



# SPFresh: Incremental In-Place Update for Billion-Scale Vector Search

Yuming Xu[1, 2] Hengyu Liang[1, 2] Jin Li[3, 1, 2*] Shuotao Xu[2] Qi Chen[2] Qianxi Zhang[2]
Cheng Li[1] Ziyue Yang[2] Fan Yang[2] Yuqing Yang[2] Peng Cheng[2] Mao Yang[2]
[1] University of Science and Technology of China [2] Microsoft Research Asia [3] Harvard University

## Abstract

Approximate Nearest Neighbor Search (ANNS) on high dimensional vector data is now widely used in various applications, including information retrieval, question answering, and recommendation. As the amount of vector data grows continuously, it becomes important to support updates to vector index, the enabling technique that allows for efficient and accurate ANNS on vectors.

Because of the curse of high dimensionality, it is often costly to identify the right neighbors of a new vector, a necessary process for index update. To amortize update costs, existing systems maintain a secondary index to accumulate updates, which are merged with the main index by globally rebuilding the entire index periodically. However, this approach has high fluctuations of search latency and accuracy, not to mention that it requires substantial resources and is extremely time-consuming to rebuild.

We introduce SPFresh, a system that supports in-place vector updates. At the heart of SPFresh is LIRE, a lightweight incremental rebalancing protocol to split vector partitions and reassign vectors in the nearby partitions to adapt to data distribution shifts. LIRE achieves low-overhead vector updates by only reassigning vectors at the boundary between partitions, where in a high-quality vector index the amount of such vectors is deemed small. With LIRE, SPFresh provides superior query latency and accuracy to solutions based on global rebuild, with only 1% of DRAM and less than 10% cores needed at the peak compared to the state-of-the-art, in a billion scale disk-based vector index with a 1% of daily vector update rate.

***CCS Concepts:*** • **Information systems** → **Information storage systems**; **Information retrieval**.

***Keywords:*** Vector Search, Incremental Update, Billion-scale



*SOSP '23, October 23–26, 2023, Koblenz, Germany*

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613166

**ACM Reference Format:**
Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search . In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3600006.3613166

## 1 Introduction

Today deep learning models can embed almost all types of data, including speech, vision, and text information, into *multi-dimensional vectors* with tens or even hundreds of dimensions. Such vectors are critical for complex semantic understanding tasks [42, 49]. To enable effective vector analysis, vector nearest neighbor search (NNS) systems have become critical system components for an increasing number of online services like search [35] and recommendation [57].

To satisfy the strict query latency requirement for these online services, vector search systems often resort to *approximate nearest neighbor search* (ANNS) [17, 22, 34, 51, 56, 59, 65, 69], to locate as many correct results as possible (i.e., query accuracy). At the heart of a large-scale NNS system is a *vector index*, a key data structure that organizes high-dimensional vectors efficiently for high-accuracy low-latency vector searches [3, 9, 14, 38, 58, 67].

Like traditional indices, a high-quality vector index organizes a quick "navigation map" of vectors based on the vector proximity in a high dimensional space. The proximity measurements are often implemented with "shortcuts", which only exist between a pair of vectors with a short distance. A search query traverses the datasets based on "shortcuts" the result set. The quality of the index for efficient traversal is highly dependent on the quality of shortcuts, where insufficient shortcuts miss relevant vectors, and extraneous shortcuts incur excessive traversal and storage costs. For high-dimensional data, vector indices require careful construction to produce a sufficient amount of high-quality "shortcuts", often as vector partitions [14, 67] or graphs of vector data [56, 65].

To add fuel to fire, there is a strong desire to support *fresh update* of vector indices because current systems generate a vast amount of vector data continuously in various settings. For example, 500+ hours of content are uploaded to

*Work done during his final-year study at USTC and his internship at MSRA.

YouTube[21] every minute, one billion new images are updated in JD.com every day [34], and 500PB fresh unstructured data are ingested to Alibaba during a shopping festival [69]. Fresh updates require vector indices to incorporate new vectors at unprecedented scale and speed while maintaining their high-quality to produce low query latency and high query accuracy of approximate vector searches.

However, it is non-trivial for vector indices to maintain high-quality "shortcuts" when updating vectors with hundreds of dimensions. Graph-based indices have inherent high cost to update vectors in place, because each insertion or removal of vector datum often requires examining the entire graph to update the edges in a high-dimensional space. One silver lining to fast vector index update is that cluster-based index, which is less costly to update than the graph-based index. Vector insertion and removal only require constant local modification to vector partitions(s). Nevertheless, as updates accumulate per vector partition, the index quality deteriorates because the data distribution skews over time, which makes partition sizes uneven and hence hurts both query latency and accuracy [34].

Because of the difficulty of running vector index updates in place, existing ANNS system support vector updates [34, 53, 59, 65] out-of-place, by periodic *rebuilding of global index*. A batch of vector updates are accumulated and indexed separately, i.e., out-of-place, and are periodically merged to the base index by rebuilding the entire index. Such practices introduce significant resource overheads to ANNS systems. For example, to build a global DiskANN index for a 128G SIFT dataset with a scale of 1 billion, it would require a peak memory usage of 1100GB for 2 days, or 5 days under a memory usage of 64GB with 32 vCPUs [56]. Such rebuilding can even consume more resources than the index serving costs (§2.3). In addition, such an out-of-place update method hurts the search performance of online services because it follows a Log-Structured-Merge (LSM) style for updates [53], which trades read performance for write optimization.

To scale to large vector datasets with lower costs, this paper presents SPFresh, a disk-based vector index that supports lightweight incremental in-place local updates without the need for global rebuild. SPFresh is based on the state-of-the-art cluster-based vector index design, capable of incorporating vector index updates online with low overheads while maintaining good index quality for high search performance and accuracy for billion-scale vector datasets. The core of SPFresh is LIRE, a Lightweight Incremental REbalancing protocol that accumulates small vector updates of local vector partitions, and re-balances local data distribution at a low cost. Unlike expensive global rebuilds, LIRE is capable of maintaining index quality by fixing data distribution abnormalities locally on-the-fly.

The key design rationale behind LIRE is to leverage a vector index that is *already* in a well-partitioned state. Small vector updates to a high-quality vector partition may only incur changes in itself and its neighboring partitions. Because the updates are small, the corresponding changes are most likely to be limited in a local region. This makes the entire rebalancing process lightweight and affordable.

Despite this opportunity, rebalancing is still non-trivial. In particular, LIRE needs to address the following challenges. 1) In order to keep search latency short, LIRE needs to maintain an even distribution of partition sizes via timely split and merge. 2) In order to keep search accuracy high, LIRE needs to identify the smallest set of vectors that cause data imbalance in the index. These vectors should be reassigned to maintain high index quality. 3) An implementation of LIRE should be lightweight with negligible performance impacts on the foreground search.

LIRE tackles these challenges by making the following four contributions:

- LIRE keeps partition size distribution uniform by splitting and merging partitions proactively and incrementally.
- LIRE formally identifies two necessary conditions for vector reassignment based on the rule of nearest neighbor posting assignment (NPA). With the necessary conditions, LIRE opportunistically identifies a minimal set of neighborhood vectors to adapt to data distribution shifts.
- An implementation of LIRE is decoupled as a two-stage feed-forward pipeline, which moves the background split-reassign off from the critical path of foreground update. Each pipeline stage is multi-threaded to saturate the high IOPS of a high-performance NVMe device.
- An SSD-backed user-space storage engine dedicated to LIRE, which bypasses legacy storage stack, prioritizes partition reads, and optimizes for partition appends.

Experiments show SPFresh outperforms state-of-the-art ANNS systems that support fresh updates on all fronts, with low and constant search/insert latency, high query accuracy, as well as efficient resource usages for billion-scale vector datasets. Instead of an additional 1000GB memory and 32 cores needed by DiskANN global rebuild, SPFresh outperforms DiskANN by 2.41× lower tail latency on average with only 10GB memory and 2 cores. Moreover, SPFresh reaches the IOPS limitation with stable performance and resource utilization. It simultaneously reaches peak 4K QPS search throughput and 2K QPS update throughput on a single NVMe SSD disk with 15 cores.

## 2 Background and Related Work

In this section, we present the basic operations in ANNS-based vector search and introduce two mainstream on-disk vector indices and their respective index update challenges.

### 2.1 Vector Search and ANNS

A common use-case of vector search involves finding the most similar items in a large dataset based on a given query. This process is often used in recommendation systems, search

engines, and natural language processing tasks. As shown in Figure 1, to find similar images from dataset given a query image, the system first represents each image in the dataset as a high-dimensional vector through a deep learning model. The query image is also encoded into a vector in the same high-dimensional space. Then, the system calculates the similarity between the query vector and each vector in the dataset using a similarity metric, such as cosine similarity or Euclidean distance. The system ranks the images based on their similarity scores and returns the top results to the user. Essentially, the search is to find the query vector's nearest neighbors in a high-dimensional space.

Formally, given a vector set $X \in \mathbf{R}^{n \times m}$ containing $n$ $m$-dimensional vectors and a query vector $\mathbf{q}$, vector nearest neighbor search aims to find a vector $\mathbf{x}^*$ from $X$ such that $\mathbf{x}^* = \arg\min_{\mathbf{x} \in X} \text{Dist}(\mathbf{x}, \mathbf{q})$, where Dist is the similarity metric discussed above. This definition can be extended to $K$-nearest neighbor (KNN) search [67]. Modern machine learning models typically generate vectors with dimensions ranging from 100 to 10,000, or even more. For example, GPT-3 generates four sizes of embedding vectors with dimensions ranging from 1024 to 12288 [48]. The high dimensionality makes it challenging to find the exact $K$ nearest neighbors efficiently [12]. To address this issue, recent systems commonly rely on *approximate* nearest neighbor search (ANNS) [51, 56, 67] to make the effective trade-off across resource cost, result quality, and search latency, thus scaling to large vector datasets.

