NaviX: A Native Vector Index Design for Graph DBMSs With Robust Predicate-Agnostic Search Performance

by

Gaurav Sehgal

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Gaurav Sehgal 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

There is an increasing demand for extending existing DBMSs with vector indices to become unified systems that can support modern predictive applications, which require joint querying of vector embeddings and structured properties and connections of objects. We present NaviX, a **Na***tive* **v***ector* **i***nde***X** for graph DBMSs (GDBMSs) that has two main design goals. First, we aim to implement a disk-based vector index that leverages the core storage and query processing capabilities of the underlying GDBMS. To this end, NaviX is a *hiearchical navigable small world* (HNSW) index, which is itself a graph-based structure. Second, we aim to evaluate *predicate-agnostic* vector search queries, where the k nearest neighbors (kNNs) of a query vector v_Q is searched across an arbitrary subset S of vectors that is specified by an ad-hoc selection sub-query Q_S. We adopt a prefiltering-based approach that evaluates Q_S first and passes the full information about S to the kNN search operator. We study how to design a pre-filtering-based search algorithm that is robust under different selectivities as well as correlations of S with v_Q. We propose an adaptive algorithm that utilizes local selectivity of each vector in the HNSW graph to pick a suitable heuristic at each iteration of the kNN search algorithm. We demonstrate NaviX's robustness and efficiency through extensive experiments against both existing prefiltering- and postfiltering-based baselines that include specialized vector databases as well as DBMSs.

Acknowledgements

First, I would like to thank my supervisor, Dr. Semih Salihoglu. I cannot properly express my gratitude for the guidance and support he has given me throughout my master's program. He took a chance on me, and I will forever be grateful. His passion for research motivated me to maintain a positive attitude and consistently strive to do my best work.

Next, I would like to thank my friends in the DSG lab: Sairaj, Anurag, Tejasvi, Fadhil, Shirley, Karem, Ronak, Kriti, and Benson. They became like family to me throughout this period. Without their unwavering support during difficult times and our many research discussions, I would not be able to complete my work. When I first arrived, Kriti, Fadhil, and Tejasvi helped me settle in and showed me how things worked here. I'm truly grateful to Shirley for helping me through personal challenges last year and for our fun gym sessions together. Through discussions and working on several projects with Anurag, Sairaj, Karem, Benson and Ronak around various research and technical topics, I expanded my knowledge of the AI ecosystem and gained new ideas for my research. Finally, seeing Anurag in the lab every day, even on weekends, working hard inspired me to stay dedicated to my goals.

I would like to thank my best friends, Gulzar Ahmed and Vaibhav Sharma, who were not physically present here but were always looking out for me and supported me throughout this journey.

I would like to thank my grandparents, Sunita Sehgal and Tilak Prakash Sehgal, who have always loved me and prayed for me throughout my life.

I would also like to thank my younger brother, Sahil Sehgal. Since our father passed away, he has been taking care of our mom in India while I'm here. He has always been my biggest support and true backbone. Without him, I wouldn't have been able to come here and pursue my research.

Finally, I would like to thank my parents, Deepak Sehgal and Geeta Sehgal. Both have sacrificed so much throughout their lives to support my studies. Without them, I would not be able to achieve anything. They were always there for me during tough times, believed in me unconditionally, and guided me when I went wrong. This master is dedicated to them.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures viii

List of Tables x

1 Introduction 1

2 Background 6

2.1 Basic Terminologies 6

2.2 HNSW 7

2.3 Predicate Agnostic Filtered Search, Prefiltering & Postfiltering 10

2.4 Query Correlation 11

2.5 Kùzu Overview 12

2.5.1 Storage Structures 12

2.5.2 Node Semimasks 12

3 System Design and Implementation **13**
3.1 Index Creation . . . 13
3.2 Index Search Query Syntax and Plan . . . 15
3.2.1 In-Buffer Manager Distance Computations . . . 16
3.3 Adaptive-Local Search Algorithm . . . 17
3.3.1 Design Space of Fixed Heuristics . . . 17
3.3.2 Adaptive Algorithms . . . 20

