerge` process to merge 30M inserts into an index of size roughly 800M using $T = 40$ threads. This takes around 8400s per cycle. To prevent the TempIndex from growing too much while the merge procedure is running, we throttle the inserts to around 3500 inserts per second, so that the TempIndex accumulates under 30M newly inserted points in one merge cycle. Since the insertion latencies into in-memory FreshVamana indices is around 1ms (Figure 6), we allocated a total of 3 threads concurrently feeding into the system. This ensured that the system never backlogged throughout the ramp-up experiment.

In the steady-state experiment where the index maintains a constant size of about 800M points and is updated in cycles of equal sized insertions and deletions of 30M points, the `StreamingMerge` procedure takes about 16277 seconds as it has to process deletes in addition to the inserts. Hence, in order to ensure that the system does not get backlogged, we throttled the insertion throughput to around 1800 inserts per second (and similarly for deletes). We achieved this by using two threads for the insertions, and one thread (with a sleep timer) for deletes to match the insertion throughput.

Trade-off of Varying Number of Merge Threads T. If we increase the merge threads T, the merges happen faster, which means we can ingest insertions and deletions into the system at a faster throughput (without the TempIndex size growing too large). On the other hand, if T is large, the SSD-bandwidth used by the `StreamingMerge` process increases and this adversely affects the search throughput. We examine the merge times with varying threads in Figure 7 (left) and the search latencies when different numbers of threads are performing background merge in Figure 8.

I/O Cost of Search. Running search with candidate list size $L_s = 100$ gives us the desired steady-state recall in these experiments. For this L_s value, the average I/O complexity of searches ends up being a *mere* 120 *random* 4*KB reads per query*, and the total number of distance comparisons made is around 8000, *a really tiny fraction of the cost of doing brute*

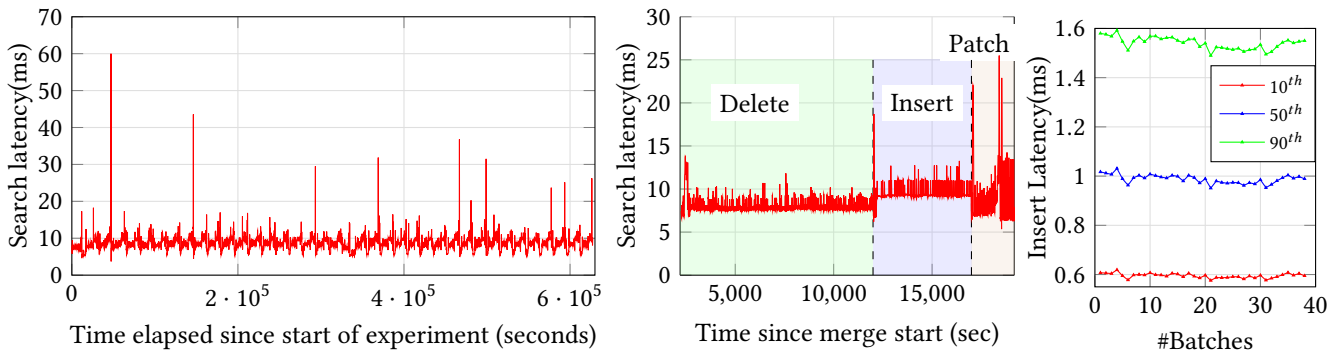

Figure 6. Mean latency[4]measurements for the week-long *steady-state* experiment with an 800M FreshDiskANN index processing concurrent inserts, deletes, and periodic background merge. (left) Search latency with $Ls = 100$ over the entire experiment; (middle) Search latency during one `StreamingMerge` run, zoomed in from the left plot; (right) 10^{th}, 50^{th} and 90^{th} percentile insert latency over the entire experiment.

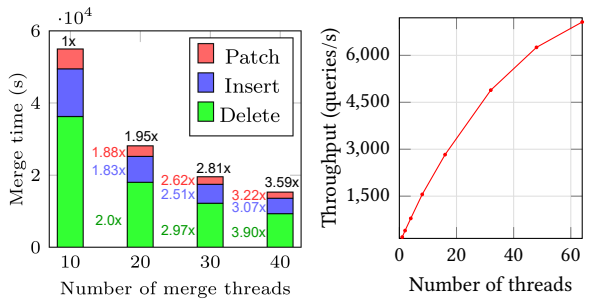

Figure 7. (left) `StreamingMerge` runtime with different number of threads to merge 30M inserts and 30M deletes into a 800M SIFT index, and (right) Trend of search throughput with increasing search threads.

