

Compass: General Filtered Search across Vector and Structured Data

Chunxiao Ye

The Chinese University of Hong Kong
Hong Kong, China

Xiao Yan

Wuhan University
Wuhan, China

Eric Lo

The Chinese University of Hong Kong
Hong Kong, China

Abstract—The increasing prevalence of hybrid vector and relational data necessitates efficient, general support for queries that combine high-dimensional vector search with complex relational filtering. However, existing filtered search solutions are fundamentally limited by specialized indices, which restrict arbitrary filtering and hinder integration with general-purpose DBMSs. This work introduces COMPASS, a unified framework that enables general filtered search across vector and structured data without relying on new index designs. Compass leverages established index structures – such as HNSW and IVF for vector attributes, and B+-trees for relational attributes – implementing a principled cooperative query execution strategy that coordinates candidate generation and predicate evaluation across modalities. Uniquely, Compass maintains generality by allowing arbitrary conjunctions, disjunctions, and range predicates, while ensuring robustness even with highly-selective or multi-attribute filters. Comprehensive empirical evaluations demonstrate that Compass consistently outperforms NaviX, the only existing performant general framework, across diverse hybrid query workloads. It also matches the query throughput of specialized single-attribute indices in their favorite settings with only a single attribute involved, all while maintaining full generality and DBMS compatibility. Overall, Compass offers a practical and robust solution for achieving truly general filtered search in vector database systems.

Index Terms—filtered vector similarity search

I. INTRODUCTION

The rapid proliferation of unstructured data—including images, videos, and free-form documents—has precipitated the rise of *vector databases*, wherein high-dimensional embeddings enable semantic retrieval via approximate k -nearest neighbor (A- k NN) search. These systems mark a significant shift in query capabilities, allowing similarity-based access beyond simple exact matches. However, practical workloads increasingly demand queries that jointly reason over semantic similarity and structured relational predicates: for example, retrieving “products similar to a reference item but priced below \$100,” or “images analogous to a query example but timestamped after 2020.” Addressing such requirements necessitates *filtered search*, integrating vector-search and attribute-filtering in the same query.

Despite recent efforts, most existing filtered search solutions [1]–[6] remain ad-hoc and fragile under *general filtering conditions*. The majority design specialized indices that tightly couple the vector embedding with one designated relational attribute, delivering high efficiency for specific, fixed filter types. Yet such approaches are fundamentally limited. They

cannot support general relational filtering—encompassing numeric range predicates, multi-attribute queries, and complex conjunctions or disjunctions—unless ad-hoc pre- and post-filtering steps are introduced. As a result, their performance degrades when handling multiple attributes, or varied predicate combinations. Moreover, each index must pre-select the target relational attribute during index build time, resulting in one specialized index per relational attribute — a solution that is neither scalable nor space-efficient.

Within published literature, ACORN [2] and NaviX [7] are the notable exceptions, distinguished by its generality and seamless integration with database management systems. They decouple vector and relational indexing, upholding compatibility with general query processing. Nevertheless, they are hindered by a core limitation: relational filters disrupt the traversal connectivity of graph-based vector indices such as HNSW [8], as many neighbors are pruned by predicate evaluation. To compensate, they expand traversal to explore beyond the immediate neighbors, regaining coverage but paying the price in overhead and reduced query throughput (QPS).

This paper introduces COMPASS, a versatile filtered search framework that seamlessly integrates efficiency, robustness, and compatibility with DBMS. Rather than creating new specialized indices, Compass leverages established indices, such as HNSW and IVF for vector attributes, and B+-trees or even learned indices [9], [10] for relational attributes, which are all already battle-tested and adopted by industrial products. The key innovation of Compass is its *shared candidate queue*, which facilitates cooperative query execution across these indices. The vector index operates mostly as usual, while the system dynamically supplements candidates from relational indices that meet the necessary filters when required. This architecture enables Compass to efficiently expand the search space while rigorously enforcing relational constraints, all without compromising generality or ease of integration.

Empirical results demonstrate that Compass consistently outperforms NaviX across a wide range of query patterns, including single- and multi-attribute filters, varying selectivities, and both conjunctions and disjunctions. Remarkably, Compass achieves throughput comparable to that of specialized single-attribute indices even in scenarios that favor such indices, involving only one relational attribute. This is accomplished while maintaining full generality and leveraging proven database indexing components.

TABLE I
A SUMMARY OF EXISTING METHODS FOR FILTERED VECTOR SEARCH AND COMPARISON WITH OUR COMPASS.

Method	Discrete Attribute Support				Continuous Attribute Support			Index Structure Property		
	Equality	Comparison	Conjunction	Disjunction	Range	Conjunction	Disjunction	Insertion	Index Size	Build Time
FilteredDiskANN	✓	×	×	✓	×	×	×	✓	Moderate	Normal
iRangeGraph	✓	✓	□	□	✓	□	□	×	Large	Long
DSG	✓	✓	□	□	✓	□	□	✓	Large	Long
SeRF	✓	✓	□	□	✓	□	□	×	Moderate	Long
ACORN	✓	✓	✓	✓	✓	✓	✓	✓	Moderate	Moderate
NaviX	✓	✓	✓	✓	✓	✓	✓	✓	Normal	Normal
Compass	✓	✓	✓	✓	✓	✓	✓	✓	Normal	Normal

✓: Full Support □: Partial Support ×: No Support

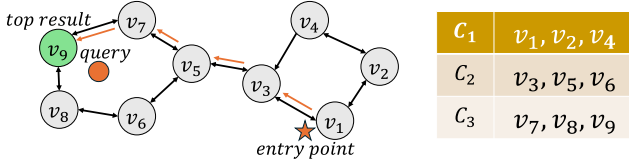


Fig. 1. An illustration of proximity graph (left) and IVF index (right).

II. PRELIMINARIES

A. Problem Definitions

Definition 1. General Filtered Search. Let $\mathcal{D} = \{(v_i, a_i) \mid v_i \in \mathbb{R}^d, a_i \in \mathcal{A}\}$ be a dataset where each record consists of a vector representation $v_i \in \mathbb{R}^d$ and a tuple of relational attributes a_i defined over schema $\mathcal{A}$. A filtered query is defined as $Q = (q, p)$, where $q \in \mathbb{R}^d$ is the query vector and $p : \mathcal{A} \rightarrow \{\text{true}, \text{false}\}$ is a Boolean predicate over the attributes in $\mathcal{A}$, composed of conjunctions and/or disjunctions of attribute conditions. Let $\delta : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ be a distance function. Then, the general filtered search (GFS) problem is to find the set

$$S_k^* = \text{Top-}k\{(v_i, a_i) \in \mathcal{D}' \text{ ranked by } \delta(q, v_i)\}.$$

where

$$\mathcal{D}' = \{(v_i, a_i) \in \mathcal{D} \mid p(a_i) = \text{true}\}$$

is the subset of records that satisfy the predicate p .

Same as all previous work, since exact vector search requires $O(n)$ time, we tackle the approximate GFS.

Definition 2. Approximate GFS returns a set $S_k \subseteq \{(v, a) \in \mathcal{D} \mid p(a) = \text{true}\}$, $|S_k| = k$, and the quality of the approximate result set is measured by recall R

$$R = \frac{|S_k \cap S_k^*|}{|S_k^*|}. \quad (1)$$

Our goal is to reach a high recall (e.g., 0.85 or 0.95) with a short query processing time or a small number of vector distance computations.

B. Indices for Vector Search

Two indices are the most popular for vector search, i.e., proximity graph [8], [11]–[13] and inverted file (IVF) [14]–[17]. We provide an illustration of them in Figure 1.

In a proximity graph index, the nodes are the vectors and edges connect similar vectors. Vector search is conducted by a best-first-search-style graph traversal, which starts with a random or fixed entry node and uses a candidate queue to manage the node to visit and a top queue to manage nodes visited. When visiting a node, the graph traversal computes the distances between the query and all neighbors of the node and adds these neighbors to the candidate queue; then, an unvisited node with the smallest distance is selected from the candidate queue as the next to visit. Both search cost and result quality is controlled by the size of the top queue (usually denoted as *efs*), with a larger queue size leading to more distance computations but higher result quality. There are many variants of the proximity graph index, e.g., HNSW [8], NSG [11], Vamanna [12], and SSG [13]; they mainly differ in the entry point selection and edge pruning rule; while the graph traversal procedure for query processing is similar.

The Inverted File (IVF) index partitions a dataset’s vectors into clusters (e.g. via K-means), with each cluster represented by a centroid. During query processing, all centroids are initially evaluated. Subsequently, only vectors within the top-ranking clusters (i.e., those with the nearest centroids) are inspected as candidate results. While proximity graphs generally exhibit higher search efficiency—requiring fewer distance computations for a given recall rate—IVF offers the advantages of larger data access granularity and more regular data access patterns, rendering it highly amenable to parallelization.

III. RELATED WORK

In this section, we give an overview of work related to general filtered search. Table I gives a summary and we elaborate them as follows.