Due to its approximate nature, search result accuracy becomes an important metric to gauge the quality of a vector index. In ANNS, `RecallK@K` is commonly used to measure result quality. For an approximate KNN query, `RecallK@K` is defined as $\frac{|Y \cap G|}{|G|}$, where $Y$ is the query's result set, and $G$ is the query's ground truth result set, $|Y| = |G| = K$.

### 2.2 Vector Index Organization

A vector index can be abstracted as a logical graph, where a vertex represents a vector, and an edge denotes the close proximity of two vectors in terms of distance. And vector indices for ANNS can be categorized into *fine-grained graph-based vector indices* and *coarse-grained cluster-based vector indices*. These two methods can be applied to both in-memory or on-disk scenarios.

In this paper, we only focus on on-disk vector indices since they are more cost-effective for large-scale vector-sets. Meanwhile, they pose a unique challenge for vector updates since disk writes are much more costlier than DRAM writes. ***Fine-grained graph-based vector indices*** represent each vector as a vertex, and an edge exists between two vertices if they are close in distance. Locating $K$ nearest vectors often involves best-first graph traversals, where neighboring vertices are explored in ascending distance order.

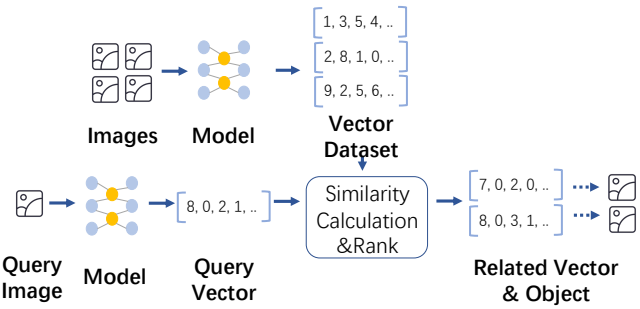


**Figure 1.** An example of vector search: to find the most similar images in an image dataset. Query images and data images are all represented as vectors.

Example vector index solutions based on fine-grained graphs include *neighborhood-graph based methods* [15, 16, 19, 20, 23, 38, 60, 63] which organize all the vectors into a neighborhood graph with each vector connected to its nearest vectors, and *hybrid methods* [26, 27, 62, 68] which consist of space-partition trees and a neighborhood graph to take advantage of both tree and graph data structures. *Space-partition-tree based methods* [4, 6, 8, 10, 13, 18, 36, 39, 43, 45, 46, 55, 64, 72] can be treated as a special kind of fine-grained graphs. They use a tree to represent the space division and the vector to subspaces mapping. Most of these solutions are based on in-memory implementations for performance and are expensive to scale to billion-scale data-sets.

Only a few fine-grained graph-based vector indices are optimized for secondary storage (e.g., DiskANN [56] and HM-ANN [51]). Similar to external graph systems [31, 33, 52, 74], these fine-grained graph-based vector indices are stored in two parts: vertex data and edge data as vertex adjacency lists. Edge data are stored in secondary storage, and vertex data are either on disk [51] or in memory [56], where in-memory vertex data speed up computation, i.e. distance calculation in the case of ANNS [56].

To reduce search costs, DiskANN [56] employs a fixed graph traversal strategy, where it caches the neighborhood of the fixed starting point *in memory* to speed up graph traversal in the initial stage. DiskANN further maintains an in-memory copy of compressed vertex data (using product quantization) to speed up distance calculation during graph traversals. In contrast, HM-ANN [51] constructs a hierarchical in-memory graph where it can navigate to the nearest entry point to the main graph on secondary storage, and thus efficiently identify the target region for nearest vectors.

Although effective for vector search, graph-based vertex indices are unfriendly to updates (details in §2.3).
***Coarse-grained cluster-based vector indices*** organize vector indices via clustering, where vectors in close proximity are kept in the same partition. Logically, vectors in each partition represent a fully-connected graph, while vectors across different partitions have no edge. Since no explicit

edge data are required, coarse-grained cluster-based vector indices require much smaller storage. Vector search on cluster-based vector indices first identifies candidate partitions by measuring the distance to the partitions' centroids and then calculates the $K$ nearest vectors from the candidate partitions via a full scan.

Coarse-grained cluster-based vector indices include *hash-based methods* [14, 24, 28, 32, 44, 50, 54, 61, 70, 71] which use multiple locality-preserved hash functions to do the vector-to-partition mapping, and *quantization-based methods* [5, 7, 17, 30, 73] which use Product Quantization(PQ) [29] to compress the vectors and KMeans to generate the vector-to-partition mapping codebooks.

A cluster-based vector index should preserve the balance across partitions to achieve low tail search latency. However, ANNS indices leveraging locality-sensitive hashing [14, 28, 66, 70, 71] and k-means [37] for clustering pay less attention to partition balance. Such ANNS indices often produce uneven partitions and thus are only adopted by in-memory systems where the absolute tail latency is much less pronounced than that of an on-disk solution.

SPANN [67] is the first on-disk vector index that achieves low tail search latency through balanced clustering. SPANN divides a vector-set into a large number of balanced partitions stored on disk and keeps the centroids of the partitions in the memory for quick identification of candidate partitions during search. It employs several techniques to ensure a well-balanced partition state (details in §3.1). SPANN achieves state-of-the-art performance on memory cost, result quality, and search latency across multiple billion-scale datasets.

Cluster-based vector indices are *friendly to updates* because each vector insertion or deletion only involves local modifications of vector data in the corresponding partition. However, a naive update on local partitions may eventually lead to imbalanced clusters and consequently deteriorate search tail latency and accuracy (more in §2.3).

### 2.3 Freshness Demands and Challenges

Modern ANNS systems are required to accommodate billions of vector updates every day while still preserving low query latency and high query accuracy. With the new popular OpenAI ChatGPT retrieval plugin [47], some AI applications built atop even require real-time updates to keep up with the updates on their personal documents or contexts, such as files, notes, emails, and chat histories, all in the form of vector, in order to retrieve most relevant snippets as new prompts. However, it is non-trivial for vector indices to maintain index quality when updating vectors.

**Out-of-place update.** For vector inserts, fine-grained graph-based indices have to connect a new vector to hundreds of neighboring vectors in order to maintain sufficient shortcuts in the high-dimensional space. Deletions of vectors are even more expensive as they often involve the total scan of a unidirectional graph.

| | Memory | CPU | Time |
|---|---|---|---|
| DiskANN | 1100 GB | 32 cores | 2 days |
| | 64GB | 16 cores | 5 days |
| SPANN | 260 GB | 45 cores | 4 days |

**Table 1.** Global rebuild costs of disk-based ANNS indices for billion-scale datasets.

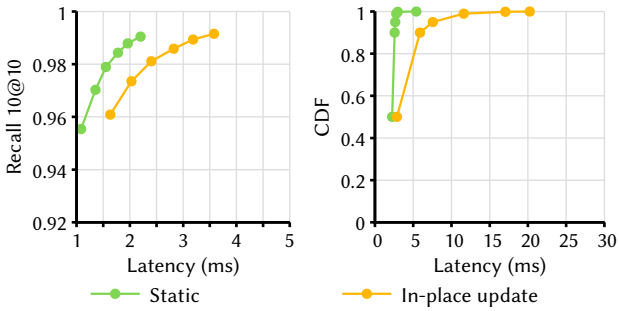


**Figure 2.** Recall and tail latency in two system settings, namely, *static* and *in-place update*. The static setting refers to an index of 2 million vectors, while the in-place update setting refers to an index built by applying 0.5 million vector updates to a base index of 1.5 million vectors. The index system is SPANN, and the dataset is sift [1].

To overcome the difficulty of in-place updates, existing systems resort to out-of-place updates with *periodical global updates*. These systems accumulate and index delta vector updates in a separate, secondary in-memory index, which is periodically merged to the base index by a global index rebuilding process to maintain good index quality. Many popular ANNS systems, such as ADBV [69] and Milvus [65], use this method. To defer expensive global updates, Milvus even introduces multiple delta indices in memory. However, this approach requires vector search to examine both main and secondary indices, which increases resource demands and hurts search performance. Table 1 shows global rebuilds are both resource-hungry and time-consuming. For example, rebuilding a 1-billion vector index [56] for DiskANN, a recent disk-based system, needs 1100GB DRAM, 32 vCPUs, for 2 days. When limiting resources to 64GB memory and 16 vCPUs, the rebuilding time becomes significantly longer, e.g., 5 days for DiskANN. This stressful setting could also lead to a catastrophic drop in query performance because of severe computational resource starvation.

**Early attempts to in-place update.** Compared to out-of-place vector updates, few systems support in-place updates. Vearch [34] is one of such systems based on cluster-based in-memory vector indices, where it inserts a new vector to its nearest partition (a.k.a. posting, the partition is implemented as a posting list) and supports deletions by maintaining a tombstone bitmap for result filtering.

To understand the impact of Vearch's design to on-disk index, we apply Vearch's design to SPANN, the *only* partition-based on-disk vector index system. Figure 2 shows that updating one-third of the vectors degrades the query recall by more than one point and increases tail latency by 4X, compared to static index building. The reasons are two-fold: 1) With the growth of the data size, query latency will increase due to the expansion of the posting length. 2) Since the centroids for each partition are fixed, the recall will decline as static centroids cannot capture the gradual distribution shift in the partition. To conclude, to maintain high index quality and stabilize search latency, although Vearch and the modified SPANN do not require *out-of-place* data structure, they still require periodical global rebuilds. For instance, Vearch performs weekly rebuilds. The rebuild overheads might be acceptable for *in-memory* vector indices. However, for disk-based indices like SPANN, global rebuilds are expensive, as shown in Table 1.

In summary, existing graph-based and cluster-based solutions, regardless of in-place or out-of-place, all rely on periodic global index rebuilding to preserve index quality and stabilize search performance. However, this process entails considerable resource consumption.

### 2.4 Our Goals

To this date, efficient fresh update for disk-based vector index is still an open challenge. In this paper, we aim to propose a new disk-based ANNS system to fulfill the following goals: 1) low resource cost to maintain the index for large-scale vector datasets; 2) support high throughput and low latency vector queries for both search and update; and 3) new vectors can be recalled in high probability.