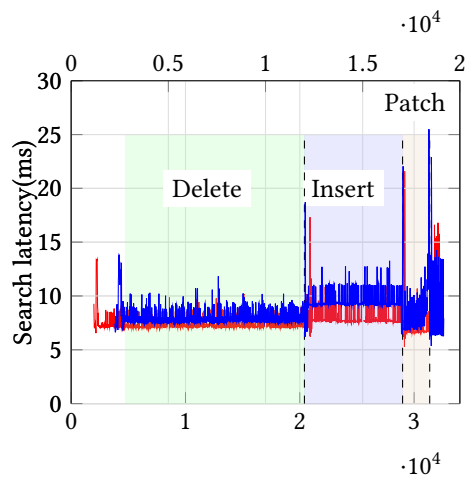

Figure 8. Trend of search latencies for 92% search recall, zoomed in over one cycle of merging 30M inserts and deletes into a 800M index, using 20 threads (red) and 40 threads (blue) for merge (time-axes are normalized to align the phases).

force. In contrast, systems like SRS [53] end up scanning $\approx 15\%$ of similar-sized datasets for achieving moderate recall.

I/O Cost of Updates. Inserts and deletes involve reading and writing the entire LTI ($\approx$ 320GB), twice over. Since our system amortizes this cost over $30M$ inserts and deletes, the SSD write cost per update operation is around $10KB$, which is very small for a high dimensional problem that requires data structure and algorithm with random access patterns.

Scaling of Search Throughput. When the index is not processing any inserts, deletes or merges, search throughput scales almost linearly with the number of threads issuing search queries (see Figure 7) (right), and with lesser latency than in Figure 6. With 64 threads, the system can support a throughput of $\sim$ 6500 queries/sec with a mean and 99% latency of under 10 and 12ms respectively.

The Cost of `StreamingMerge`. The `StreamingMerge` procedure with 40 threads takes around 16000 seconds to merge 30M inserts and deletes into a 800M point LTI (a 7.5% change), which is 8.5% of the $\approx$ 190000 seconds it would take to rebuild the index from scratch with a similar thread-count. We conclude that the merge process is significantly more cost-effective than periodically rebuilding the indices, which is the current choice of system design for graph indices. Further, `StreamingMerge` scales near linearly with the number of threads (see Figure 7). While the Delete phase scales linearly, the Patch and Insert phases scale sub-linearly due to intensive SSD I/O. Using fewer threads also results in more predictable search latencies (esp. 99% latency) due to the reduced SSD contention. This allows us to set the number of threads `StreamingMerge` uses to meet the desired update rate – 3600 updates/sec require 40 threads, but if we were only required to support 1000 updates/sec, we could choose to run `StreamingMerge` with 10 threads, and take advantage of higher search throughput and predictable latencies.

[4]Mean latency computed on a batch of 10k query points with one query per search thread

7 Conclusion

In this paper, we develop FreshVamana, the first graph-based fresh-ANNS algorithm capable of reflecting updates to an existing index using compute proportional to the size of updates, while ensuring the index quality is similar to one rebuilt from scratch on the updated dataset. Using update rules from FreshVamana, we design a novel two-pass `StreamingMerge` procedure which reflects these updates into an SSD-resident index with minimal write amplification. Using FreshVamana and `StreamingMerge`, we develop and rigorously evaluate FreshDiskANN, a highly-scalable fresh-ANNS system that can maintain a dynamic index of a billion points on a commodity machine while concurrently supporting inserts, deletes, and search operations at millisecond-scale latencies.

References

[1] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. 1993. Efficient similarity search in sequence databases. In *Foundations of Data Organization and Algorithms*, David B. Lomet (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 69–84.

[2] Laurent Amsaleg and Hervé Jegou. 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/. [Online; accessed 20-May-2018].

[3] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *Commun. ACM* 51, 1 (Jan. 2008), 117–122. https://doi.org/10.1145/1327452.1327494

[4] Alexandr Andoni and Ilya Razenshteyn. 2015. Optimal Data-Dependent Hashing for Approximate Near Neighbors. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing* (Portland, Oregon, USA) *(STOC '15)*. ACM, New York, NY, USA, 793–801. https://doi.org/10.1145/2746539.2746553

[5] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-Index: Pushing the Scalability-Accuracy Boundary for Approximate kNN Search in High-Dimensional Spaces. *Proceedings of the VLDB Endowment* 11 (04 2018). https://doi.org/10.14778/3204028.3204034

[6] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (Austin, Texas, USA) *(SODA '93)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 271–280. http://dl.acm.org/citation.cfm?id=313559.313768

[7] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020). http://www.sciencedirect.com/science/article/pii/S0306437918303685

[8] A. Babenko and V. Lempitsky. 2012. The inverted multi-index. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 3069–3076.