A. Specialized Indices for Label Filtering

Early work on filtered search focused exclusively on the equality comparison of *discrete* attributes, typically referred to as label filtering [1], [18], [19]. Among them, [1] and [18] fuse the relational attributes into the vector to build the graph on the fused representation. FilteredDiskANN [19] extends the graph-based ANN structure by applying a label-aware pruning strategy—which ensures path navigability for filtered queries.

At query time, FilteredDiskANN dynamically maintains a priority queue of candidates by iteratively adding only those neighboring nodes that satisfy the query’s label filter predicate.

B. Specialized Indices for 1D Numerical Filtering

Recent advances [3]–[6], [20]–[23] extend to support continuous attribute but limits the number of attribute to one. Particularly, Super-Post-filtering [4] proposes partitioning the relational attribute domain using segment tree and building separate graph index for each segment. iRangeGraph [5] streamlines such index construction by dynamically composing only the subgraphs relevant to a given query range. Both Super-Post-filtering and iRangeGraph, however, require maintaining separate graph structures for each segment. As attribute cardinality or range granularity increases, the storage overhead becomes substantial, making these methods impractical for scenarios involving large attribute domains or fine-grained filtering requirements. In our experiments, they typically resulted in an index size of up to $6\times$ the original index. Other partitioning methods either face the similar problem of bloated index size [21]–[23] or introduce statically-maintained partitions subject to query workload change [4], [20].

SeRF [3] compresses multiple segment-specific graph indices into a single unified structure, resulting in a more practical overall index size. However, this compactness is achieved by constructing the index according to the sorted order of a chosen relational attribute, inherently limiting support for dynamic vector insertions. Furthermore, since SeRF is not inherently designed to support general filtered search, leveraging it for multi-attribute filtering requires constructing a separate vector-integrated index for each relational attribute. For example, consider a schema $\mathcal{A}$ with four attributes a_1 , a_2 , a_3 , and a_4 . If we construct a SeRF for attribute a_1 , it will only be able to serve queries that impose a predicate p on a_1 . That is, it cannot serve other predicates such as $a_2 \wedge a_3$. In order to serve predicates on *any* relational attributes on $\mathcal{A}$, we would need to build four SeRF indices. This redundancy – duplicating the vector component once per relational attribute – leads to prohibitive storage overhead.

DSG [6] extends SeRF by relaxing the strict ordering constraint, thereby enabling dynamic insertions. Yet, this flexibility comes at the cost of additional space overhead, reintroducing the index size issue that SeRF originally addressed. In our experiments, we observe that the index size of DSG even surpasses that of iRangeGraph.

C. Pre-filtering

Pre-filtering is a baseline approach for supporting general filtered vector search: it applies all relational predicates to the dataset first and then performs vector search on the resulting filtered subset. While flexible, pre-filtering is only efficient when the combined relational filters yield a extremely selective predicate. The inefficiency stems from the lack of an index over the runtime-generated filtered result, forcing vector search to fall back on a brute-force scan over potentially large intermediate filtered result – a process that rapidly

becomes impractical once the filtered result exceeds a few hundred entries. Consequently, pre-filtering is only effective for predicates with extremely low passrates—typically below 0.1% for million-scale datasets—where only a handful of vectors remain after filtering.

Moreover, accurately estimating query selectivity with multiple attributes remains a long-standing *cardinality estimation* [24] challenge, despite advances including recent learning-based approaches [25]–[27]. As a result, reliance on pre-filtering introduces high risk of unpredictable latency due to mis-estimation, highlighting the need for a general solution that is less dependent on precise cardinality estimation.

D. Post-filtering

Post-filtering is another common technique for supporting general filtered vector search. For conjunctive predicates (e.g., $a_1 \wedge a_2$), a set of k' candidate records is retrieved using vector search as the first step, and then this set is filtered according to the attribute predicate as the second step. However, post-filtering is also fundamentally challenged by the cardinality estimation problem: it is difficult to determine an appropriate initial search size, k' , that ensures sufficient candidates will satisfy the later relational filtering. As a result, post-filtering often devolves into multiple search rounds with progressively increasing k' , leading to inefficient and unpredictable performance. This inefficiency is further exacerbated as the predicate’s passrate decreases – the lower the selectivity, the poorer the performance. This means that more selective predicates can actually increase query latency, contrary to the typical database expectation that query cost should decrease with lower passrates due to less data being accessed. Nonetheless, post-filtering offers a small advantage over pre-filtering for filtered search: it can leverage any specialized indices built for 1D filtering for vector-search in the first step.

For disjunction predicates (e.g., $a_1 \vee a_2$), the most efficient method is to leverage the pre-built 1D specialized index to locate the eligible records for each queried attribute, union them, and sort the union according to their vector distance from the query. In other words, for this approach, we expect a degradation in QPS when more attributes are queried in a conjunction predicate.

E. In-filtering and General Filtered Search

ACORN [2] and NaviX [7] stand out as the only general and universal algorithms currently available for supporting filtered vector search without incurring substantial space overhead. Rather than introducing a novel index, they leverage the widely adopted graph-based index HNSW, applying traversal heuristics to restore graph connectivity disrupted by relational filtering. For instance, they may explore two-hop neighbors instead of standard one-hop traversal, enabling efficient navigation among eligible records after applying attribute filters. By restricting vector distance computations to records that satisfy the predicate, they effectively limit unnecessary comparisons and prioritize vectors passing the filters.

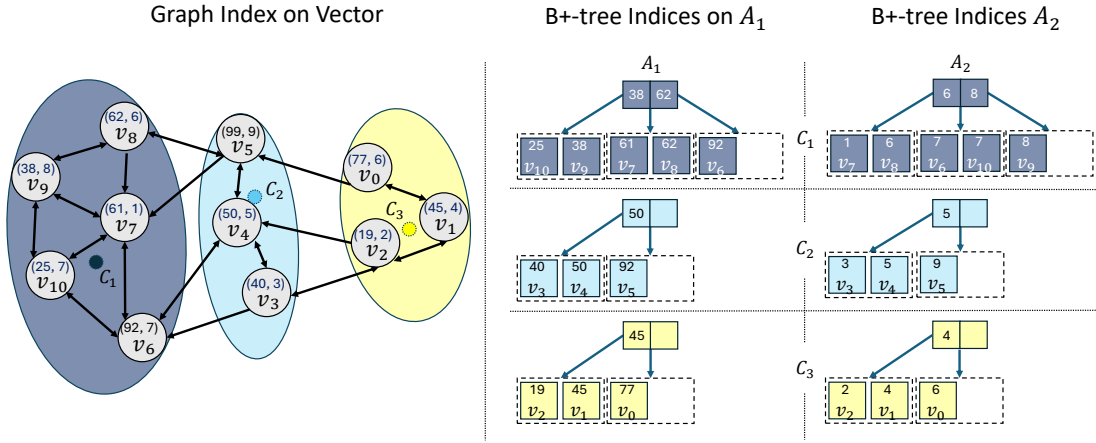


Fig. 2. An illustration of COMPASS index, only the bottom layer graph is shown for HNSW and each color denotes a cluster in the IVF index.

However, this in-filtering approach does not always yield high query throughput in practice. The computational savings from reduced distance calculations can be offset by the overhead of locating predicate-passing vectors within the index. As such, while their design ensures that the number of distance comparisons decreases with lower predicate passrate, this improvement do not necessarily translate into proportional increases in query-per-second (QPS), due to unavoidable costs in candidate identification and traversal during query processing.

SIEVE [28], as another approach to general filtered search, builds multiple proximity graphs for different filtered subsets within a memory size budget and based on the history query workload. SIEVE’s design is orthogonal to other methods that either build specialized indices [3], [5], [6], [21]–[23] or employ special traversal strategies [2], [7].

IV. THE COMPASS ALGORITHM

A. Index Construction

COMPASS assumes a schema where each record contains a vector and one or more numerical attributes. We build a proximity graph index $\mathcal{G}$ (HNSW [8] by default) on the vectors of all records, leveraging the high efficiency of proximity graph for A- k NN [29], [30]. For the numerical attributes, we first group all records into clusters based on their vectors, using an IVF index like [17]. Then, within each cluster, we build a separate B+-tree for each numerical attribute. We collectively refer to the combination of IVF and B+-tree as *clustered B+-trees* and denote it as $\mathcal{B}$.