To achieve this, motivated by the above understandings, we choose to follow the coarse-grained cluster-based approach to build our on-disk index, but differ from existing solutions significantly by avoiding global rebuilds completely. The proposed solution, SPFresh, performs in-place, incremental updates in the index data structure to adapt to the data distribution shift. To this end, SPFresh incorporates a Lightweight Incremental RE-balancing (LIRE) protocol, which efficiently identifies a minimal amount of partition updates introduced by new vectors for maintaining index property and thus eliminating visible accuracy loss. Equally importantly, we also address a few system challenges to make LIRE re-balance sufficiently fast and cheap to alleviate negative impacts on search latency, in particular, *tail latency*. Essentially, LIRE can be considered as an efficient compaction technique in the high-dimensional space.

## 3 LIRE Protocol Design

LIRE is built on SPANN [67], the state-of-the-art disk-based vector index system. In this section, we first introduce SPANN briefly and then elaborate LIRE in detail.

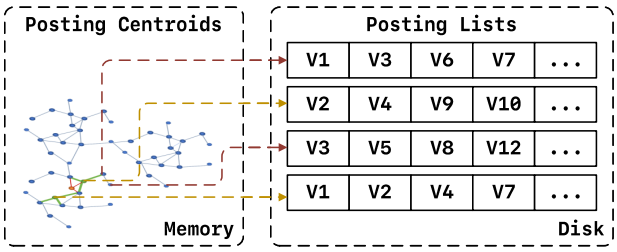


**Figure 3.** SPANN index data architecture.

### 3.1 SPANN: A Balanced Cluster-based Vector Index

SPANN [67] is a billion-scale cluster-based vector index optimized for secondary storage. Figure 3 shows the overall SPANN index structure. SPANN stores the vectors as a large number of postings* on disk, each represents a cluster of close-by vectors. Moreover, SPANN organizes a graph-based in-memory index, SPTAG [68], for the centroids of all postings, to quickly identify relevant postings for a query.

For a query, it first identifies the closest posting centroids through the in-memory index, and then loads the corresponding postings from disk to memory for further search. To control the tail latency and maintain high search recall, SPANN makes postings *well balanced* by maintaining two key properties. 1) SPANN divides the vectors *evenly* into a large number of small-sized postings by a fast hierarchical balanced clustering algorithm, so that each query visits a similar amount of vectors for bounded search tail latency. 2) SPANN replicates a few vectors in boundaries across postings, which sufficiently maintains high search recalls.

The *balanced* SPANN index inspires us to propose a new lightweight incremental re-balancing (LIRE) protocol. *The intuition here is a single vector update to a well-balanced index may only incur changes in a local region.* This makes the entire rebalancing process lightweight and affordable.

### 3.2 LIRE: Lightweight Incremental RE-balancing

A key property of a well-partitioned vector index is the *nearest partition assignment* (NPA): each vector should be put into the nearest posting so that it can be well represented by the posting centroid. As continuous vector updates to a posting may degrade query recalls and latency, SPFresh will split a posting after it grows to the preset maximum length. However, a naive splitting can violate the NPA property of the index.

Figure 4 illustrates a case of NPA violation. Originally, there were two postings, A and B, near each other, where the blue dots represent their centroids. At a certain point, posting A exceeds the length limit upon vector insertions and is split into two new postings, A1 and A2. The orange dots represent the new centroids of A1 and A2 elected after

*We use "posting" and "partition" interchangeably in this paper.

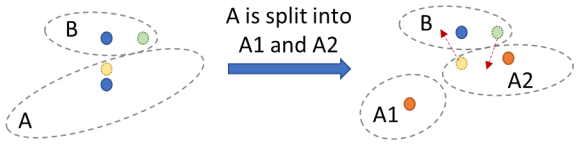


**Figure 4.** Posting split violates the NPA property.

the split. With a naive split, the vectors in posting A will *only* go to Posting A1 and A2 respectively, based to their distance to new centroids. For illustrative purposes, we assume the yellow dot (a vector) goes to A2.

However, the creation of new centroids via a spit makes previous NPA-compliance obsolete for vectors in the nearby postings, A1, A2 and B. First, the nearest posting of the yellow dot changes to B, since B's centroid is closer to the yellow dot than A2's centroid. In this case, using the centroid of A2 to represent the yellow dot violates NPA. Second, the nearest posting of the green dot, which was B before the split, changes to A2. These two violations degrade the index quality and result in low recalls.

To fix the NPA violations after splits and maintain the high index quality, we design LIRE protocol, which reassigns vectors in nearby postings of a split. At its core, LIRE protocol consists of five basic operations: Insert, Delete, Merge, Split, and Reassign.

***Insert & Delete:*** LIRE directly inserts a new vector to the nearest partition following the original SPANN index design. LIRE also ensures the deleted vectors will not appear in the search results and will eventually be removed from the corresponding postings.

Note that Insert and Delete are *external interfaces* exposed to users. The remaining three operations are *internal interfaces* and thus are oblivious to users. These three operations work together to keep the size of the posting small and balanced and to ensure vectors are assigned to the right posting, following the NPA property.

***Split:*** When a posting exceeds a length limit, LIRE evenly splits the oversized posting into two smaller ones. As introduced in the previous section, vectors in the neighboring postings may violate the NPA property after the split. Thus, a reassign process (detailed in §3.3) will be triggered for the vectors in the split postings as well as nearby postings.

***Merge:*** When a posting size is smaller than a lower threshold, LIRE identifies its nearest posting as candidates for merging. In particular, LIRE's merge process deletes one posting with its centroid (e.g., the shorter posting), and appends them to the other posting directly. After that, a reassign process is required for the vectors of the deleted posting because the deletion of their old centroid might break the NPA rule after being merged with the other posting. Reassignments will not induce splits of the merged posting because vectors can only be reassigned out. However, a reassigned vector may trigger the split to the target posting. Despite such cascading effects, §3.4 shows that LIRE's split-reassignment process will always converge.

### 3.3 Reassigning Vectors

Reassigning vectors can be expensive, because they require expensive changes to the on-disk postings for each reassigned vector. Thus it is critical to identify the right set of neighborhoods (neighboring postings) to avoid unnecessary reassignment. For a merged posting , only vectors from deleted posting require reassignment, because the deletion of a centroid does not break NPA compliance of vectors from undeleted postings.

On the other hand, a split not only deletes a centroid but creates two new ones. Therefore a split creates more complex scenarios of potential NPA violations. After examining Figure 4, we derive two necessary conditions for reassignment after splitting, assuming the high-dimensional vector space is Euclidean.

First, a vector $v$ in the old posting with centroid $A_o$ is required to consider being reassigned if:

$$D(v, A_o) \leq D(v, A_i), \forall i \in 1, 2 \tag{1}$$

where $D$ denotes the distance, $A_o$ represents the old centroid before splitting, and $A_i$ represents any of the two new centroids. This reassignment condition means that if the old (deleted) centroid $A_o$ is the closest centroid to the vector $v$, compared to new centroids ($A_1$ and $A_2$), then it cannot be ruled out the possibility that $v$ is closer to a centroid of some nearby posting than new centroids (e.g., $B$ in Figure 4). Note that this is a *necessary condition.* On the contrary, if $D(v, A_o) > D(v, A_i)$, this shows $v$ is having a better centroid than the old one. In this case, the neighboring certroid (e.g., $B$) cannot be better than the new ones, i.e., $D(v, B) > D(v, A_i)$ based on the NPA property of $D(v, B) > D(v, A_o)$. Thus there is no need to check reassignment in this case.

Second, a vector $v$ in the nearby posting with centroid $B$ needs to consider being reassigned if:

$$D(v, A_i) \leq D(v, A_o), \exists i \in 1, 2 \tag{2}$$

This is a *necessary condition* for a vector in posting $B$ to be reassigned to a newly split posting with centroid $A_i$. Equation 2 suggests $v$'s new neighboring centroids are getting closer (better) than the old (deleted) one. Therefore it is necessary to check if the new and closer centroids are in fact closer than $v$'s existing centroid B (the blue dot w.r.t the green dot in Figure 4). On the other hand, if the two new centroids are farther away from any vector $v$ outside the old posting, this means the two centroids are worse than the old one $A_o$, which is already farther away than $v$'s existing centroid. In this case, there is no need to check the reassignment of $v$. Hence the necessary condition.

According to the two necessary conditions, a complete checking process would be extremely expensive, because it

requires computing and comparing $D(v, A_o)$ and $D(v, A_i), i \in 1, 2$ for *all vectors* in the dataset. To minimize the cost, LIRE only examines nearby postings for reassignment check by selecting several $A_o$'s nearest postings, over which two condition checks were applied to generate the final reassign set. Experiments in Section 5 show empirically that only a small number of nearby postings for the two necessary condition checks is enough to maintain the index quality.

After obtaining vector candidates for reassignment, LIRE executes the reassignment. For vector candidate $v$, LIRE first searches $v$'s new closest posting , then performs NPA check to get rid of false-positives: if a vector actually does not need reassignment, the reassign operation is aborted. Otherwise, LIRE appends $v$ in the newly identified posting that is NPA-compliant and then deletes $v$ in the original posting.

### 3.4 Split-Reassign Convergence

In this section we prove that a split-reassign action to the vector index, despite the potential of triggering cascading split-reassign actions, will converge to a final state and terminate in finite steps. We first formally define the states of vector index and the events triggering state transitions. Then we prove that state transition will converge and terminate.

***Index State:*** The state of a vector index for a vector data-set $V$ comprises of two parts.

$$C : \text{set of posting centroids.} \tag{3}$$

$$M : \text{vector membership to centroid(s).} \tag{4}$$

According to LIRE, given $C$, each vector in $M$ is assigned to its nearest centroid in $C$, i.e., $M$ is *uniquely* determined by $C$.