[9] Artem Babenko and Victor S. Lempitsky. 2014. Additive Quantization for Extreme Vector Compression. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 931–938. https://doi.org/10.1109/CVPR.2014.124

[10] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 2055–2063. https://doi.org/10.1109/CVPR.2016.226

[11] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *The European Conference on Computer Vision (ECCV)*.

[12] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. *CoRR* abs/1802.02422 (2018). arXiv:1802.02422 http://arxiv.org/abs/1802.02422

[13] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD Rec.* 19, 2 (May 1990), 322–331. https://doi.org/10.1145/93605.98741

[14] 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

[15] Erik Bernhardsson. 2018. *Annoy: Approximate Nearest Neighbors in C++/Python*. https://pypi.org/project/annoy/ Python package version 1.13.0.

[16] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover Trees for Nearest Neighbor. In *Proceedings of the 23rd International Conference on Machine Learning* (Pittsburgh, Pennsylvania, USA) *(ICML '06)*. Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/1143844.1143857

[17] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover Trees for Nearest Neighbor. In *Proceedings of the 23rd International Conference on Machine Learning* (Pittsburgh, Pennsylvania, USA) *(ICML '06)*. Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/1143844.1143857

[18] Leonid Boytsov. [n.d.]. https://github.com/nmslib/nmslib/issues/73

[19] A. Camerra, E. Keogh, T. Palpanas, and J. Shieh. 2010. iSAX 2.0: Indexing and Mining One Billion Time Series. In *2013 IEEE 13th International Conference on Data Mining*. IEEE Computer Society, Los Alamitos, CA, USA, 58–67. https://doi.org/10.1109/ICDM.2010.124

[20] Kenneth L. Clarkson. 1994. An Algorithm for Approximate Closest-point Queries. In *Proceedings of the Tenth Annual Symposium on Computational Geometry* (Stony Brook, New York, USA) *(SCG '94)*. ACM, New York, NY, USA, 160–164. https://doi.org/10.1145/177424.177609

[21] Kunal Dahiya, Deepak Saini, Anshul Mittal, Ankush Shaw, Kushal Dave, Akshay Soni, Himanshu Jain, Sumeet Agarwal, and Manik Varma. 2021. DeepXML: A Deep Extreme Multi-Label Learning Framework Applied to Short Text Documents. In *Proceedings of the 14th International Conference on Web Search and Data Mining* (Jerusalem, Israel) *(WSDM '21)*. Association for Computing Machinery, New York, NY, USA, 8.

[22] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin, and Heike Wehrheim. 2014. Quiescent Consistency: Defining and Verifying Relaxed Linearizability. In *FM 2014: Formal Methods*, Cliff Jones, Pekka Pihlajasaari, and Jun Sun (Eds.). Springer International Publishing, Cham, 200–214.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[24] 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* (Hyderabad, India) *(WWW '11)*. ACM, New York, NY, USA, 577–586. https://doi.org/10.1145/1963405.1963487

[25] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *Proc. VLDB Endow.* 13, 3 (2019), 403–420. https://doi.org/10.14778/3368289.3368303

[26] Evelyn Fix and J. L. Hodges. 1989. Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties. *International Statistical Review / Revue Internationale de Statistique* 57, 3 (1989), 238–247. http://www.jstor.org/stable/1403797

[27] Cong Fu and Deng Cai. [n.d.]. https://github.com/ZJULearning/efanna

[28] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graphs. *PVLDB* 12, 5 (2019), 461 – 474. https://doi.org/10.14778/3303753.3303754

[29] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 4 (2014), 744–755. https://doi.org/10.1109/TPAMI.2013.240

[30] Ruiqi Guo, Quan Geng, David Simcha, Felix Chern, Sanjiv Kumar, and Xiang Wu. 2019. New Loss Functions for Fast Maximum Inner Product Search. *CoRR* abs/1908.10396 (2019). arXiv:1908.10396 http://arxiv.org/abs/1908.10396

[31] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[32] 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* (Dallas, Texas, USA) *(STOC '98)*. ACM, New York, NY, USA, 604–613.