Example 1. Figure 2 illustrates an example of the Compass index, where each record consists of one vector and two numerical attributes, A_1 and A_2 . For instance, vector v_8 has two attribute values (62, 6) in the figure. For the vector component, we construct an HNSW, and only the bottom layer of HNSW is shown. For the two relational attributes A_1 and A_2 , we first cluster the vectors using an IVF and then build indices for the relational attributes within each cluster

Algorithm 1 COMPASSSEARCH

Require: graph index $\mathcal{G}$, clustered B+-trees $\mathcal{B}$, query vector q , predicate p , #result k , expansion factor ef

- 1: $SharedQ \leftarrow$ empty min-heap
- 2: $Visited \leftarrow$ empty bitmap
- 3: $TopQ \leftarrow$ empty max-heap
- 4: $\mathcal{G}.OPEN(q, p, SharedQ, Visited)$
- 5: $\mathcal{B}.OPEN(q, p, SharedQ, Visited)$
- 6: **while** $TopQ$ has not reached size ef **do**
- 7: $records, sel = \mathcal{G}.NEXT(SharedQ, Visited)$
- 8: **for** $record \in records$ **do**
- 9: $TopQ.PUSH(record)$
- 10: **if** $sel < \beta$ **then**
- 11: $records = \mathcal{B}.NEXT(SharedQ, Visited)$
- 12: **for** $record \in records$ **do**
- 13: $TopQ.PUSH(record)$
- 14: **while** $TopQ.SIZE() > k$ **do**
- 15: $TopQ.POP()$
- 16: **return** $TopQ$

of vectors. In Figure 2, the dataset is partitioned into three clusters, C_1 , C_2 , and C_3 . Within each cluster, we build two B+-trees — one for each attribute.

B. Query Processing

The key idea of Compass is to jointly leverage the proximity graph $\mathcal{G}$ (efficient at similarity-based vector search) and the relational indices (efficient at identifying records that satisfy the predicate) for filtered vector search. We employ the proximity graph $\mathcal{G}$ as the primary driving force given its high efficiency. However, if only a small number of current candidate’s neighbors pass the predicate (i.e., low neighborhood passrate¹), the graph traversal can become confined to a component disconnected from other graph regions containing the predicate-satisfying records [2]. To address this, the

¹We refer to the *neighborhood passrate* of a node in the proximity graph as the portion of its neighbors that satisfy the given predicate.

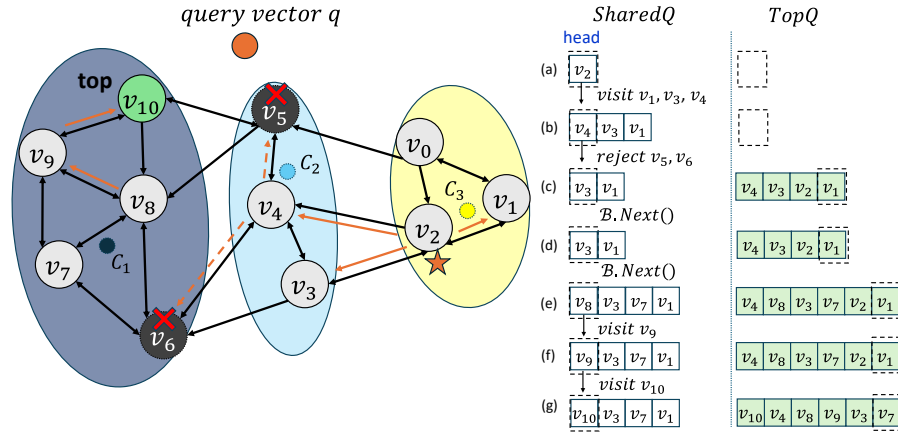


Fig. 3. An illustration of COMPASSSEARCH with single attribute A . Gray nodes pass the predicate while black ones do not. Green node marks the top-1 result. Orange dot marks the query vector while orange star marks the graph entry point. Cyan, blue and yellow represent different clusters. For conciseness, B-tree is hidden in the figure.

clustered B+-trees $\mathcal{B}$ help the graph traversal escape these isolated components. Specifically, we use $\mathcal{B}$ to retrieve a batch of predicate-satisfying records from IVF clusters whose centroids are close to the query vector. The proximity graph and clustered B+-trees cooperate via a shared candidate queue that ranks candidates by their vector distance to the query; both indices can contribute to this shared queue.

Algorithm 1 details the overall query processing of Compass and formalizes the ideas discussed above. It begins by creating a shared candidate queue, *SharedQ*, which maintains the candidate records to visit; a shared visited bitmap, *Visited*, which flags the records whose distances have been computed; and the top queue, *TopQ*, which stores the intermediate query result (Lines 1 to 3). Then, the query vector, predicate, shared queue and visited bitmap are passed to the proximity graph ($\mathcal{G}$) and the clustered B+-trees ($\mathcal{B}$) to initialize their respective search states (Lines 4 to 5). Both $\mathcal{G}$ and $\mathcal{B}$ follow the pull-based iterator interface [31]. Their OPEN and NEXT procedures are detailed in Algorithm 2 and Algorithm 3, respectively. Currently, we can view the NEXT interface of $\mathcal{G}$ as returning a batch of vectors that are encountered during the graph traversal and pass the predicates, and the NEXT interface of $\mathcal{B}$ as returning a batch of vectors that pass the predicates and consecutive NEXT calls return the vectors in their cluster order as discussed earlier.

The main loop (Lines 6 to 13) continues until the *TopQ* reaches the preset search size ef (Line 6). As such, we can use *TopQ* to control the recall and query processing time. In the beginning of the loop, a batch of candidates that pass the predicates are pulled from the proximity graph via $\mathcal{G}.NEXT$ (Line 7). Beside returning the candidates, $\mathcal{G}.NEXT$ also returns the neighborhood passrate sel around the currently visiting candidate (Line 7). If sel is low than a threshold β (set to 0.05 by default, Algorithm 1), the algorithm pulls a batch of candidates that pass the filters from the clustered B+-trees via $\mathcal{B}.NEXT$ (Line 11). This injection of candidates mitigates the connectivity issue caused by low passrate, allowing the graph

search to continue from these candidates from the clustered B+-trees. This cooperative hand-off is enabled by maintaining the shared candidate queue by both $\mathcal{G}$ and $\mathcal{B}$ during their NEXT operations. The main loop ends when there are enough candidates in the top queue *TopQ*, and by then the top- k result from *TopQ* are returned (Lines 14 to 16) as the final search result.

Example 2. Figure 3 illustrates the search process of Compass for an example query on one attribute. In the figure, nodes colored black represent vectors whose relational values do not satisfy the predicate, while nodes in gray indicate those that do. The search starts with using the graph index and selects an entry point v_2 and (a) enqueues it into the shared candidate queue *SharedQ*. Since v_2 's entire neighborhood passes the predicate, (b) it visits all its neighbors v_1, v_3, v_4 and pushes them into *SharedQ*. Next, the search explores v_4 's unvisited neighbors v_5, v_6 and finds that they all fail the predicate filter. This indicates that the neighborhood selectivity around v_4 is poor, indeed 0. In this case, the search would (c) consult the clustered B+-trees, which examines the cluster that is currently closest to the query (C_2 in this example), and use its corresponding B+-tree to retrieve the predicate-passing records: v_3, v_4 .

Since v_3 and v_4 have already been visited, the clustered B+-trees would examine the next cluster closest to q (C_1 in this example) in order to return enough tuples for its NEXT operation. In the example, it (d) retrieves predicate-passing records from the B+-tree of C_1 and adds them into *SharedQ*. If there are too many new candidates, the clustered B+-trees would only insert into the *SharedQ* and return a sample of them (v_7 and v_8). After that, the main loop of Algorithm 1 starts a new iteration and goes back to Algorithm 1 of Algorithm 1. Inside $\mathcal{G}.NEXT$, it internally (e) first visits v_8 , then (f) visits v_9 , and (g) finally reaches the optimal result v_{10} . Since v_8, v_9 have already been visited and the only unvisited neighbor v_5 of v_{10} fails to pass the predicate, $\mathcal{G}.NEXT$ in Line

TABLE II
SYMBOL TABLE OF PROGRESSIVE SEARCH STATE

Symbol	Explanation
<i>CandiQ</i>	shared min-heap storing next candidate to expand
<i>Visited</i>	shared bitmap flagging the visited status of records
<i>TopQ</i>	internal max-heap storing visited top records with max size <i>efs</i>
<i>RecycQ</i>	internal min-heap storing visited records not in internal <i>TopQ</i>
<i>ResQ</i>	internal min-heap storing visited top filtered records from $\mathcal{G}$
<i>efs</i>	internal expansion factor controlling the search width of $\mathcal{G}$
<i>stepsize</i>	step size to increase <i>efs</i> to enlarge search width
<i>CandiQ</i>	shared min-heap storing next candidate to expand
<i>Visited</i>	shared bitmap flagging the visited status of records
<i>RelQ</i>	internal min-heap storing visited top filtered records from $\mathcal{B}$
<i>efi</i>	internal expansion factor of relational indices

7 of Algorithm 1 returns v_9 and v_{10} as output. Finally, since the size of *TopQ* is already large enough in this step, the search ends by returning the top-*k* elements in *TopQ*.