***Index-State Transitions:*** The state transition is triggered by two types of events:

$$E_{insert} : \text{a vector } v \text{ is inserted into the vector index.} \tag{5}$$

$$E_{delete} : \text{a vector } v \text{ is deleted into the vector index.} \tag{6}$$

A reassign of vector $v$ is considered as an $E_{delete}$ of $v$ followed by an $E_{insert}$ of $v$. Note that an event will change the state of $M$, however it would not necessarily alter the state of $C$.

Also note that only an event $E_{insert}$ may incur a *split action* of the vector index, which alters the state of $C$ and subsequently $M$. Since $C$ uniquely determines $M$, we can only focus on the state change of $C$.

***Split-Reassign Convergence Proof:*** $E_{delete}$ may eventually trigger a merge during a search process (according to LIRE), and the merge obviously will terminate.

Suppose an $E_{insert}$ triggers a sequence of changes of $C$, denoted as $C_i$,$C_{i+1}$, ..., $C_{i+N}$. To prove the convergence is to show that $N$ is a finite number.

We note that $C$ has the following properties:

- $|C| \leq |V|$: The cardinality of $C$ is bounded and no greater than the cardinality of $V$, i.e., the vector dataset.

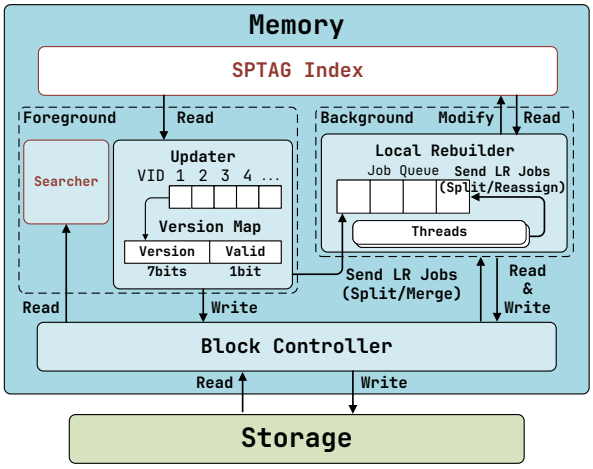


**Figure 5.** SPFresh architecture (LR means Local Rebuild).

- $|C_{i+1}| = |C_i|+1$: Each split action will delete an old centroid from $C_i$, and adds two new centroids to it. Therefore, the cardinality of $C$ *always* increases by one per split action.

Based on Property 2, $|C_{i+N}| = |C_i| + N$. And according to Property 1, $N \leq V - |C_i|$ because $|C_{i+N}| \leq V$. Since $|V|$ is finite, $N$ must also be finite. Therefore the split action must terminate in finite steps. □

## 4 SPFresh Design and Implementation

### 4.1 Overall Architecture

Figure 5 shows the system architecture of SPFresh. SPFresh reuses the SPANN SPTAG index (depicted in Figure 3) for fast posting centroid navigation as well as its searcher to serve queries. It further introduces three new modules to implement LIRE, namely, a light-weight *In-place Updater*, a low-cost *Local Rebuilder*, and a fast storage *Block Controller*.

***Updater*** appends a new vector at the tail of its nearest posting and maintains a *version map* to keep track of vector deletion by setting a corresponding tombstone version to prevent deleted vectors from appearing in the search results. The map is also used to trace the replica of each vector. By increasing the version number, it marks the old replicas as deleted. The system keeps a global in-memory version map and stores vectors along with the version number on disk. A vector is stale if the in-memory version number is greater than that on the disk. This can be used for garbage collection caused by reassignment. The use of version can defer and batch the garbage collection so as to control the I/O overhead of vector removal. After vector insertions are completed in-place, the Updater checks the length of the posting and then sends a split job to *Local Rebuilder* if the length exceeds the split limit. The actual data deletions are performed asynchronously as a batch during local rebuild phase when the posting length exceeds the limit.

***Local Rebuilder*** is the key component to implement LIRE. It maintains a job queue for split, merge, and reassign jobs and dispatches jobs to multiple background threads for concurrent execution.

- A *split job* is triggered by *Updater* when a posting exceeds the split limit. It cleans deleted vectors in the oversized posting and splits it into small ones if needed.
- A *merge job* is triggered by the *Searcher* if it finds some postings are smaller than a minimum length threshold. It merges nearby undersized postings into a single one.
- A *reassign job* is triggered by a split or merge job, which rebalances the assignment of vectors in the nearby postings.

When the background split and merge jobs are complete, SPFresh will update the memory SPTAG index with the new posting centroids to replace the old one.

***Block Controller*** serves posting read, write, and append requests, as well as posting insertion and deletion operators on disk. It uses the raw block interface of SSD directly to avoid unnecessary read/write amplification incurred by some general storage engines, such as Log-structured-merge-tree-based KV store. Each posting may span multiple SSD blocks, each of which stores multiple vectors (including vector ID, version ID, and raw data). The *Block Controller* also maintains an in-memory mapping from the posting ID to its used SSD blocks as well as the free SSD blocks pool.

Next, we will discuss the design and implementation of *Local Rebuilder* (§4.2) and *Block Controller* (§4.3) in detail.

### 4.2 Local Rebuilder Design

In order to move split, merge, and re-assign jobs off the update critical path, SPFresh divides the update process into two parts, a foreground *Updater* and a background *Local Rebuilder*. These two components form a feed-forward pipeline, where *Updater* is the producer of requests to the *Local Rebuilder*. In this pipeline, the background *Local Rebuilder* is a key module that implements merge, split, and reassign operators of LIRE protocol efficiently to keep up with the foreground *Updater*.

#### 4.2.1 Rebuild Operators of LIRE protocol.

*Local Rebuilder* implements LIRE with three basic operators.

***Merge:*** To execute a merge job, the *Local Rebuilder* simply follows the merge protocol described in §3.2.

***Split:*** After receiving an oversized posting split job, the *Local Rebuilder* first garbage collects deleted vectors in the posting and verifies whether the posting length after garbage collection still exceeds the split limit. If not, the *Local Rebuilder* writes the garbage-collected posting back to storage and completes the split job.

Otherwise, a *balanced clustering process* is triggered to split the oversized posting into two smaller ones. In particular, *Local Rebuilder* leverages the multi-constraint balanced clustering algorithm in [67] to generate high-quality centroids and balanced postings.

After splitting, *Local Rebuilder* puts two new postings back to the index and deletes the original oversized postings.

***Reassign:*** A reassign job is generated by merge or split jobs. It checks if vectors in the new postings and/or their neighbors need to be relocated to re-balance the data distributions in the local region. The reassignment check is based on the two *necessary conditions* in §3.3. Note that neighbor posting check is *not* required for merge-triggered reassign.

Reassigning a vector without deleting its replicas in the unexamined postings increases the replica number. This not only increases storage overheads but also increases split and reassign frequency since the extraneous replicas take up spaces of postings. In order to efficiently identify stale vectors after reassignment without actual deletes, *Local Rebuilder* uses a version map to record *the version number* for each vector. A version number takes one byte and is stored in memory to record the version changes of a vector: seven bits for re-assign version and one bit for deletion label. When reassigning a vector, we increase its version number in the version map and append the raw vector data with its new version number to the target posting. All the old replicas with a stale version number are dropped during the search. The replicas will be garbage collected later.

#### 4.2.2 Concurrent Rebuild.

In SPFresh, *Local Rebuilder* is multi-threaded with efficient concurrency control of updates to the in-memory and on-disk data structures. Concurrent rebuild can avoid drops of index quality due to slow re-balance.

***Concurrency Control for Append/Split/Merge:*** Since append, split, and merge may update the same posting and the in-memory block mapping concurrently, We add a fine-grained posting-level write lock between these three operations to ensure a posting change is atomic.

Posting read does not require a lock. Therefore, identifying vectors for reassignment is lock-free since it only searches the index and checks the two necessary conditions. Our experiments show that even in a skewed workload, write lock contention is low, i.e., less than 1% contention cases. This is because only a small portion of postings are being edited concurrently.

During a reassign process, it is possible that a vector appends to a stale posting, which happened to be deleted concurrently. In such a case, we abort the reassignment and re-execute the reassign job for this vector. In our experiments, there are only less than 0.001% of total insertion requests encountering the posting-missing problem caused by split. As a result, the abort and re-execution overhead is minor.

***Concurrent Reassign:*** SPFresh avoids concurrently reassigning the same vector at the same time. When collecting vector candidates for reassignment, *Local Rebuilder* gathers the current version of the candidates. *Local Rebuilder* atomically executes reassignment operations by leveraging atomic primitives of compare-and-swapping (CAS) for the version

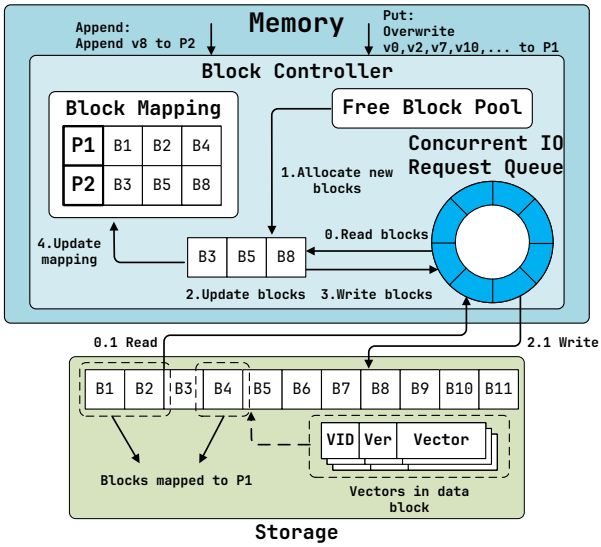


**Figure 6.** SPFresh storage overview.

number. If an atomic CAS operation fails on the vector version map, reassignment is aborted since the vector becomes a stale version. Otherwise, we let the corresponding reassign proceed to the end.

### 4.3 *Block Controller* Design

*Block Controller* is a light-weight storage engine highly optimized for reading. It offers *append-only* operation on postings. This design takes advantage of the characteristics of postings, where old posting data is immutable, and the update introduces no additional overheads for reading (unlike a log-structured file system, where multiple additional out-of-place reads are required). It keeps appending vector updates to a posting before exceeding a length limit. When a posting exceeds the length limit, the old posting is destroyed after being split into two new ones. To avoid unnecessary overheads in file systems or other storage engines (e.g., KV store), *Block Controller* operates directly on raw SSD interfaces.