https://doi.org/10.1145/276698.276876
[33] M Iwasaki. [n.d.]. https://github.com/yahoojapan/NGT/wiki
[34] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data.
[35] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (Jan. 2011), 117–128. https://doi.org/10.1109/TPAMI.2010.57
[36] Herve Jegou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*. 861–864. https://doi.org/10.1109/ICASSP.2011.5946540
[37] Qing-Yuan Jiang and Wu-Jun Li. 2015. Scalable Graph Hashing with Feature Transformation. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, 2248–2254.
[38] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734* (2017).
[39] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. 2329–2336. https://doi.org/10.1109/CVPR.2014.298
[40] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *Proceedings of the VLDB Endowment* 11 (03 2018). https://doi.org/10.14778/3184470.3184472
[41] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2020), 1475–1488. https://doi.org/10.1109/TKDE.2019.2909204
[42] Wei Liu, Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2011. Hashing with graphs. In *ICML*.
[43] Yury A. Malkov and D. A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR* abs/1603.09320 (2016). arXiv:1603.09320 http://arxiv.org/abs/1603.09320
[44] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, USA.
[45] Milvus. [n.d.]. https://milvus.io/
[46] M. Muja and D. G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (2014), 2227–2240.
[47] Header only C++/python library for fast approximate nearest neighbors. [n.d.]. https://github.com/nmslib/hnswlib
[48] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. 2015. Neighbor-Sensitive Hashing. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 144–155. https://doi.org/10.14778/2850583.2850589
[49] Nick Pentreath, Abdulla Abdurakhmanov, and Rob Royce. 2017.
[50] Michael Sokolov. 2020. https://issues.apache.org/jira/browse/LUCENE-9004
[51] Suhas Jayaram Subramanya, Fnu Devvrit, Rohan Kadekodi, Ravishankar Krishnawamy, and Harsha Vardhan Simhadri. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 13748–13758. http://papers.nips.cc/paper/9527-rand-nsg-fast-accurate-billion-point-nearest-neighbor-search-on-a-single-node
[52] Kohei Sugawara, Hayato Kobayashi, and Masajiro Iwasaki. 2016. On Approximately Searching for Similar Word Embeddings. 2265–2275. https://doi.org/10.18653/v1/P16-1214
[53] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 1–12. https://doi.org/10.14778/2735461.2735462
[54] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming Similarity Search over One Billion Tweets Using Parallel Locality-Sensitive Hashing. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1930–1941. https://doi.org/10.14778/2556549.2556574
[55] Julie Tibshirani. 2019. https://www.elastic.co/blog/text-similarity-search-with-vectors-in-elasticsearch
[56] Vespa. [n.d.]. https://vespa.ai
[57] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. 2012. Scalable k-NN graph construction for visual descriptors. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 1106–1113. https://doi.org/10.1109/CVPR.2012.6247790
[58] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *CoRR* abs/2101.12631 (2021). arXiv:2101.12631 https://arxiv.org/abs/2101.12631
[59] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 194–205. http://dl.acm.org/citation.cfm?id=645924.671192
[60] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165. https://doi.org/10.14778/3415478.3415541
[61] Yair Weiss, Antonio Torralba, and Rob Fergus. 2008. Spectral Hashing. In *Proceedings of the 21st International Conference on Neural Information Processing Systems* (Vancouver, British Columbia, Canada) *(NIPS'08)*. Curran Associates Inc., Red Hook, NY, USA, 1753–1760.
[62] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A Fast and Accurate LSH Framework for High-Dimensional Approximate NN Search. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 643–655. https://doi.org/10.14778/3377369.3377374

A Recall Stability of FreshVamana Indices

Ramp-Up. We now measure the recall of an FreshVamana index as it grows in size. We start with a Vamana index built on a subset of 100K points randomly sampled from the million point datasets. In each cycle, we delete 10K points from the index at random, and insert 12K new points from the remaining pool of points, so that index grows by 2000 points. The experiment concludes when the index reaches the full size of a million points. We plot the search recall varies over the cycles in Figure 9 with varying search list size. While the recall trends down for a fixed search list size

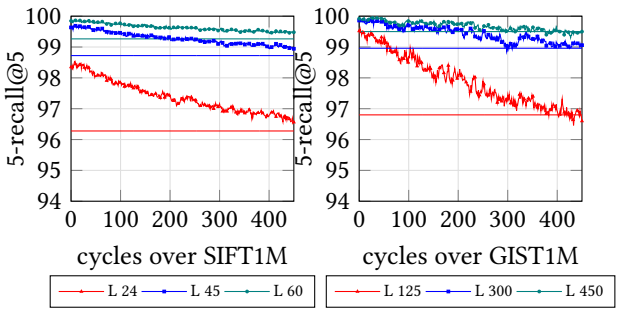

Figure 9. Search 5-recall@5 after each cycle of 12K insertions and 10K deletions to FreshVamana, ramping up from 100K to 1M points. Horizontal lines indicate recall of the corresponding batch built Vamana index.