C. Progressive Search

Motivations. As discussed earlier, we employ the proximity graph as the main driving force in Compass. The search process of proximity graph is controlled by two priority queues, i.e., a min-heap candidate queue that stores the potential records to visit, and a max-heap top queue that maintains the nearest neighbors discovered thus far. The traversal stops when the closest node in the candidate queue is farther from the query than the most distant node in the *efs*-sized top queue. As such, *efs* controls the query processing time and result quality. To utilize proximity graph for filtered search, ACORN [2] and NaviX [7] adopt “in-filtering” by computing distances only for records that pass the predicate. However, when the predicate’s selectivity is low or moderate, the resulting subgraph of predicate-satisfying nodes often becomes disconnected. This leads to a significant performance degradation: the search gets trapped in a local region, wasting computations on nodes that are locally proximate but globally distant from the true, predicate-satisfying nearest neighbors.

We attribute this “trapping” problem to using a fixed *efs* in existing methods, making traversal carry on without being able to identify the disconnectivity problem. To overcome this, Compass starts with a small initial *efs* and progressively enlarges it in discrete steps. In particular, at the end of each step, i.e., when the current *efs* limit is reached, the algorithm evaluates its search progress based on the neighborhood passrate. A high neighborhood passrate indicates the graph search is effective and the visited graph region has not become a trap that isolate current candidate from other predicate-satisfying regions; and the algorithm proceeds by enlarging *efs* to continue its traversal. If the neighborhood passrate is low, it signals that the search is likely confined by disconnected subgraph. The algorithm then pivots, querying the clustered B+-trees, which inject new and diverse candidates from which the graph traversal can continue improving, to navigate out of the current local region. After that, *efs* can be enlarged accordingly. This technique essentially introduces checkpoints

Algorithm 2 Proximity Graph’s Iteration Interface

```

1: function  $\mathcal{G}.$ OPEN( $q, p, SharedQ, Visited$ )
2:    $this.q = q, this.p = p$ 
3:    $this.CandiQ = SharedQ$ 
4:    $this.Visited = Visited$ 
5:    $this.TopQ \leftarrow$  empty max-heap
6:    $this.ResQ \leftarrow$  empty min-heap
7:    $this.RecycQ \leftarrow$  empty min-heap
8:    $CandiQ.PUSH(SELECTENTRYPOINT(this.q))$ 
9: function  $\mathcal{G}.$ NEXT( $SharedQ, Visited$ )
10:   $this.EXPANDSEARCH()$ 
11:  while  $SharedQ$  is not empty do
12:     $dist, record = SharedQ.POP()$ 
13:    if  $dist > TopQ.TOP().dist$  then break
14:     $sel =$  the  $record$ ’s neighborhood passrate
15:    if  $sel \geq \alpha$  then ONEHOPEXPAND()
16:    else if  $sel \geq \beta$  then TWOHOPEXPAND()
17:    else break
18:   $records = []$ 
19:  while  $ResQ.NOTEMPTY()$  and  $cnt++ < k$  do
20:     $records.PUSH(ResQ.POP())$ 
21:  return  $records, sel$ 
22: function  $\mathcal{G}.$ EXPANDSEARCH
23:   $this.efs += this.stepsize$ 
24:  while  $RecycQ.NOTEMPTY()$  and  $TopQ.SIZE() < this.efs$  do
25:     $top = RecycQ.POP()$ 
26:     $TopQ.PUSH(top)$ 
27:    if  $top$  never added to  $SharedQ$  then
28:       $SharedQ.PUSH(top)$ 
29:      if  $p(top)$  is true then
30:         $ResQ.PUSH(top)$ 

```

by progressively enlarging *efs*, and thus we call it *progressive search*.

To support progressive search, we identify the key variables that describe this process and separately list them for $\mathcal{G}$ and $\mathcal{B}$ in Table II. In the following, we describe the OPEN and the NEXT interface for both the proximity graph object $\mathcal{G}$ and clustered B+-trees object $\mathcal{B}$.

Operations on the proximity graph. Algorithm 2 lists the OPEN and NEXT procedures for the proximity graph object $\mathcal{G}$. In particular, $\mathcal{G}.$ OPEN begins by referencing the query vector and predicate. (Line 2). It then references the shared queue *SharedQ* as well as the shared visited bitmap *Visited*, and initializes the internal top queue *this.TopQ* (marked with *this* to differentiate with the global top queue in the main loop) like in standard proximity graph search, with the key exception that its candidate queue and visited bitmap are shared with the clustered B+-trees (Line 3 to Line 5). The graph index also internally maintains its own result queue *ResQ* to store the filtered results and recycle queue *RecycQ* to support the NEXT interface (Line 6 to Line 7). Finally, it finds the entry point and pushes it to the shared candidate queue like in standard

proximity graph search (Line 8).

The graph search begins by enlarging its search size efs , enabling it to continue from where it left off in previous step (Line 10). It then pops the best candidate from the shared queue and checks the stop condition like standard proximity graph search (Line 12 to Line 13).

For the popped candidate, the search employs an adaptive expansion strategy based on its neighborhood's predicate pass-rate (Line 14). If the passrate is moderately large ($\geq \alpha$, with α set to 0.3 by default), it opts for a one-hop expansion (Line 15), i.e., visiting all the unvisited one-hop neighbors of the current candidate. If the passrate is moderately low ($\geq \beta$ but $< \alpha$, with β set to 0.05 by default), it employs a limited two-hop expansion (Line 16), i.e., visiting the unvisited predicate-passing one-hop neighbors as well as a subset of unvisited predicate-passing two-hop neighbors of the current candidate. The rationale is that two-hop neighbor expansion leads to predicate-passing records outside the neighborhood. We visit only a subset of predicate-passing two-hop neighbors to avoid excessive attribute filtering.

If the passrate is extremely low (i.e., $< \beta$), the proximity graph determines it is disconnected from other predicate-passing regions and prepares to consult the clustered B+-trees for connectivity enhancement (Line 17). Finally, the close, predicate-passing records found in this round are returned (Line 18 to Line 21). The graph search's termination is dynamically determined (Line 13). Furthermore, because the graph index is predicate-agnostic, the number of predicate-passing records found can vary between rounds. Therefore, our NEXT function returns a batch of results, rather than a single-item iterator (cf. [31]).

To detail the ENLARGESearch mechanism, note that the parameter efs sets the graph search width. At the beginning of $\mathcal{G}.$ ENLARGESearch, this parameter is incremented, which semantically enlarges the search width (Line 23). To materially execute this expansion, a "recycle queue" is employed. This queue maintains the intermediate visited records, which are used to set the candidate queue and top queue to the precise state they would have been in if the search width had been this large from the start (Line 24 to Line 30).

Operations on clustered B+-trees. When the graph traversal become trapped at local region due to low passrate and poor connectivity, Compass pivots to the clustered B+-trees to inject new candidates to navigate the graph traversal out of the local region. The central challenge then becomes efficiently selecting the closest clusters to probe for predicate-passing records while keeping the selection overhead low.

A straightforward solution is to utilize a linear scan over all cluster centroids like a standard IVF. The computation cost is high as there are usually many centroids. Moreover, we seldom need to query the predicate-passing records from all the clusters, and thus the solution wastes computation. To reduce computation, an alternative is to probe a pre-determined number of closest clusters, $nprobe$, via a separate approximate similarity search on the centroids (e.g., with a proximity graph

Algorithm 3 Clustered B+-trees' Iteration Interface

```

1: function  $\mathcal{B}.$ OPEN( $q, p, SharedQ, Visited$ )
2:    $this.q = q, this.p = p$ 
3:    $this.beg = nil, this.end = nil$ 
4:    $this.CandiQ = SharedQ$ 
5:    $this.Visited = Visited$ 
6:    $this.RelQ \leftarrow$  empty min-heap
7:    $this.\mathcal{G}'.OPEN(q, \mathbf{TRUE})$ 
8: function  $\mathcal{B}.$ NEXT( $SharedQ, Visited$ )
9:    $cnt = 0$ 
10:  while  $cnt < efs$  do
11:    if  $beg$  reached end then
12:       $clusidx = \mathcal{G}'.NEXT()$ 
13:       $beg, end = Indices[clusidx].SEARCH(p)$ 
14:    if ! $Visited[beg]$  then
15:       $RelQ.PUSH(\{DIST(q, *beg), *beg\})$ 
16:       $Visited[beg] = true$ 
17:       $cnt++$ 
18:     $beg++$ 
19:   $cnt = 0, records = []$ 
20:  while  $RelQ.NOTEMPTY()$  and  $cnt++ < k/2$  do
21:     $SharedQ.PUSH(RelQ.POP())$ 
22:     $records.PUSH(RelQ.POP())$ 
  return  $records$ 

```

on the centroids). This solution requires difficult parameter tuning: a conservative $nprobe$ cannot inject a sufficient number of new candidates to navigate out of the local region, while an aggressive $nprobe$ incurs superfluous computational overhead on clusters that do not contribute to the final search results.

To resolve the problem, we propose a more dynamic, "on-demand" cluster ranking strategy. In particular, we build a proximity graph on the cluster centroids, named as cluster graph $\mathcal{G}'$, and reuse the previous progressive search method to fetch close clusters. Each time $\mathcal{B}.$ NEXT is called, the efs' for searching the cluster graph is similarly incremented to obtain more clusters. Such a design avoids both the exhaustive computation of a full ranking and the ad-hoc nature of a fixed $nprobe$ heuristic, while intrinsically balancing computational efficiency with the required candidate sufficiency.