***SPDK-based Implementation:*** Figure 6 overviews the storage design. *Block Controller* is implemented on top of SSD. It leverages the raw *block* interface offered by SPDK [25], a high-performance NVMe SSD library by Intel. SPDK offers a set of user-space IO libraries for accessing high-speed NVMe devices, which allow us to bypass the legacy storage stack to perform SSD I/Os directly.

***Storage Data Layout:*** As shown in Figure 6, a *Block Controller* consists of in-memory *Block Mapping*, a *Free Block Pool*, and a *Concurrent I/O Request Queue*.

*Block Mapping* maps a posting ID to its SSD block offsets. Since the posting ID is a continuous integer, block mapping is implemented as an in-memory dense array, where each element stores the block metadata of a posting length and its SSD block offsets. A posting consists of a list of tuples in the form of <vector id, version number, raw vector>, which typically takes three to four SSD blocks. A block mapping entry only consumes 40 bytes of memory. For one billion vectors, there only exist 0.1 billion postings. In this case, block mapping only consumes about 4GB of memory.

*Free Block Pool* maintains all free SSD blocks. It keeps track of the offsets of all the free blocks to serve disk allocation, and garbage collects stale blocks after spilt and reassign.

*Concurrent I/O Request Queue* is implemented using an SPDK circular buffer, which sends asynchronous read and write to SSD device for maximized IO throughput and low I/O latency.

***Posting API & Implementation:*** *Block Controller* provides a set of posting APIs as follows:

- `GET` retrieves posting data by the given ID. The request first looks up the block mapping to identify the corresponding SSD blocks. Asynchronous I/Os are then sent to the current I/O Request Queue. Later, all desired blocks are collected upon the completion of all I/Os.
- `ParallelGET` reads multiple postings in parallel to amortize the latency of individual `GET`s. This ensures fast search and update. `ParallelGET` allows sending a batch of I/O requests to fetch all the candidate postings, which hides the I/O latency and boosts disk utilization.
- `APPEND` adds a new vector to a posting's tail. Instead of read-modify-write at the posting-granularity, `APPEND` only involves read-modify-write of the last *block* of a posting, which reduces the amount of read/write amplification significantly. As shown in Figure 6, `APPEND` first allocates a new block, reads the original last block if the last block is not full, appends new values to the values from the last block, and then writes it as a new block. After a new block is written, it atomically updates the corresponding in-memory *Block Mapping* entry via a compare-and-swap operation to reflect the change. The old block will be released to the free block pool for later usage.
- `PUT` writes a new posting to SSD. Like `APPEND`, it allocates new blocks and writes for the entire posting blocks in bulk. Then it atomically updates the *Block Mapping* entry. If `PUT` overwrites an old posting, it releases old blocks to the *Free Block Pool*.

*Block Controller* provides a common abstraction and implementation, which can be generalized for other read-intensive applications (such as widely-used inverted index for search engine [2, 11]).

### 4.4 Crash Recovery

SPFresh adopts a simple crash recovery solution, which combines snapshot and write-ahead log (WAL). Specifically, an index snapshot is taken periodically, and all update requests between adjacent snapshots are collected into a WAL so that a crash can be recovered from the latest snapshot, followed

by replaying the WAL. The WAL will be deleted when a new snapshot is generated.

To take a snapshot for a vector index, we need to record both the in-memory and on-disk data structures. For in-memory index data, we create snapshots for centroid index, version maps in *Updater*, and block mapping and block pool in *Block Controller*, and flush the snapshots to disk. Snapshots are relatively cheap because these data structures take only 40GB for billion-level dataset, which costs 2~3 seconds for a full flush on a PCI-e based NVM SSD. For disk data, thanks to our block-level copy-on-write mechanism, we can collect all the released blocks during two snapshots into a *pre-release* buffer, which will be added to the *Free Block Pool* after the next snapshot is recorded. Thus all the data blocks modified in the interim can be rolled back to be consistent with the previous snapshot. This solution saves a large amount of disk space since we only delay the space release for old blocks during two snapshots.

## 5 Evaluation

In this section, we conduct experiments to answer the following questions:

- How does SPFresh compare with state-of-the-art baselines in terms of performances, search accuracy and resource usage? (§5.2)
- What is the maximum performance of SPFresh? (§5.3)
- Can SPFresh solve the data shifting problem illustrated in Figure 2? (§5.4)
- How to properly configure SPFresh? (§5.5)

### 5.1 Experimental Setup

***Platform:*** All experiments run on an Azure lsv3 [41] VM instance, which is a storage-optimized virtual machine with locally attached high-performance NVMe storage. In particular, we configured the VM with 16 vCPUs from a hyper-threaded Intel Xeon Platinum 8370C (Ice Lake) processor and 128GB memory for our experiments.

***Datasets:*** We use two widely-used vector datasets to evaluate SPFresh:

- SIFT1B [40] is a classical image vector dataset for evaluating the performance of ANNS algorithms that support large-scale vector search. It contains one billion of 128-dimensional byte vectors as the base set and 10,000 query vectors as the test set.
- SPACEV1B [1] is a dataset derived from production data from commercial search engines. It represents a different form of vector encoding: deep natural language encoding. It contains one billion of 100-dimensional byte vectors as a base set and 29,316 query vectors as the test set.

***Baselines:*** We compare SPFresh with two baselines:

- *DiskANN* is the state-of-the-art disk-based fresh ANNS system [53]. It is based on a graph ANNS index and uses an out-of-place update solution. We configure DiskANN with the same settings as in their paper [53]. For update configurations, DiskANN baseline processes *streamingMerge*, a lightweight global graph rebuild, for every new 30M vectors, where graph degree $R$ equals 64 and insert candidate list equals 75. For search configurations, DiskANN baseline uses the default setting with beamwidth equal to 2 and search candidate list L equal to 40 for recall10@10.
- *SPANN+*, a modified version of SPANN [67] which appends updates locally to a posting *without splitting and reassigning*. This is an *append-only* version of SPFresh without the *Local Rebuilder* module.

***Workloads:*** Three workloads are used in the experiments.

- *Workload A* simulates a realistic vector update scenario with 100 million scale of SPACEV vectors. The reason to reduce the scale from 1 billion to 100 million is that DiskANN requires several TBs of DRAM to run 1-billion scale fresh updates (shown in Table 1), which exceeds our machine's capacity. In particular, workload A simulates 1% update daily over 100 days. To generate updates realistically, we extract two disjoint SPACEV 100M datasets from SPACEV1B, where one is used as the base ANNS index data-set and the other as the update candidate pool. Each daily update epoch deletes 1% of vectors randomly from the base ANNS index, and inserts 1% of vectors randomly selected from the update data pool to the base index.
- *Workload B* has the same data scale and sampling method as Workload A but with a 100 million scale of the SIFT vector dataset.
- *Workload C* scales up our experiment data-set to be *billion-scale* using both the SIFT dataset and the SPACEV dataset. This workload aims at stress testing SPFresh, also with a 1% daily update rate.

***Metrics:*** SPFresh is designed for online ANNS streaming scenarios. Thus, our evaluation focuses on the following four categories of metrics.

- Search Performance: We measure tail (P90, P95, P99, and P99.9) latency and query per second (QPS) throughput. In particular, we have a hard cut of 10ms for SPFresh and all baselines, where the system finishes the result immediately and returns the current search results.
- Search Accuracy: We use the percentage of ground truths recalled by SPFresh system to measure accuracy.
- Update Performance: insertion and deletion throughputs.
- Resource Usages: the memory and CPU consumption.

### 5.2 Real-World Update Simulation

In this experiment, we compare all the metrics of SPFresh with all the baselines on the real-world situation. We use Workload A and B (see §5.1) to simulate 100 days of real-world updates, and we show that SPFresh outperforms baselines in all evaluation metrics.

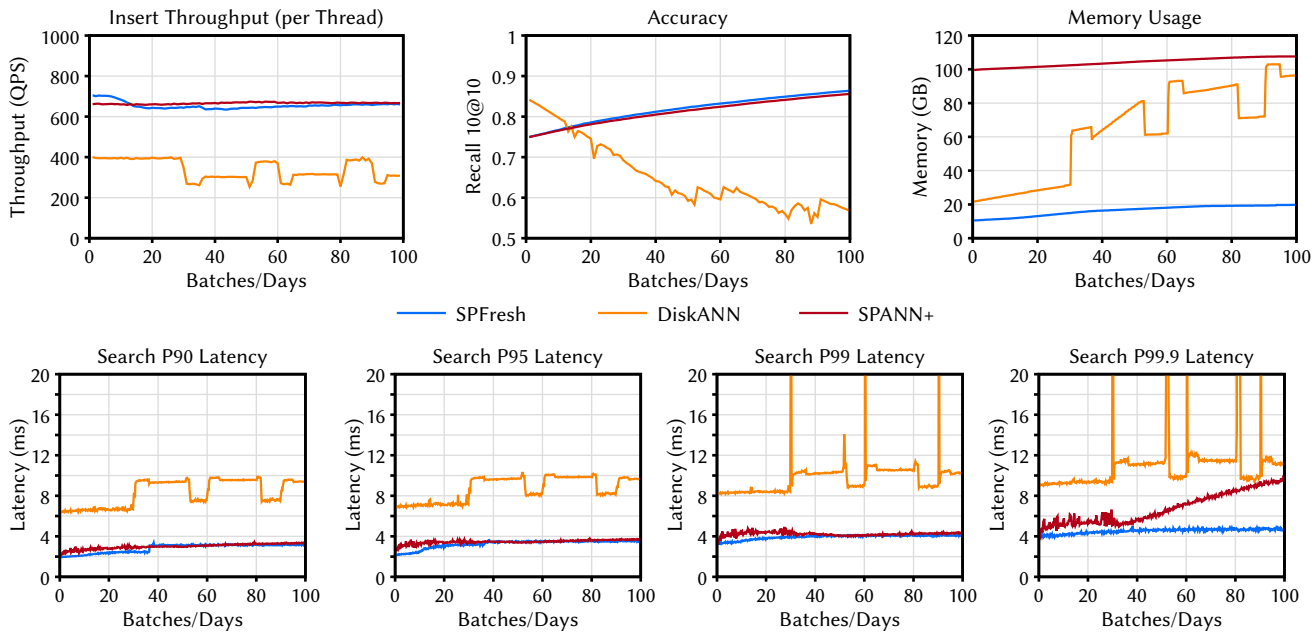


**Figure 7.** SPFresh overall performance (SPACEV: data distribution shifts over time).

| | DiskANN | SPANN+ | SPFresh |
|---|---|---|---|
| Insert | 3 | 1 | 1 |
| delete | 1 | 1 | 1 |
| Search | 2 | 2 | 2 |
| Background | 10 | 2 | 2 |
| Total | 16 | 6 | 6 |