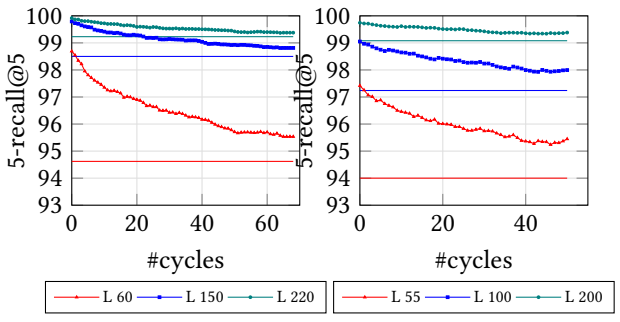

Figure 10. Search recall FreshVamana on SIFT100M while (left) ramping up from 1 point; and (right) ramping up starting from 30M points, and steady-state after 45 cycles. Horizontal lines indicate recall of the Vamana index with the same build time.

L as expected[5], note that the *final index quality* is at least as good as indices built in one shot using Vamana, whose recall for the same parameters is marked by horizontal lines.

B Index build times

In Table 1 we compare the build time of Vamana and FreshVamana for the same build parameters. The trade-off for this speed-up comes in the form of increased search latency for the same k-recall@k. In Figure 11, we show that using FreshVamana to make updates to the index takes only a fraction of the time to rebuild it from scratch using Vamana. We show a similar comparison of DiskANN and FreshDiskANN in Table 2. Despite using more than double the resources, building a 800M index from scratch using DiskANN takes more than 7x the time that FreshDiskANN takes to reflect the same changes into the index.

C Effect of α on recall stability

To determine the optimal value of α, we perform the FreshVamana steady-state experiments with different values of α. In the plots in Figure 3, we use the same value of α for building the intial Vamana index and for updating it. Other build and update parameters are same for each plot (R = 64, L = 75). We compare the evolution of search recall in the 95% range and average degree with different α. Finally we compare search recall versus latency for static indices built with different α to choose the best candidate. For all $\alpha > 1$, average degree increases over the course of the experiments and recall stabilises around the initial value. For static indices, latency at the same recall value improves from 1 to 1.2 after which further increasing α shows now noticeable improvement as evidenced by recall-vs-latency plots for Vamana indices in Figure 13. Since we want to minimise the memory footprint of our index, we choose the α value with best search performance and lowest average degree, which in this case is 1.2.

[5]This is true of any index – a larger index over data from the same distribution will provide lower recall with the search parameter/complexity.

Table 1. Index build times for Vamana and FreshVamana on `mem` with $R = 64$, $L_c = 75$, $\alpha = 1.2$

Dataset	Vamana	FreshVamana	Speedup
SIFT1M	32.3s	21.8 s	1.48x
DEEP1M	26.9s	17.7 s	1.52x
GIST1M	417.2s	228.1 s	1.83x
SIFT100M	7187.1s	4672.1s	1.54x

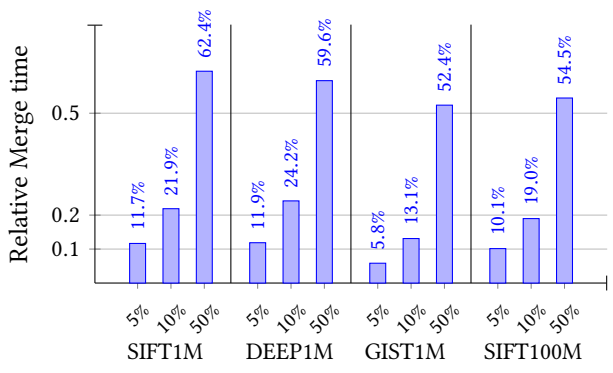

Figure 11. Time taken to merge delete and re-insert of 5%, 10%, and 50% of index size into a FreshVamana index, expressed relative to index rebuild time for Vamana.