Algorithm 3 lists the OPEN and NEXT procedures for the clustered B+-trees object $\mathcal{B}$. At the beginning of $\mathcal{B}.$ OPEN, the query vector and predicate are referenced, and the relational iterators are initialized (Line 2 to Line 3). It then references the shared candidate queue $SharedQ$, shared visited bitmap $Visited$ and initializes its own internal "relational queue" for storing close, predicate-passing candidates (Line 4 to Line 6). Clustered B+-trees maintains the small cluster graph ($\mathcal{G}'$) built on the cluster centroids to progressively retrieve close clusters. Since this cluster graph's purpose is to find centroids by vector proximity, it is passed with an "always-true" predicate, causing it to degenerate into a pure progressive similarity search. $\mathcal{B}.$ OPEN concludes by initializing the search state for the cluster graph without sharing candidate queue or visited bitmap (Line 7).

Algorithm 4 VISIT

Require: unvisited *record*, passing predicate or not *passed*

```
1: Visited[record] = true
2: dist = DIST(q, record)
3: if TopQ.SIZE() < efs or dist < TopQ.TOP().dist then
4:   SharedQ.PUSH({dist, record})
5:   TopQ.PUSH({dist, record})
6:   if TopQ.SIZE() ≥ efs then
7:     RecycQ.PUSH(TopQ.POP())
8:   if passed then
9:     ResQ.PUSH({dist, record})
10: else
11:   RecycQ.PUSH({dist, record})
```

When the clustered B+-trees are invoked to propose candidates via *B*.NEXT, it fetches a fixed number, *efi*, of predicate-passing records from the close clusters (Line 9 to Line 18) by querying relational indices inside each cluster. If current cluster does not contain sufficient number of records, a new cluster is pulled from the cluster graph $\mathcal{G}'$ to continue the relational candidate proposal (Line 11 to Line 13). We note that cluster graph does not share candidate queue or visited bitmap by explicitly omitting them in the function arguments (Line 12). The expansion factor *efi* is analogous to the proximity graph search’s expansion factor *efs*. Both parameters ensure that each component performs more work (e.g., distance computations) than the number of results returned in a single batch to return quality close records. The $k/2$ batch size is chosen to accommodate the potentially-varied number of records (from 0 to k) returned by the proximity graph search *G*.NEXT.

Details. There are several details omitted from the discussion of the algorithms. First, the proximity graph $\mathcal{G}$ and the clustered B+-trees $\mathcal{B}$ share a common bitmap to track the visited status of all records, ensuring that the vector distance for any given record is computed only once. Second, during graph traversal, the “visit” to a record, as detailed in Algorithm 4, entails computing the record’s vector distance and updating the corresponding queues, serving the purpose of supporting progressive search. Particularly, other than flagging visited, computing distance and maintaining the *SharedQ* and *TopQ* like in a standard HNSW (Line 1 to Line 7), the record is further pushed into result queue to be returned as filtered close record if it passes the predicate (Line 8 to Line 9). If top queue is full and the record is not close enough to the query vector, it is pushed into the recycle queue to be popped out potentially in future step (Line 10 to Line 11).

D. Discussions

By combining a proximity graph with IVF-enhanced relational indices, COMPASS benefits from the following advantages.

Generality. First and most importantly, Compass generalizes across different numbers and types of attribute filters, tackling the general filtered vector search problem. This is due to

that Compass’ proximity graph refers to attribute information on demand only during the index search, instead of being influenced by the attribute information during the index construction.

This is a stark difference from existing specialized indices, e.g., SeRF [3] and iRangeGraph [5], that modify the underlying proximity graph structure to support a limited number of attribute filter (indeed 1 numerical attribute filter). When there is update on the attribute value, these methods need to completely rebuild the index from scratch. While in our case, only the B+-trees need to be updated with a small overhead. We note that supporting general filter is important because our industry collaborator handles tens of attributes and arbitrary conjunctions and disjunctions over the filters on individual attributes, forming extremely composite predicate.

Efficiency. As will be shown in our experiments, Compass maintains a reasonably high query processing efficiency across predicate patterns (e.g. single- and multi-attributes, varying passrates, conjunction, disjunction). In particular, when the passrate is high or moderate, “in-filtering” traversal is efficient by leveraging the graph connectivity. In this case, Compass will seldom engage the clustered B+-trees and mainly employ the proximity graph. When the passrate is low, pulling from the clustered B+-trees can supply quality candidates. In this case, Compass will mainly rely on the clustered B+-trees to identify the predicate-passing records in the order of their cluster centroid distance to the query. Compass smoothens the switch between the two cases by using a shared candidate queue between the proximity graph and clustered B+-trees with the neighborhood passrate as the signal to pivot inbetween.

Compass is also efficient in the index construction and storage. The IVF index and the relational indices can be built quickly while a normal proximity graph construction dominates the index construction time, in comparison to specialized indices like SeRF [3], iRangeGraph [5] and DSG [6] that incur significantly longer time for index construction.

In terms of storage, the clustered B+-trees store the cluster centroids, edge of a small cluster graph and relational indices. These overhead is small compared to the storage required for the proximity graph’s edge information. Overall, Compass introduces only a minor storage overhead on top of the base proximity graph index, in comparison to specialized indices that would require one index per attribute in multi-attribute setting.

Flexibility. By separating the indices for vector similarity and attribute filtering, Compass benefits from the flexibility in index choice. For instance, the HNSW index can be replaced with a different proximity graph algorithm like NSG [11], or the per-attribute B+-trees could be replaced with a single multi-dimensional tree like R-tree [32]. This allows Compass to seamlessly integrate with the latest development in indexing technique. For example, the update on vectors (e.g., insertion or deletion) can be easily supported by recently-developed algorithms to update the proximity graph index [33], the IVF index [34] respectively. Besides, update to the relational

TABLE III
DATASET SPECIFICATION.

Dataset	#Vectors	#Dimension	Type
CRAWL	1,989,995	300	text embedding
VIDEO	1,000,000	1024	video embedding
GIST	982,694	960	image descriptor
GLOVE100	1,183,514	100	word embedding

attributes only requires update to relational indices, leaving the proximity graph and IVF clusters intact.

Limitations. The highly-modular and general-filter design of COMPASS inevitably influences its search performance when compared to highly-specialized indices in their optimal settings (specifically in single-attribute case). However, as demonstrated in our experiments, the performance gap between Compass and these specialized indices in their preferred settings is often minimal, and in some cases, Compass even outperforms them. Additionally, it’s important to highlight that these specialized indices, when applied to a relational schema with n attributes, require n times the storage redundancy for the vector component.

Nevertheless, for Compass, we identify that a more optimized design on relational predicate query can be employed. Currently, when querying the predicate in the clustered B+-trees, COMPASS selects the B+-tree on a random attribute and conducts a linear scan to filter on the remaining attributes over the return records on that selected attribute. Though the distance computation is expensive on vectors and the cost of attribute filtering is relatively low for the entire vector search process, we believe, when integrated into more sophisticated system, the cost of attribute filtering can be further reduced by deciding an optimal evaluation order by following the classic query planning literature [35].

V. EVALUATION

In this section, we evaluate the performance of Compass against a range of existing methods on various datasets.

A. Datasets, Workloads, and Metrics

We evaluate on four vector datasets: CRAWL, GIST, VIDEO, and GLOVE100. CRAWL² consists of 300-dimensional text embeddings [13] derived from crawled web content. VIDEO³ contains 1,024-dimensional video feature vectors subsampled from the YouTube-8M dataset. GIST⁴ comprises 960-dimensional floating-point image feature descriptors. GLOVE100⁵ contains 100D word embeddings obtained from GLoVe algorithm [36]. They span a variety of source data modalities and number of dimensions. The number of base vector and the dimension of the base vector for every dataset is detailed in Table III. We note that GIST and VIDEO contain duplicate vectors and we have deduplicated for them

in the remaining evaluation. For each vector, we augment it with four uniformly generated relational attributes.

By default, each query is a general range-filtered query with $k=10$ and a selectivity (passrate) of 30% for each relational attribute, achieved by appropriately adjusting the query range. Each experiment runs a workload of 200 queries, focusing solely on search operations, with no insertions or deletions.

Following existing works [2]–[5], we measure the average throughput in the unit of queries per second (QPS) and measure the average accuracy using recall defined as $|\mathcal{R} \cap \mathcal{R}'|/k$, where $\mathcal{R}$ is the result set and $\mathcal{R}'$ is the groundtruth set, supposing all the queries can return up to $k=10$ nearest vectors. Additionally, we track the number of vector distance computations (#Comp).

Platform and Configuration. All our code are implemented with C++. All the methods are compiled with GCC version 10.2.1 and compilation option `-O3 -march=native`. SIMD instructions have been enabled for all compared methods. All the experiments are conducted on Debian 11 with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and 256GB of RAM. Index search performance is evaluated with single thread.