**Table 2.** Threads allocation for overall performance.

**5.2.1 System Setup.** Table 2 lists the thread allocation for each system. Specifically, we allocate threads to each system's sub-components to meet the processing requirements of handling update throughput of 600~1200 QPS. The setting of update QPS is based on Alibaba's daily update speed (100 million each day) [69]. Each system only needs one thread to serve delete requests because deletion uses tombstones to record the deletions, which is lightweight. For DiskANN, due to its high insert latency, we set its number of insert threads to 3 and the number of background merge threads to 10, because it is the minimum to keep up with the update and garbage collection process. Further increasing the number will impact the query performance in the foreground negatively. The two remaining threads are used for foreground search for DiskANN.

To be comparable with DiskANN, SPFresh and SPANN+ also set 2 search threads. Both SPANN+ and SPFresh only need one insert thread to serve 600+ QPS insert throughput and two background threads for SPDK I/O and garbage collection (or local rebuilder).

**5.2.2 Experiment Results.** Figure 7 records a daily time series of the search tail latency, insert throughput per thread, search accuracy, and memory usage of Workload A. We can see that SPFresh achieves the best and the most stable performance on all the metrics during the 100 days.

***Low and Stable Search Tail Latency:*** Figure 7 shows that SPFresh achieves low and stable tail latency in all percentile measurements. Since the overall tail latency trends are similar, we focus our discussion on the most stringent tail latency measurement, P99.9.

Experiments indicate that LIRE is able to keep posting distribution uniform because SPFresh has a stable low search P99.9 latency around 4ms. In comparision, other systems' P99.9 latency is both worse and less stable than SPFresh.

DiskANN's P99.9 latency fluctuates significantly with a dramatic increase to more than 20ms during global rebuilds because a search thread could be blocked by a global rebuild even with 10ms hard latency cut. The search P99.9 latency of SPANN+ increases significantly from 4ms to more than 10ms because its posting keeps growing, inducing data skews and the increase of I/O and computation cost.

Overall, SPFresh maintains 2.41x lower P99.9 latency to DiskANN on average and expands its latency advantage to SPANN+. The low and stable search tail latency can be attributed to the LIRE protocol. During the experiment, we found that only 0.4% insertion will cause rebalancing. Among them, the average split number is 2, and the maximum split

number is 160, with a cascading length of 3. The merge frequency is only 0.1% of the update (insertion and deletion). On average, each time 5094 vectors are evaluated, and only 79 are actually reassigned.

***High Search Accuracy:*** SPFresh achieves a higher search accuracy compared to baselines. Both SPANN+ and SPFresh would not violate the NPA of a cluster-based index. Therefore, the accuracy of SPANN+ and SPFresh grows gradually since newly inserted vectors are all assigned to a subset of postings due to the data shifting. Therefore, queries to these new vectors can easily hit since search and insertion follow the same search path to get the nearest postings.

Although SPANN+ search accuracy increases in a similar trend like SPFresh, the gap in accuracy increases over time. The increasing gap between these two systems is because the index quality of SPANN+ degrades as partition distributions skew over time.

DiskANN proposes an algorithm to reduce the overhead of global rebuild by eliminating outdated edges from all vertices and populating edges for a new vertex using the neighborhood of its deleted neighbors. This method aims to reduce the decline in accuracy due to a decreased number of edges caused by vector deletions without reconstructing its graph-based index completely. However, experiments show that such a method cannot prevent DiskANN's search accuracy from decreasing over time.

***High and Stable Update Performance:*** SPFresh achieves 1.5ms average insert latency and stable tail latency. On the other hand, DiskANN suffers from the heavy computation caused by in-memory graph traversal, and thus results in higher latency and lower throughput. Compared to SPANN+, we can see that SPFresh's lightweight *Local Rebuilder* will not affect the foreground insert performance.

***Low Resource Utilization:*** For resource usage, SPFresh achieves as low as 5.30X lower memory usage than baselines during the whole update process. DiskANN occupies an extra 60G memories for background streamingMerge and 15GB for the second in-memory index for the update. SPANN+ needs much larger block-mapping entries to allow a larger posting length. SPFresh keeps the memory under 20GB, which only grows slightly over time because new metadata are created for each new posting triggered by splits. SPFresh also maintains a reasonable disk size. In the index, we find that 86% of the total vectors have more than one replica, and on average, one vector has 5.47 replicas, which is similar to the index built statically.

We also ran the same experiment on Workload B and reached a similar conclusion with DiskANN. Note that SPANN+ achieves similar performance with SPFresh on the SIFT dataset, which is almost uniformly distributed. This is expected because background garbage collection should be able to prune stale vectors on SPANN+ without splits on uniform data-set. Consequently, SPANN+ achieves a similar index quality as that of SPFresh because its posting distribution does not shift much.

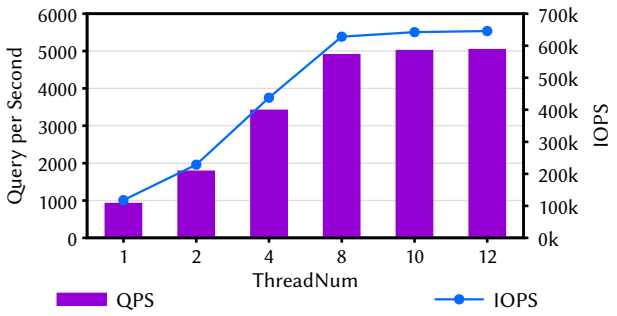


**Figure 8.** Search throughput/disk IOPS vs # of SPFresh search threads on Azure lsv instance [41].

| | #threads | | #threads |
|---|---|---|---|
| Delete/Re-insert | 4 | Search | 8 |
| Background | 3 | Total | 15 |

**Table 3.** Thread allocation for SPFresh in billion-scale tests.

### 5.3 Billion-Scale Stress Test

We scale up vector data size to billion-level and configure the system to show the best performance SPFresh achieves with the given resource. We use Workload C (see §5.1) to simulate a 20-day real-world update scenario. We demonstrate that SPFresh has fully saturated SSD's bandwidth and performed well with stable resource utilization.

**5.3.1 System Setup.** Table 3 lists thread allocation for SPFresh's stress tests. To fully achieve the IOPS of SSD, our setting maximizes search throughput while supporting maximum update throughput.

Azure lsv3 has a max *guaranteed* NVMe IOPS of 400K [41]. We first run an experiment to find out the max search throughput lsv3 can handle. As Figure 8 shows, the IOPS and search throughput almost reach the peak at 8 search threads on a single SSD disk.

When search thread count is set to 8 for max search throughput, a fore-ground thread count of 4 saturates the update throughput. Therefore we set the thread counts as in Table 3 for this experiment. Search is the most important part of ANNS service, so the stress test we will maximize our search throughput while support the maximum update throughput. Since Azure document[41] note that the max NVMe disk throughput is 400K and can go higher but not be guaranteed to keep the IOPS higher than 400K, we make a simple test on lsv3 NVMe SSD by running SPFresh Search On Azure lsv3 to find out that the MAX performance of lsv3 NVMe SSD. and we can see on figure 8 that when we set

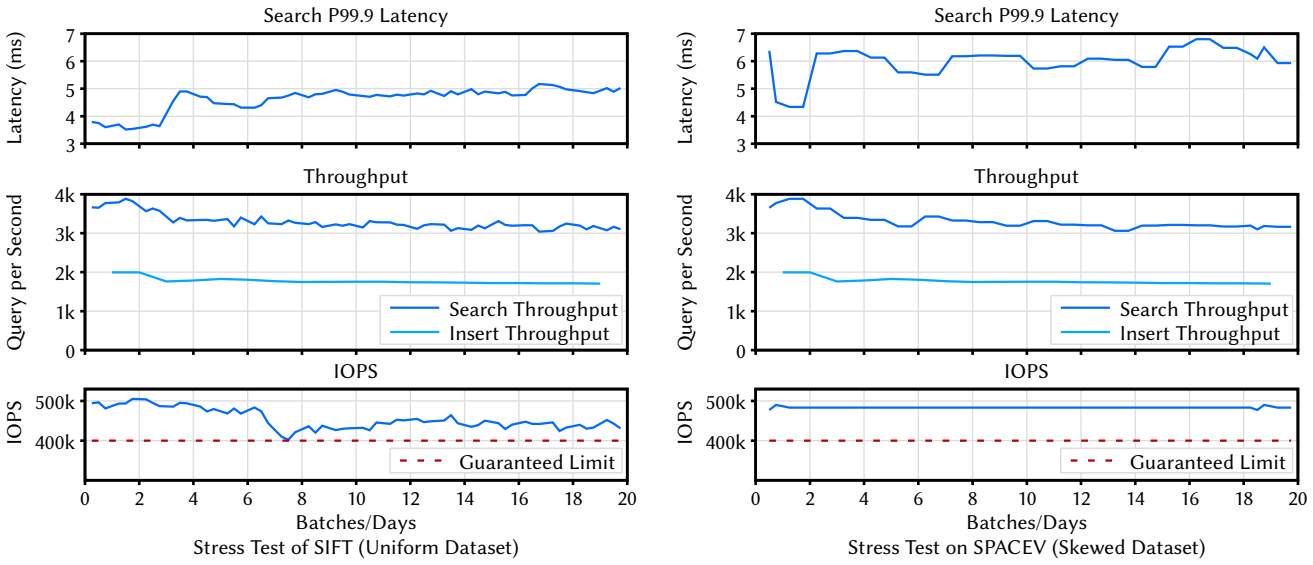


**Figure 9.** Billion-scale stress test on uniform and skew datasets. SPFresh saturates the I/O with a stable P99.9 latency while keeping accuracy stable and higher than 0.862 for uniform dataset and 0.807 for skew dataset by searching nearest 64 postings, the memory and CPU utilization keep in about 74 GB and 1300% in both uniform and skew datasets.

search thread to 8, the QPS of SPFresh and the IOPS of Disk will no more grows, so in stress test we will set search thread to 8 and then maximize the update throughput, and we find out that when fore-ground thread set to 4 the IOPS will reach the peak during the update.