4 Evaluation **22**
4.1 Experimental Setup . . . 22
4.1.1 Baselines . . . 22
4.1.2 Datasets . . . 23
4.1.3 Query Workloads . . . 25
4.1.4 Evaluation Metric . . . 27
4.1.5 Index Configurations . . . 27
4.1.6 Other Experimental Setup Details . . . 27
4.2 Prefiltering Fixed Heuristics and Adaptive-G . . . 29
4.3 Adaptive-G and NaviX . . . 32
4.4 Comparison Against External Baselines . . . 34
4.4.1 Comparisons Against Weaviate and Milvus . . . 34
4.4.2 Comparisons Against External Postfiltering Baselines . . . 35
4.5 Evaluation of Navix and Acorn: . . . 40
4.6 Evaluation of In-BM Distance Calculations: . . . 42

5 Related Work **43**
5.1 Approximate Nearest Neighbours Indices . . . 43
5.1.1 Clustering Based Indices . . . 44
5.1.2 Graph-Based Indices . . . 45

5.2 Predicate-agnostic Vector Search 46
5.3 Specialized indices for predicates on low cardinality attributes 47
5.4 Native Vector Indices in DBMS 48

6 Conclusion **49**

References **50**

List of Figures

2.1 HNSW Index (replicated from reference [26]). 7

2.2 Filtered Vector Search Strategies 10

2.3 Pictorial depiction of different correlations S can have with v_Q's neighborhood. Edges between vectors in HNSW are omitted. (replicated from reference [38]) 11

3.1 NaviX construct plan. 15

3.2 NaviX search plan. 16

3.3 Space of heuristics in a modified search algorithm. 17

3.4 Very low selectivity S can disconnect HNSW. 18

3.5 Pictorial descriptions of fixed heuristics. Vecors with + signs are selected. Patterns identify explored vectors. `onehop-s` only explores selected 1st degree vectors. `directed` explores all 1st degree but aims to explore the 2nd degree vectors that are closer to v_Q. Blind explores selected 1st degree vectors and an arbitrary subset of 2nd degree ones. 19

3.6 Selectivity regions where different fixed heuristics are optimal and the threshold formulas we use to switch between them. 21

3.7 `adaptive-local`'s decision tree to pick a heuristic. `adaptive-g` uses the same tree using σ_g instead of σ_l. 21

4.1 Wiki Schema 24

4.2 Vector search time vs Selectivity for different heuristics within 95% to 95.5% recall 28

4.3 `t-dc` vs `s-dc` metrics of `blind` and `directed`. 29

4.4 Vector search time vs selectivity for NaviX and `adaptive-g` for Wiki, Tiny, Arxiv, and GIST uncorrelated workloads. 31
4.5 Vector search time vs selectivity for NaviX and `adaptive-g` for Wiki correlated workloads. 31
4.6 Wiki Negatively Correlated heuristics call distribution. 32
4.7 Execution Time vs Selectivity for prefiltering baselines with >95% recall and within 1% of each other 33
4.8 Execution Time vs Selectivity for postfiltering baselines with >95% recall and within 1% of each other 35
4.9 Execution Time vs Selectivity for two-hop join postfiltering baselines with >95% recall and within 1% of each other 37
4.10 Vector search time (ms) vs selectivity for NaviX and Acorn-δ. 38
4.11 Distance Computations vs selectivity for NaviX and Acorn-δ. 39
4.12 Time (ms) spent on scanning neighbors vs selectivity for NaviX and Acorn-δ. 40
4.13 Vector Search Time with and without zero-copy optimization for 95% recall 42

5.1 Pictorial representation of clustering-based indices where Ci represents a cluster in n-dimensional space 44
5.2 Proximity graph where edge (C,A) is dropped and (C,X) is created due to the RNG property 45