B. Evaluated Methods and Index Size

We mainly compare with the following existing works.

- **SeRF** [3]. SeRF represents the state-of-the-art in specialized indexing for 1D attribute filtering. When evaluating with multiple attributes, we build a 1D-specialized index for each relational attribute and use post-filtering for conjunction. Following [3], we set the construction expansion factor $K=200$ across all datasets, set maximum out degree $M=32$ on CRAWL and GIST and $M=64$ on VIDEO and GLOVE100 for SeRF. For its other specific index construction parameter, i.e. *efmax*, we use its default value 500. Table IV reports the size of SeRF under this configuration alongside other methods. For all 1D-specialized indices (SeRF, iRangeGraph [5], DSG [6]), we have to build four of them, one per relational attribute. Based on their sizes, we exclude SeRF’s successor, DSG [6], from our performance study: under our experimental setting, which does not involve vector insertions, DSG would offer no performance benefit over SeRF but incurs substantially greater space overhead. For the same reason, we exclude iRangeGraph [5] from our performance study as its index size is almost an order larger than SeRF and Compass. Any observed performance advantage of iRangeGraph would therefore reflect a trade-off between speed gains and the unreasonable expense of memory bloat.
- **NaviX** [7] and **ACORN** [2] are, to date, the only two solutions specifically designed for general filtered vector search. **Milvus** [37] and **Weaviate** [38] also integrate general pre-filtering and post-filtering mechanisms within their engines. However, prior studies have shown that these general-purpose systems are outperformed by specialized approaches such as SeRF and NaviX by a substantial

²<https://commoncrawl.org/>

³<https://research.google.com/youtube8m/>

⁴<http://corpus-texmex.irisa.fr/>

⁵<https://nlp.stanford.edu/projects/glove/>

TABLE IV
COMPARING THE INDEX SIZES OF COMPASS WITH BASELINES.

Dataset	Compass (Graph + IVF + Clustered B+trees)	SeRF	NaviX	iRangeGraph	DSG
CRAWL	275+13+48*4=480MiB	269*4=1076MiB	592MiB	1.8*4=7.2GiB	3.7*4=14.8GiB
VIDEO	137+80+24*4=313MiB	129*4=516MiB	999MiB	0.7*4=2.8GiB	0.9*4=3.6GiB
GIST	138+38+24*4=272MiB	150*4=600MiB	915MiB	1.1*4=4.4GiB	2.4*4=9.6GiB
GLOVE100	164+9.3+24*4=269.3MiB	139*4=556MiB	135MiB	0.9*4=3.6GiB	1.6*4=6.4GiB

margin [3], [7]. NaviX is shown to be $2\times$ to $3\times$ better than ACORN [2] in terms of QPS. Therefore, including and comparing against SeRF and NaviX alone is sufficient for a fair and representative evaluation.

NaviX employs plain HNSW as its index. In consistence with SeRF, we set the construction expansion factor $K=200$, set maximum out degree $M=16$ on CRAWL and GIST, $M=32$ on VIDEO and GLOVE100, since the out degree of bottom-level HNSW graph doubles that amount.

For Compass, its construction mainly employs HNSW and clustering algorithm. We set the expansion factor during construction $K=200$, set maximum out degree $M=16$, number of clusters $nlist=10000$ on CRAWL and GIST, $M=32$ and $nlist=20000$ on VIDEO and GLOVE100. As all methods' query execution is controlled by the expansion factor ef , we vary ef from 10 to 1000, incremented by 5 before 100, by 10 before 200, by 50 before 500, by 100 before 1000.

Now, we take a closer look at the index sizes of Compass, SeRF, and NaviX in Table IV.

Compass, as a general-purpose solution, maintains three complementary structures: (1) a vanilla HNSW graph index to store the neighbor IDs of base vectors; (2) IVF centroids together with a small cluster graph; and (3) a B+-tree for each relational attribute within each cluster. Unlike specialized 1D indexing with post-filtering, Compass requires no vector index duplications across relational attributes, its index size is about 50% of SeRF, 5% of iRangeGraph, and 2.5% of DSG. Notably, Compass's index size could be further reduced by replacing the B+-trees with learned indexes [9], and even more so by leveraging the static nature of the dataset—since no vector insertions occur—making it possible to employ static learned indices like PGM [10], which are even more compact. NaviX, as a general-purpose solution like Compass, exhibits index sizes that are comparable to those of Compass.

C. Conjunctions

Figure 4 and Figure 5 present the query throughput (queries per second, QPS) and the number of vector distance computations for Compass, SeRF (with post-filtering), and NaviX as the number of conjunctive relational predicates varies from one to four. Each attribute forms a part of a conjunctive predicate, with experiments conducted under three recall thresholds: 0.85, 0.9 and 0.95. Figure 4 presents the full results of recall 0.9 on all four datasets. Figure 5 presents the results of recall 0.85 and 0.95 on VIDEO and GIST only due to space reasons (results on the other two datasets are similar).

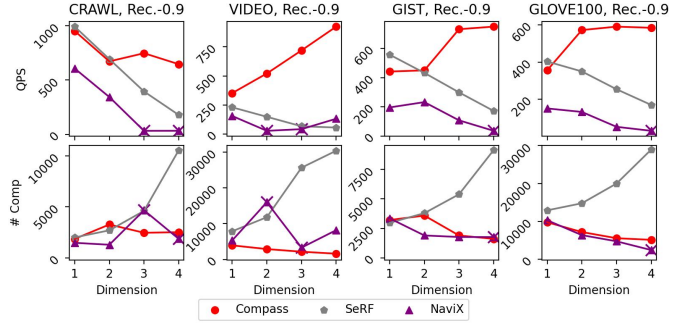


Fig. 4. Conjunction Range Filtering. 0.9 Recall.

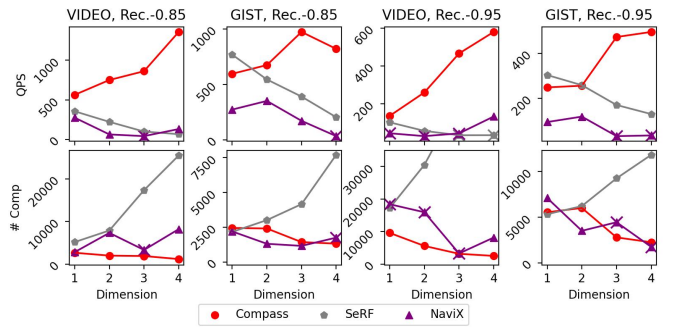


Fig. 5. Conjunction Range Filtering. 0.85/0.95 Recall. On VIDEO and GIST.

Since we set the selectivity (passrate) of each attribute to 30%, the overall passrate for the conjunctive predicate decreases multiplicatively – from 30% with one attribute, to $0.3^2 \approx 10\%$ for two attributes, $0.3^3 \approx 3\%$ for three attributes, and $0.3^4 \approx 1\%$ for four attributes. These scenarios reflect practical settings, where range predicates in traditional databases typically span moderately selective (30% passrate) to highly selective (1% passrate) queries [39], [40].

The results are consistent across all recall thresholds: Compass achieves QPS comparable to SeRF (with post-filtering) in low-dimensional scenarios (1D and 2D) and outperforms SeRF in higher dimensions (3D and 4D). In all cases, Compass consistently surpasses NaviX in QPS performance.

We observe that NaviX generally incurs fewer distance computations than Compass but ultimately achieves a lower QPS. This is because NaviX only calculates vector distances for predicate-passing records. However, to maintain navigation through the graph despite potential disruptions caused by attribute filtering, it needs to frequently visit two-hop neighbors.

TABLE V
SPEEDUP IN QPS BY COMPASS OVER BEST BASELINE. * INDICATES THE METHOD HAS NOT REACHED TARGET RECALL.

QPS Speedup Ratios of Conjunctions at Recall=0.9																
Method	CRAWL				VIDEO				GIST				GLOVE100			
	1D	2D	3D	4D	1D	2D	3D	4D	1D	2D	3D	4D	1D	2D	3D	4D
SeRF	991	693	395	179	230	149	67	56	556	432	298	171	402	349	253	169
Navix	606	344	33*	33*	156	28*	43*	132	195	231	107	35*	150	131	51	29*
Compass	951	672	747	646	352	520	718	922	441	449	731	749	355	572	590	584
Speedup	0.95x	0.96x	1.89x	3.6x	1.53x	3.49x	10.71x	6.98x	0.8x	1.04x	2.45x	4.38x	0.88x	1.64x	2.33x	3.46x

QPS Speedup Ratios of Disjunctions at Recall=0.9																
Method	CRAWL				VIDEO				GIST				GLOVE100			
	1D	2D	3D	4D	1D	2D	3D	4D	1D	2D	3D	4D	1D	2D	3D	4D
SeRF	991	492	421	324	230	125	89	69	556	276	205	172	402	236	150	111
Navix	606	362	542	521	156	112*	83*	188	195	204	229	275	150	204*	174*	142
Compass	951	1652	1546	1885	352	375	424	402	441	537	631	647	355	401	447	511
Speedup	0.95x	3.35x	2.85x	3.61x	1.53x	3x	4.76x	2.13x	0.8x	1.95x	2.75x	2.35x	0.88x	1.7x	2.98x	3.6x

As a result, substantial time is spent on predicate evaluation for a quadratic number of neighbors, which severely hurts its overall QPS.