#### 5.3.2 Experiment Results.

Figure 9 records a daily time series of the search P99.9 latency on both uniform and skew datasets, search/insert throughput, and the IOPS of Work-Load C. We can see that SPFresh reaches the IOPS limitation with stable performance and resource utilization throughout the entire run.

***High NVMe SSD IOPS Utilization:*** As we can see from Figure 9, SPFresh always fully utilizes the NVMe's bandwidth, even exceeding the max *guaranteed* IOPS of Azurelsv3. Thanks to LIRE's lightweight protocol, we can see SPFresh's bottleneck is in the disk IOPS, before reaching the CPU and memory resource limit.

***Stable Search and Update Performance:*** As the data scale grows from 100M to 1B, the search latency is stable, just like that in §5.2. There is some slight increase of P99.9 latency in the beginning when the first split jobs are triggered. In this case, the P99.9 latency increases slightly because of the gradual growth of in-memory index size, which makes the in-memory computation more costly over time.

***Stable Accuracy:*** During the entire stress test, the accuracy of SPFresh remains stable, which is higher than 0.862 for the uniform dataset and 0.807 for the skew dataset by searching the nearest 64 postings.

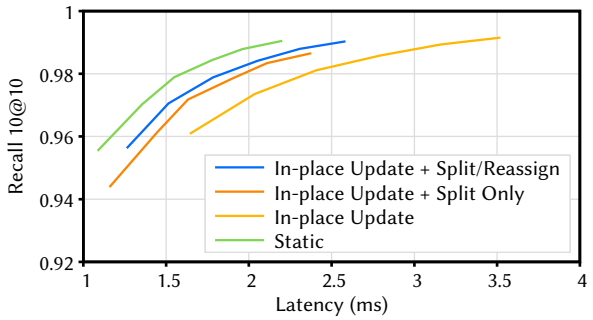


**Figure 10.** The trade-off on search accuracy and latency with various update techniques under an increasingly skewed data distribution.

### 5.4 Data Distribution Shifting Micro-benchmark

In this experiment, we replay the experiment in §2.3 to demonstrate that LIRE is required to re-balance the shifting data distribution. We compare four systems in this experiment, where *Static* is our target since it has no updates. For the rest of the three systems, we start with a naive system with in-place update only, i.e., SPANN+, and gradually add sub-components of LIRE into the system.

***Experiment Results:*** Figure 10 shows the recall and latency trade-off result. With a relaxed search latency, the figure shows that recall improves for all four systems. Meanwhile, as the curve moves northwest, the system shows a higher ANNS index quality with a more accurate recall and a lower latency. An in-place update-only solution may have a high

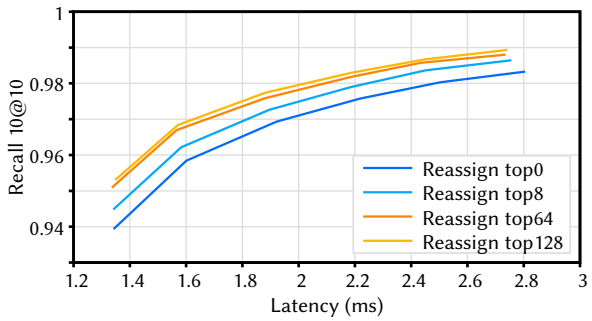


**Figure 11.** Parameter Study: Reassign Range, top64 (nearest 64 postings) is enough for reassign scanning.

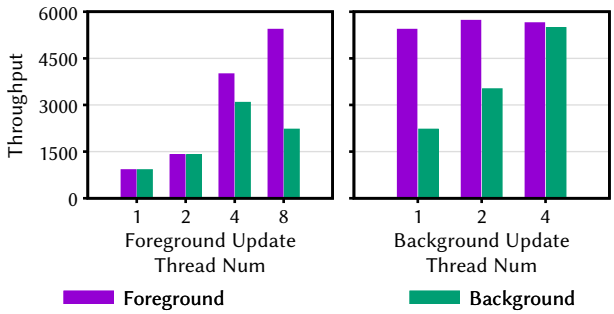


**Figure 12.** Foreground Scalability (Background Thread Number = 1) and Background Scalability (Foreground Thread Number = 8).

recall but at the expense of high latency. Adding a split component into the in-place update decreases the search latency with the same accuracy. Adding the reassignment component further decreases the search latency. As shown in Figure 10, the performance of SPFresh with in-place update + split/reassign is the closest one to the Static's results, which represent the ideal cases.

### 5.5 Parameter Study

In this experiment, we investigate the proper parameter configurations for SPFresh to achieve maximum performance. Experiment results show that the Reassignment only requires checking a limited scope of proximate postings for scanning to attain a good index quality. Furthermore, SPFresh demands a minimal increase in computational resources, specifically in terms of threads, to accomplish high throughput while also demonstrating good scalability.

***Reassign Range:*** The first parameter we examine is the reassign range, i.e., the size of the local rebuild range. Reassign range is measured by the number of nearby postings to check for vector reassignment after a new posting list is created. In this experiment, we use the same setting as in §2.3.

In Figure 11, we vary reassign range from the nearest 0, i.e., only process reassign in the split posting, to the nearest 128 postings. As the reassign range increases, accuracy also increases with the same search time budget because more NPA-violating vectors are identified and reassigned. The accuracy increase rate wanes off as the reassign range increases, where there is only a marginal increase from range 64 to 128. Consequently, we chose 64 as SPFresh's default reassign range.

***Fore/Back-ground Update Resource Balance:*** The foreground *In-place Updater* and background *Local Rebuilder* work as a feed-forward pipeline as detailed in §4.2. In this experiment, we examine the proper resource ratio for *In-place Updater* and *Local Rebuilder* to make their processing speed balanced in the pipeline. Specifically, we configure the foreground and background threads and measure the throughput to see when the update resource is balanced.

As the left part of Figure 12 shows, a single-threaded background *Local Rebuilder* can keep up the foreground *In-place Updater* until foreground threads are set to 2. Similarly, as on the right side of Figure 12, an 8-threaded foreground *In-place Updater* needs at least four threads for foreground *In-place Updater* to generate enough requests for the *Local Rebuilder*. Based on the result, SPFresh sets a thread ratio of 2:1 between the foreground *In-place Updater* and the background *Local Rebuilder* balances the feed-forward pipeline.

## 6 Conclusion

SPFresh supports incremental in-place update for billion-scale vector search. It implements LIRE, a Lightweight Incremental RE-balancing protocol to *split* overly large postings and *reassign* vectors across neighboring postings when necessary. Experiments show that SPFresh can incorporate continuous updates faster with significantly lower resources than existing solutions while maintaining high search recalls by (1) LIRE identifies a minimal set of neighborhood vectors in the large index space for updating to adapt to data distribution shift; (2) the index re-balancing operations and the foreground queries are decoupled, and handled by efficient concurrency control mechanisms, avoiding operation interference. SPFresh's solid single-node performance builds a strong foundation for the future distributed version.

## Acknowledgments

We thank all the anonymous reviewers for their insightful feedback, and our shepherd, Nitin Agrawal, for his guidance during the preparation of our camera-ready submission. This work is supported in part by the National Natural Science Foundation of China under Grant No.: 62141216, 62172382 and 61832011, and the University Synergy Innovation Program of Anhui Province under Grant No.: GXXT-2022-045. Cheng Li and Qi Chen are the corresponding authors.

## References


[1] Hervé Jégou. Romain Tavenard. Matthijs Douze. Laurent Amsaleg. 2011. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/.

[2] Apache. [n. d.]. Apache Lucene is a high-performance, full-featured text search engine library written in Java. https://github.com/apache/lucene.

[3] 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. *Proc. VLDB Endow.* 11, 8 (apr 2018), 906–919. https://doi.org/10.14778/3204028.3204034

[4] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM* 45, 6 (1998), 891–923. https://doi.org/10.1145/293347.293348

[5] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* 37, 6 (2014), 1247–1260.

[6] Artem Babenko and Victor Lempitsky. 2017. Product Split Trees. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6316–6324. https://doi.org/10.1109/CVPR.2017.669

[7] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer-Verlag, Berlin, Heidelberg, 202–216.

[8] J.S. Beis and D.G. Lowe. 1997. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *1997 Conference on Computer Vision and Pattern Recognition {CVPR}'97, June 17-19, 1997, San Juan, Puerto Rico*. IEEE Computer Society, USA, 1000–1006. https://doi.org/10.1109/CVPR.1997.609451

[9] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (sep 1975), 509–517.

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

[11] Elasticsearch B.V. [n. d.]. Elasticsearch. https://www.elastic.co/.

[12] Kenneth L Clarkson. 1994. An algorithm for approximate closest-point queries. In *Proceedings of the tenth annual symposium on Computational geometry*. Association for Computing Machinery, New York, NY, USA, 160–164.