Table 2. Full build time with DiskANN (96 threads) versus FreshDiskANN (40 threads) to update a 800M index with 30M inserts and deletes

Dataset	DiskANN(sec)	`StreamingMerge` (sec)
SIFT800M	83140 s	15832 s

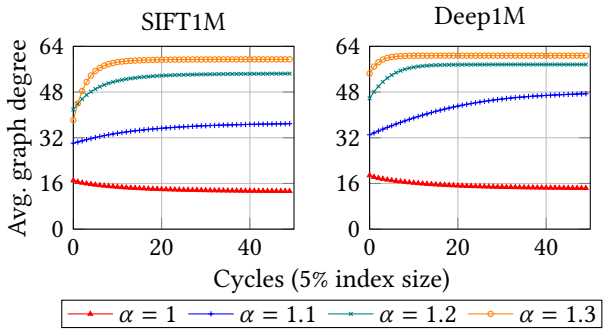

Figure 12. Evolution trends of recall and average degree for FreshVamana indices on SIFT1M and Deep1M over multiple cycles of inserting and deleting 5% of the index, where each trend represents a different α value used for building and updating the index.

Figure 13. Recall vs latency curves for Vamana indices on SIFT1M and Deep1M built with different values of α

D Amortized Cost of Delete Phase in `StreamingMerge`

Any non-trivial computation in the delete phase happens only for undeleted points $p \in P$ which have neighbors from the delete list. For each such point p, Algorithm 4 applies the pruning process on the candidate list consisting of the undeleted neighbors of p and the undeleted neighbors of the deleted neighbors of p to select the best R points from to its updated neighborhood. In order to perform an average-case analysis, let us assume that the delete set D is a randomly chosen set from the active points P, and suppose $|P| = N$ and $\frac{|D|}{N} = \beta$. The expected size of the candidate list will be $R(1-\beta) + R^2\beta(1-\beta)$. Here the first term accounts for undeleted neighbors of p and the second term accounts for undeleted neighbors of deleted neighbors of p. The expected number of undeleted points in the index is $N(1-\beta)$. Therefore the total number of expected operations in the delete phase will be proportional to $NR(1-\beta)^2(1+R\beta)$. This assumes that the complexity of the prune procedure is linear in the size of the candidate list which we validated empirically below. For large values of β, the $(1-\beta)^2$ term is diminishingly small and the deletion phase is quick. For small values of β (around 5%−10%) and typical values of $R \in [64, 128]$, $R\beta \gg 1$ and hence it dominates the value of the expression. Since $N\beta = |D|$, the time complexity becomes directly proportional to the size of the delete list.

We demonstrate the linear time complexity of Algorithm 3 in Figure 14. We delete a small fraction(10%) of SIFT1M Vamana index and record the time taken by Algorithm 3 as the candidate list size increases.

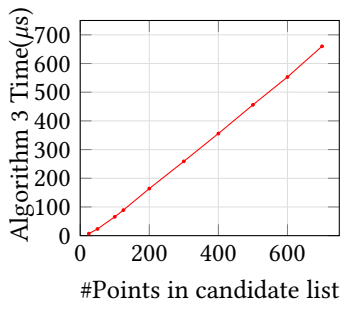

Figure 14. Trend of Algorithm 3 run time with increasing size of candidate list when 10% of the SIFT1M index is being deleted.

E k-recall@k for various k values

E.1 FreshVamana

E.1.1 Search Latency vs Recall. In Figures 15 to 17, we compare the search latencies for Vamana and build time-normalized FreshVamana (build parameters adjusted to match the build time of Vamana) for various k-recall@k. For 1-recall@1 and 10-recall@10, we compare latencies for 95%, 98% and 99% recall. For 100-recall@100, we compare over 98% and 99% recall because the lowest search list parameter L value gives 98% recall.

E.1.2 Recall stability of FreshVamana. In Figure 18, we demonstrate k-recall@k stability of FreshVamana for commonly used k values. We show the post-insertion recall trends for 1-recall@1, 10-recall@10 and 100-recall@100. For $k = 1$, we show how the 95% and 99.9% recall are stable. For $k = 10$, we show that 95% and 99% recall are stable. For $k = 100$, the lowest valid search list parameter L value is 100 and this gives 98% recall. So we show the stability of 98% and 99% recall.

E.2 FreshDiskANN

E.2.1 Search latencies over one merge cycle. In ????, we present the evolution of mean search latency for 100-recall@100 and 10-recall@10 over the course of one merge cycle in a 800M FreshDiskANN steady-state experiment.

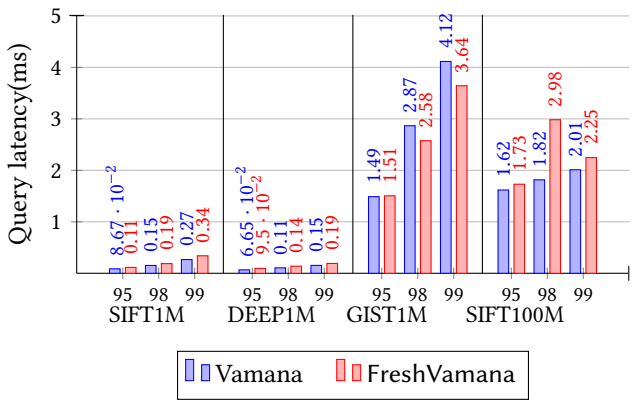

Figure 15. Query latency for Vamana and build-time normalized FreshVamana 1-recall@1 at 95%, 98%, and 99%.