We also note that NaviX often fails to reach the target recall, even when the number of computations is small. These cases are marked with “×” labels in the figures. In these figures, the QPS and the number of computations for NaviX marked with × represent the results after it has exhausted the largest search size ($ef=1000$). The reason NaviX frequently fails to achieve the target recall is that exploring two-hop neighbors is ineffective when the graph is disconnected into disjoint components due to attribute filtering. In such cases, any graph traversal will remain trapped within a component indefinitely.

Another observation is that Compass exhibits a desirable property similar to that of classical relational database systems, where QPS generally increases as the number of relational filters grows. In contrast, the other methods experience the opposite trend – QPS degrades as the number of relational attributes increases. This property of Compass is attributed to the cohesive cooperation between the proximity graph and the clustered relational indices, which help guide any disconnected traversal out of the sub-component. In contrast, other methods perceive additional relational filters as increasingly challenging due to the increased graph disconnectivity.

D. Disjunctions

Figure 6 presents the results using disjunctive predicates under recall 0.9 on all datasets while Figure 7 presents the results under recalls 0.85 and 0.95 for VIDEO and GIST only. Since the default selectivity (passrate) of each attribute is 30%, the overall passrate for the disjunctive predicate increases additively – from 30% with one attribute, to 60% for two attributes, 90% for three attributes and 100% for four attributes.

The results are consistent across all recall thresholds and align with the findings in conjunctions: Compass consistently outperforms NaviX in QPS. Furthermore, in comparison to conjunctions, Compass now significantly outperforms SeRF

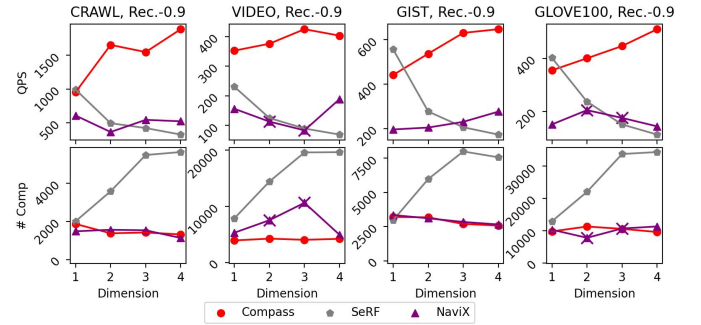


Fig. 6. Disjunction Range Filtering. 0.9 Recall.

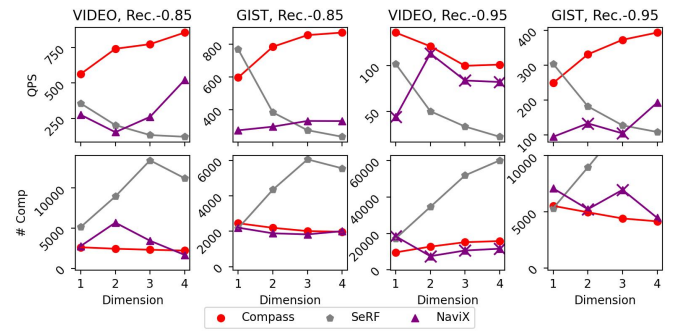


Fig. 7. Disjunction Range Filtering. 0.85/0.95 Recall. On VIDEO and GIST.

once beyond one-dimensional queries. That is because SeRF requires one graph-index traversal per queried attribute, after that, it unions and sorts the unions to return k results.

Besides the robustness Compass has demonstrated in conjunctions and disjunctions, we summarize in Table V its speedup ratios in QPS relative to the best baseline, in accordance with previous evaluation.

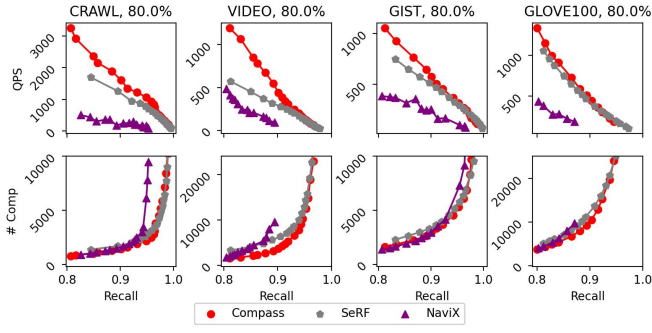


Fig. 8. QPS and Distance Computation vs Recall. 80% passrate

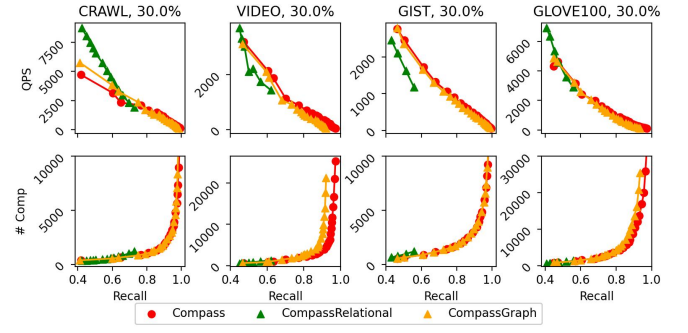


Fig. 11. Ablation Study

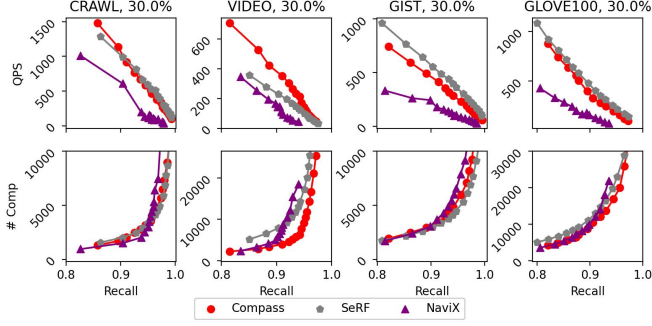


Fig. 9. QPS and Distance Computation vs Recall. 30% passrate

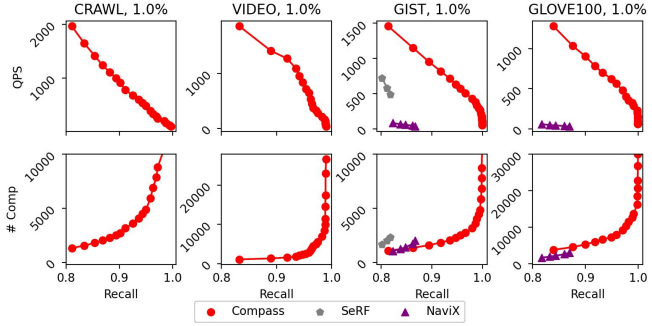


Fig. 10. QPS and Distance Computation vs Recall. 1% passrate

E. QPS / Number of Distance Computations vs Recall

Figures 8 to 10 illustrate the QPS and the number of vector distance computations for all methods on a single attribute for recall ratios from 0.8 to 1.0 by varying ef . Given that, for conjunction and disjunction, varying the number of attributes is equivalent to varying the selectivities, we present the results in three distinct selectivities for this experiment: an 80% passrate (not selective), a 30% passrate (default), and a 1% passrate (selective).

From the figures, we can see that only Compass can consistently return results with high recall across all three selectivities.

Under low passrate (Figure 10), both NaviX and SeRF have difficulty producing reasonable recall due to the graph disconnectivity and aggressive graph compression strategy [3],

[5] respectively. The proximity graph in Compass, on the other hand, can efficiently move to the other disconnected components due to the navigation from clustered B+-trees. Under high passrate (Figure 8), only NaviX fails to stably return results with high recall (≥ 0.9). This is because under high passrate, NaviX only computes vector distances for predicate-passing records, a strategy that proves myopic and insufficient when the graph connectivity is well-preserved, in contrast to Compass which also evaluates distances for these non-passing neighbors. In such case, a non-passing neighbor may itself have neighbors that are close to the query vector and pass the predicate, which are potentially ignored by NaviX.

F. Ablation Study

In this ablation study, we examine two variants of Compass: **CompassRelational** and **CompassGraph**. The CompassRelational variant is obtained by removing the proximity graph component, relying solely on the clustered B+-trees to progressively fetch candidates from clusters close to the query vector. The CompassGraph variant is obtained by setting the number of clusters $nlist=1$, which effectively leaves the proximity graph to be navigated by only a single B+-tree built on the attribute values of the entire dataset.

Figure 11 illustrates the QPS and the number of vector distance computations for recall ratios from 0.4 to 1.0 by varying ef for this study. The passrate in this study is the default passrate, 30%. All other index construction and search parameters remain the same as in our main evaluation.