[13] Sanjoy Dasgupta and Yoav Freund. 2008. Random Projection Trees and Low Dimensional Manifolds. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing* (Victoria, British Columbia, Canada) *(STOC '08)*. Association for Computing Machinery, New York, NY, USA, 537–546. https://doi.org/10.1145/1374376.1374452

[14] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (Brooklyn, New York, USA) *(SCG '04)*. Association for Computing Machinery, New York, NY, USA, 253–262.

[15] B. N. Delaunay. 1934. Sur la sphère vide. *Bull. Acad. Sci. URSS* 1934, 6 (1934), 793–800.

[16] Wei Dong, Moses Charikar, 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, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*. Association for Computing Machinery, New York, NY, USA, 577–586. https://doi.org/10.1145/1963405.1963487

[17] Facebook. 2020. Faiss. https://github.com/facebookresearch/faiss.

[18] Jerome H. Freidman, Jon Louis Bentley, and Raphael Ari Finkel. 1977. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Software* 3, 3 (1977), 209–226. https://doi.org/10.1145/355744.355745

[19] 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.

[20] K Ruben Gabriel and Robert R Sokal. 1969. A new statistical approach to geographic variation analysis. *Systematic zoology* 18, 3 (1969), 259–278.

[21] Google. [n. d.]. Youtube. https://blog.youtube/press/.

[22] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: a cloud native vector database management system. *Proceedings of the VLDB Endowment* 15 (08 2022), 3548–3561. https://doi.org/10.14778/3554821.3554843

[23] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast Approximate Nearest-Neighbor Search with k-Nearest Neighbor Graph. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. AAAI Press, 1312–1317. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-222

[24] Junfeng He, Wei Liu, and Shih-Fu Chang. 2010. Scalable Similarity Search with Optimized Kernel Hashing. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, DC, USA) *(KDD '10)*. Association for Computing Machinery, New York, NY, USA, 1129–1138. https://doi.org/10.1145/1835804.1835946

[25] Intel. [n. d.]. SPDK: Storage Performance Development Kit. https://spdk.io/.

[26] Masajiro Iwasaki. 2016. Pruned bi-directed k-nearest neighbor graph for proximity search. In *International Conference on Similarity Search and Applications*. Springer, Springer International Publishing, Cham, 20–33.

[27] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. arXiv:1810.07355 [cs.DB]

[28] P. Jain, B. Kulis, and K. Grauman. 2008. Fast image search for learned metrics. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*. 1–8.

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

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

[31] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFboost: Using Accelerated Flash Storage for External Graph Analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 411–424. https://doi.org/10.1109/ISCA.2018.00042

[32] Brian Kulis and Kristen Grauman. 2009. Kernelized locality-sensitive hashing for scalable image search. In *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, Elsevier Science Publishers B. V., NLD, 2130–2137.

[33] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 31–46.

[34] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The Design and Implementation of a Real Time Visual Search System on JD E-Commerce Platform. In *Proceedings of the 19th International Middleware Conference Industry* (Rennes, France) *(Middleware '18)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/3284028.3284030

[35] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. 2021. Embedding-Based Product Retrieval in Taobao Search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD '21)*. Association for Computing Machinery, New York, NY, USA, 3181–3189. https://doi.org/10.1145/3447548.3467101

[36] Ting Liu, Andrew W Moore, Alexander Gray, and Ke Yang. 2004. An investigation of practical approximate nearest neighbor algorithms. *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, {NIPS} 2004, December 13-18, 2004, Vancouver, British Columbia, Canada]* (2004), 825–832. http://papers.nips.cc/paper/2666-an-investigation-of-practical-approximate-nearest-neighbor-algorithms

[37] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[38] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[39] Mark McCartin-Lim, Andrew McGregor, and Rui Wang. 2012. Approximate Principal Direction Trees. In *Proceedings of the 29th International Coference on International Conference on Machine Learning* (Edinburgh, Scotland) *(ICML'12)*. Omnipress, Madison, WI, USA, 1611–1618.

[40] microsoft. 2020. SPACEV1B: A billion-Scale vector dataset for text descriptors. https://github.com/microsoft/SPTAG/tree/master/datasets/SPACEV1B.

[41] microsoft. 2023. lsv3-series. https://learn.microsoft.com/en-us/azure/virtual-machines/lsv3-series.

[42] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL]

[43] Andrew W Moore. 2000. The Anchors Hierarchy: Using the Triangle Inequality to Survive High Dimensional Data. In *{UAI} '00: Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence, Stanford University, Stanford, California, USA, June 30 - July 3, 2000*, Vol. I. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 397–405. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1{&}smnu=2{&}article{_}id=47{&}proceeding{_}id=16

[44] Yadong Mu and Shuicheng Yan. 2010. Non-Metric Locality-Sensitive Hashing.. In *AAAI*. AAAI Press, 539–544.

[45] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbour Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (2014), 2227–2240. https://doi.org/10.1109/TPAMI.2014.2321376

[46] David Nistér and Henrik Stewénius. 2006. Scalable recognition with a vocabulary tree. *2006 {IEEE} Computer Society Conference on Computer Vision and Pattern Recognition {(CVPR} 2006), 17-22 June 2006, New York, NY, {USA}* 2 (2006), 2161–2168. https://doi.org/10.1109/CVPR.2006.264

[47] OpenAI. 2022. ChatGPT Retrieval Plugin. https://github.com/openai/chatgpt-retrieval-plugin.

[48] OpenAI. 2022. GPT3 Embedding. https://platform.openai.com/docs/guides/embeddings.

[49] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global Vectors for Word Representation. *EMNLP* 14, 1532–1543. https://doi.org/10.3115/v1/D14-1162

[50] Maxim Raginsky and Svetlana Lazebnik. 2009. Locality-sensitive binary codes from shift-invariant kernels. In *Advances in neural information processing systems*. Curran Associates Inc., Red Hook, NY, USA, 1509–1517.

[51] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 895, 13 pages.

[52] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740

[53] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. arXiv:2105.09613 [cs.IR]

[54] Jingkuan Song, Yi Yang, Zi Huang, Heng Tao Shen, and Richang Hong. 2011. Multiple Feature Hashing for Real-Time Large Scale near-Duplicate Video Retrieval. In *Proceedings of the 19th ACM International Conference on Multimedia* (Scottsdale, Arizona, USA) *(MM '11)*. Association for Computing Machinery, New York, NY, USA, 423–432. https://doi.org/10.1145/2072298.2072354

[55] Robert F. Sproull. 1991. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica* 6, 1-6 (1991), 579–589. https://doi.org/10.1007/BF01759061

[56] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. *DiskANN: Fast Accurate Billion-Point Nearest Neighbor Search on a Single Node*. Curran Associates Inc., Red Hook, NY, USA.

[57] Jan Suchal and Pavol Navrat. 2010. Full Text Search Engine as Scalable k-Nearest Neighbor Recommendation System. In *International Conference on Artificial Intelligence in Theory and Practice*, Vol. 331. 165–173. https://doi.org/10.1007/978-3-642-15286-3_16

[58] 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 (sep 2014), 1–12. https://doi.org/10.14778/2735461.2735462

[59] 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 (sep 2013), 1930–1941. https://doi.org/10.14778/2556549.2556574

[60] Godfried T Toussaint. 1980. The relative neighbourhood graph of a finite planar set. *Pattern recognition* 12, 4 (1980), 261–268.

[61] Jun Wang, Sanjiv Kumar, and S. Chang. 2012. Semi-Supervised Hashing for Large-Scale Search. *IEEE transactions on pattern analysis and machine intelligence* 34 (02 2012). https://doi.org/10.1109/TPAMI.2012.48

[62] Jingdong Wang and Shipeng Li. 2012. Query-Driven Iterated Neighborhood Graph Search for Large Scale Indexing. In *Proceedings of the 20th ACM International Conference on Multimedia* (Nara, Japan) *(MM '12)*. Association for Computing Machinery, New York, NY, USA, 179–188. https://doi.org/10.1145/2393347.2393378

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

[64] Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian Sheng Hua. 2014. Trinary-projection trees for approximate nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 2 (2014), 388–403. https://doi.org/10.1109/TPAMI.2013.125

[65] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing

Machinery, New York, NY, USA, 2614–2627. https://doi.org/10.1145/3448016.3457550

[66] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. 2018. A Survey on Learning to Hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 4 (2018), 769–790.

[67] Qi Chen. Bing Zhao. Haidong Wang. Mingqin Li. Chuanjie Liu. Zengzhong Li. Mao Yang. Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*.

[68] Qi Chen. Haidong Wang. Mingqin Li. Gang Ren. Scarlett Li. Jeffery Zhu. Jason Li. Chuanjie Liu. Lintao Zhang. Jingdong Wang. 2018. SPTAG: A library for fast approximate nearest neighbor search. https://github.com/Microsoft/SPTAG.

[69] 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 (aug 2020), 3152–3165. https://doi.org/10.14778/3415478.3415541

[70] Yair Weiss, Antonio Torralba, and Rob Fergus. 2009. Spectral hashing. In *Advances in neural information processing systems*. Curran Associates Inc., Red Hook, NY, USA, 1753–1760.

[71] Hao Xu, Jingdong Wang, Zhu Li, Gang Zeng, Shipeng Li, and Nenghai Yu. 2011. Complementary hashing for approximate nearest neighbor search. In *2011 International Conference on Computer Vision*. IEEE Computer Society, USA, 1631–1638. https://doi.org/10.1109/ICCV.2011.6126424

[72] Peter N Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. *Proceedings of the Fourth Annual {ACM/SIGACT-SIAM} Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas.* (1993), 311–321. http://dl.acm.org/citation.cfm?id=313559.313789

[73] Minjia Zhang and Yuxiong He. 2019. GRIP: Multi-Store Capacity-Optimized High-Performance Nearest Neighbor Search for Vector Search Engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*, Wenwu Zhu 0001, Dacheng Tao, Xueqi Cheng, Peng Cui 0001, Elke A. Rundensteiner, David Carmel, Qi He, and Jeffrey Xu Yu (Eds.). Association for Computing Machinery, New York, NY, USA, 1673–1682. https://doi.org/10.1145/3357384.3357938

[74] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 45–58.