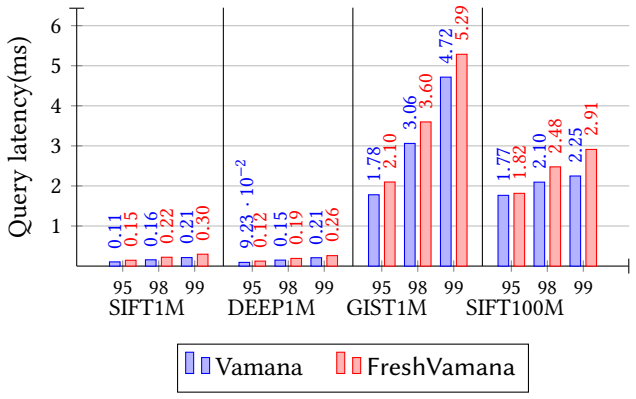

Figure 16. Query latency for Vamana and build-time normalized FreshVamana 10-recall@10 at 95%, 98%, and 99%.

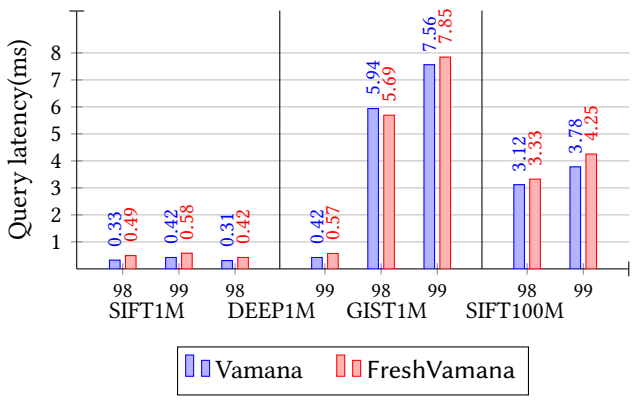

Figure 17. Query latency for Vamana and build-time normalized FreshVamana 100-recall@100 at 98%, and 99%.

F Search latency of FreshDiskANN

In Figure 21, we observe the effect of number of search threads on mean search latencies for 800M index when no merge is going on.

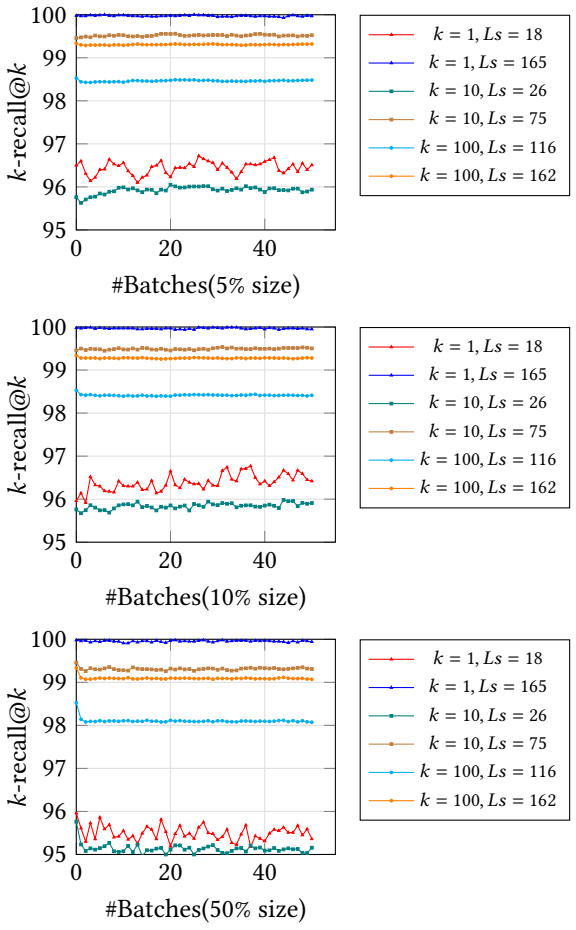

Figure 18. Post-insertion search k-recall@k for $k = 1, 10, 100$ of FreshVamana index over 50 cycles of deletion and re-insertion of 5%, 10% and 50% (rows 1, 2 and 3 respectively) of SIFT1M index with varying search list size parameter L.