As shown, CompassRelational cannot return results with high recall across any of the four datasets because it lacks the proximity graph to approach the query vector. While CompassGraph demonstrates performance similar to Compass on datasets CRAWL and GIST, it fails to return results near 100% recall on the more challenging VIDEO and GLOVE100 datasets. This occurs because the single, global B+-tree—can no longer provide a proximity guarantee. While it can still iterate records that pass the relational predicate, it cannot ensure these candidates are close to the query vector. Without this navigation, the search is unable to fully overcome the induced graph's disconnectivity. This ablation study demonstrates that the proximity graph and the clustered B+-trees are both integral and complementary components of Compass.

VI. CONCLUSIONS

In summary, as a modular solution, COMPASS cohesively leverages existing indices with minimal intrusion into their underlying designs; as a general-filter solution, its vector index construction is entirely predicate-agnostic, with all predicate-specific logic handled dynamically at search time. At the core of the method is its adaptive search strategy, which adjusts to the predicate’s passrate. When the passrate is high, the method operates as progressive post-filtering on the graph index. As the passrate transitions to a moderate level, it incorporates in-filtering techniques. When the passrate becomes low, the IVF component activates to help escape local minima. Throughout this process, Compass continuously balances two objectives: reducing vector distance and maintaining predicate satisfaction—a guided approach that inspired the name *Compass*.

VII. AI-GENERATED CONTENT ACKNOWLEDGMENT

During the preparation of this manuscript, AI language model Gemini 2.5 Pro was utilized for refining the text, improving clarity, and enhancing overall readability. All technical contributions, including the methodology, experimental design, results, analyses, and conclusions, originated entirely from the authors without AI involvement.

REFERENCES

- [1] M. Wang, L. Lv, X. Xu, Y. Wang, Q. Yue, and J. Ni, “An Efficient and Robust Framework for Approximate Nearest Neighbor Search with Attribute Constraint,” in *Thirty-Seventh Conference on Neural Information Processing Systems*, 2023.
- [2] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia, “ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data,” *Proc. ACM Manag. Data*, 2024.
- [3] C. Zuo, M. Qiao, W. Zhou, F. Li, and D. Deng, “SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search,” *Proceedings of the ACM on Management of Data*, 2024.
- [4] J. Engels, B. Landrum, S. Yu, L. Dhulipala, and J. Shun, “Approximate nearest neighbor search with window filters,” in *Proceedings of the 41st International Conference on Machine Learning, ICML’24*, JMLR.org, 2024.
- [5] Y. Xu, J. Gao, Y. Gou, C. Long, and C. S. Jensen, “iRangeGraph: Improving Range-dedicated Graphs for Range-filtering Nearest Neighbor Search,” <https://arxiv.org/abs/2409.02571v1>, 2024.
- [6] Z. Peng, M. Qiao, W. Zhou, F. Li, and D. Deng, “Dynamic Range-Filtering Approximate Nearest Neighbor Search,” *Proc. VLDB Endow.*, 2025.
- [7] G. Sehgal and S. Salihoğlu, “NaviX: A Native Vector Index Design for Graph DBMSs With Robust Predicate-Agnostic Search Performance,” *Proc. VLDB Endow.*, 2025.
- [8] Y. A. Malkov and D. A. Yashunin, “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.
- [9] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, “Are updatable learned indexes ready?,” *Proceedings of the VLDB Endowment*, 2022.
- [10] P. Ferragina and G. Vinciguerra, “The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds,” *Proceedings of the VLDB Endowment*, 2020.
- [11] C. Fu, C. Xiang, C. Wang, and D. Cai, “Fast approximate nearest neighbor search with the navigating spreading-out graph,” *Proc. VLDB Endow.*, 2019.
- [12] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnaswamy, and R. Kadekodi, “DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node,” in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2019.
- [13] C. Fu, C. Wang, and D. Cai, “High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.
- [14] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [15] M. Douze, H. Jégou, and F. Perronnin, “Polysemous Codes,” in *Computer Vision – ECCV 2016* (B. Leibe, J. Matas, N. Sebe, and M. Welling, eds.), Springer International Publishing, 2016.
- [16] A. Babenko and V. Lempitsky, “The inverted multi-index,” in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012.
- [17] H. Jégou, M. Douze, and C. Schmid, “Product Quantization for Nearest Neighbor Search,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2011.
- [18] W. Wu, J. He, Y. Qiao, G. Fu, L. Liu, and J. Yu, “HQANN: Efficient and Robust Similarity Search for Hybrid Queries with Structured and Unstructured Constraints,” in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management, CIKM ’22*, Association for Computing Machinery, 2022.
- [19] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan, A. Singh, and H. V. Simhadri, “Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters,” in *Proceedings of the ACM Web Conference 2023, WWW ’23*, Association for Computing Machinery, 2023.
- [20] J. Mohoney, A. Pacaci, S. R. Chowdhury, A. Mousavi, I. F. Ilyas, U. F. Minhas, J. Pound, and T. Rekatsinas, “High-Throughput Vector Similarity Search in Knowledge Graphs,” *Proceedings of the ACM on Management of Data*, 2023.
- [21] F. Zhang, M. Jiang, G. Hou, J. Shi, H. Fan, W. Zhou, F. Li, and S. Wang, “Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search,” *Proc. ACM Manag. Data*, 2025.
- [22] M. Jiang, Z. Yang, F. Zhang, G. Hou, J. Shi, W. Zhou, F. Li, and S. Wang, “DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter,” *Proceedings of the ACM on Management of Data*, 2025.
- [23] A. Liang, P. Zhang, B. Yao, Z. Chen, Y. Song, and G. Cheng, “UNIFY: Unified Index for Range Filtered Approximate Nearest Neighbors Search,” *Proc. VLDB Endow.*, 2025.
- [24] H. Harmouch and F. Naumann, “Cardinality estimation: An experimental survey,” *Proceedings of the VLDB Endowment*, 2017.
- [25] X. Wang, C. Qu, W. Wu, J. Wang, and Q. Zhou, “Are we ready for learned cardinality estimation?,” *Proceedings of the VLDB Endowment*, 2021.
- [26] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang, “Learned cardinality estimation: A design space exploration and a comparative evaluation,” *Proceedings of the VLDB Endowment*, 2022.
- [27] A. Kipf, T. Kipf, A. Ailijiang, K. Kempfert, E. Semenova, S. Tu, J. Hradil, O. Mutlu, A. Kemper, and T. Neumann, “Estimating correlated joins with deep learning,” in *Proceedings of the VLDB Endowment*, VLDB Endowment, 2019.
- [28] Z. Li, S. Huang, W. Ding, Y. Park, and J. Chen, “SIEVE: Effective Filtered Vector Search with Collection of Indexes,” *Proc. VLDB Endow.*, 2025.
- [29] M. Aumüller, E. Bernhardsson, and A. Faithfull, “ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms,” 2018.
- [30] H. V. Simhadri, G. Williams, M. Aumüller, M. Douze, A. Babenko, D. Baranchuk, Q. Chen, L. Hosseini, R. Krishnaswamy, G. Srinivasa, S. J. Subramanya, and J. Wang, “Results of the NeurIPS’21 Challenge on Billion-Scale Approximate Nearest Neighbor Search,” 2022.
- [31] G. Graefe, “Query evaluation techniques for large databases,” *ACM Computing Surveys*, 1993.
- [32] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” *SIGMOD Rec.*, 1984.
- [33] D. Liu, B. Zheng, Z. Yue, F. Ruan, X. Zhou, and C. S. Jensen, “Wolverine: Highly Efficient Monotonic Search Path Repair for Graph-Based ANN Index Updates,” *Proc. VLDB Endow.*, 2025.
- [34] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang, P. Cheng, and M. Yang, “SPFresh: Incremental In-Place Update for Billion-Scale Vector Search,” in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP ’23*, Association for Computing Machinery, 2023.

- [35] H. Lan, Z. Bao, and Y. Peng, “A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration,” *Data Science and Engineering*, 2021.
- [36] J. Pennington, R. Socher, and C. Manning, “GloVe: Global Vectors for Word Representation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (A. Moschitti, B. Pang, and W. Daelemans, eds.), Association for Computational Linguistics, 2014.
- [37] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, “Milvus: A Purpose-Built Vector Data Management System,” in *Proceedings of the 2021 International Conference on Management of Data*, ACM, 2021.
- [38] E. Dilocker, B. van Luijt, B. Voorbach, M. S. Hasan, A. Rodriguez, D. A. Kulawiak, M. Antas, and P. Duckworth, “Weaviate,” 2024.
- [39] C. Chasseur and J. M. Patel, “Design and evaluation of storage organizations for read-optimized main memory databases,” *Proceedings of the VLDB Endowment*, 2013.
- [40] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. Narasayya, and S. Chaudhuri, “Selectivity estimation for range predicates using lightweight models,” *Proceedings of the VLDB Endowment*, 2019.