G Concurrency during `StreamingMerge`

In this section, we present our observations on search latency during merge through in-depth experiments on FreshDiskANN merge with varying thread allocations. All experiments are 30M insertions and deletions into a 800M FreshDiskANN index.

G.1 Search threads fixed - varying merge threads

We run the merge on SIFT800M index with different thread allocations to understand the effect of merge on search latency. In Figure 8, we plot a smoothed curve of mean search latencies when merge uses 20 and 40 threads. Merge with 40 threads takes approximately half the time as that with 20, so there are two x axes adjusted to roughly align their Delete, Insert and Patch phases. As evident from the figure, search

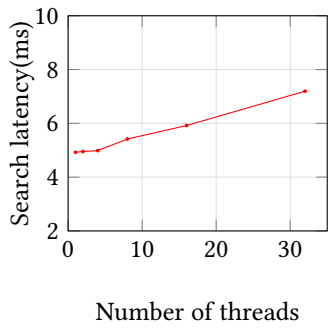

Figure 21. Trend of mean latencies for 95% search recall on a 800M SIFT index with different number of threads. Each point is calculated over a search batch of 10000 queries

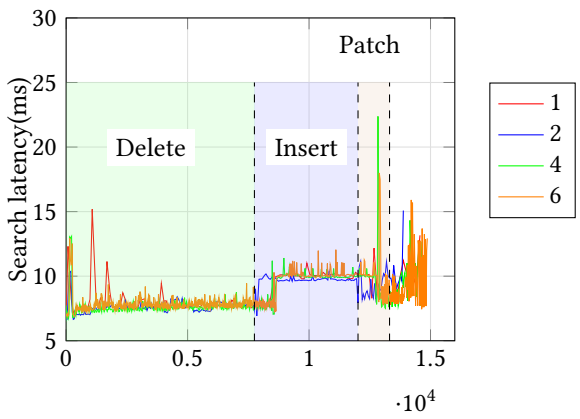

Figure 22. Trend of mean search latencies for 92% search recall, zoomed in over one cycle of inserting and deleting 30M points concurrently into a 800M SIFT index, using different number of threads for search. Each point is the mean latency over a search batch of 10000 queries.

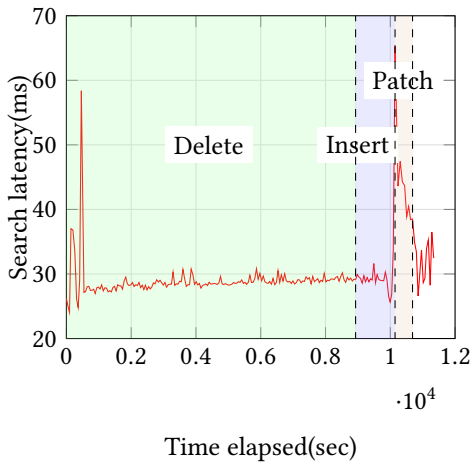

Figure 19. Trend of mean search latencies for 95% search 100-recall@100, zoomed in over one cycle of inserting and deleting 30M points concurrently into a 800M SIFT index, using different 10 for search. Each point is the mean latency over a search batch of 10000 queries.

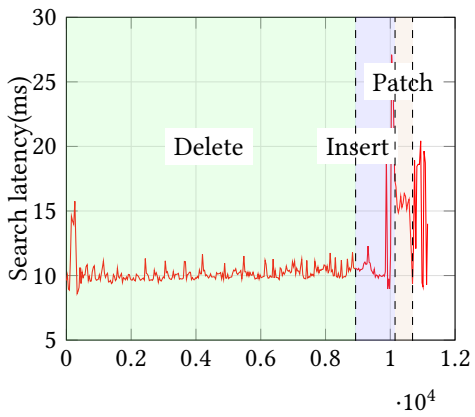

Figure 20. Trend of mean search latencies for 95% search 10-recall@10, zoomed in over one cycle of inserting and deleting 30M points concurrently into a 800M SIFT index, using different 10 for search. Each point is the mean latency over a search batch of 10000 queries.

latencies with 40 thread merge are consistently higher in the Delete and Insert phases of merge.

G.2 Merge threads fixed - varying search threads

We run the merge on SIFT800M index with different thread allocations to understand the effect of number of search threads used during merge on search latency. We increase the number of search threads while fixing 40 threads for merge, and observe how the search latency trend evolves in over one merge cycle Figure 22.