List of Tables

4.1 Datasets 23

4.2 Average correlation ratios of uncorrelated workload on each dataset. 25

4.3 Average correlation ratios of for negatively and positively correlated workloads on Wiki dataset. 25

4.4 Indexing Time (mins). All systems use 32 threads except for PGVectorScale and VBase, which construct indices single threaded. 27

4.5 Indexing Time (mins) and Avg Index Graph Degree for all configurations of Acorn-δ and NaviX used in this evaluation. 40

Chapter 1

Introduction

Many modern applications process high-dimensional vector embeddings of objects, such as images or text. Consider as an example large language model-based (LLM) question and answering (Q&A) systems in enterprises. To answer a natural language question NL_Q that requires private, i.e., enterprise-level, knowledge, these Q&A systems need to provide LLMs additional information obtained from private documents. A common approach to provide this information to LLMs is retrieval augmented generation (RAG) [23]. RAG-based Q&A systems embed the chunks of documents in a high-dimensional vector space. Then, they embed NL_Q into the same space as a vector v_Q and find chunks whose embeddings are close to v_Q. These chunks are then given to an LLM to generate an answer. Other techniques, such as *graph RAG* [20, 62, 41], further connect these chunks with structured records, such as a knowledge graph, and retrieve chunks based on a mix of these connections and a search in the embedding space.

A core querying capability these applications require is a vector index [26, 11, 18], which can find the *k-nearest neighbors* (kNN) of a query vector v_Q in a set of vectors V. In addition to kNN queries, many applications also require performing other standard database querying on their objects, such as filtering them based on other attributes. Consider an e-commerce recommendation system that recommends products with a price range as well as similarity to another product's image. Image similarity can be solved by a kNN search on the vector representations of product images, while filtering on price can be an attribute filter [58, 39]. Since existing DBMSs already implement advanced querying and storage capabilities, there is immense value in implementing a vector index in existing DBMSs. This allows users to use a single system to build applications that require a mix of regular record querying and vector search, without the need for an additional specialized vector database.

Implementing a vector index in an existing DBMS also has benefits for system implementers, who can leverage the core capabilities of an existing system to implement the index, such as the persistent storage structures, query processor, or the buffer manager. This thesis presents *NaviX*, a **Na**tive **v**ector **i**nde**X** for graph DBMSs (GDBMSs) that has two main design goals. First, we aim to implement a disk-based vector index that leverages the core storage and query processing capabilities of the underlying GDBMS. Second, we aim to efficiently evaluate *predicate-agnostic kNN queries*, i.e., kNN queries over arbitrary subsets S of vectors V, that performs robustly under queries that contain both attribute filtering and joins, and whose selected subsets have different selectivities and correlations to the query vector v_Q. We implemented NaviX in Kùzu [9], which is a modern columnar GDBMS that adopts the principles of analytical DBMSs.

NaviX adopts the *hierarchical navigable small worlds* (HNSW) vector index design [26]. In HNSW indices, the index itself is a multi-layered graph (henceforth *HNSW graph*). The lower layer of the HNSW graph contains all vectors V and a set of edges E_H that connect close pairs of vectors. The higher layers contain progressively fewer nodes and edges and are used to find good "entry vectors" in the lowest level that are close to v_Q. With a slight abuse of notation, we refer to the HNSW index simply as $G_H(V, E_H)$ ignoring the upper layers. kNNs of v_Q are found by an iterative search algorithm on G_H. The algorithm starts from an entry vector v_e. For each neighbor w of v_e, it computes the distance of w to v_Q and puts w into a `candidates` priority queue based on this distance. At each iteration, the candidate c_{min} closest to v_Q is extracted from the priority queue and its neighborhood is explored, until the closest k vectors that have been seen so far stop improving.

GDBMSs are uniquely positioned as suitable DBMSs for implementing HNSW-based indices because they already contain specialized disk-based graph storage structures as well as graph-optimized querying capabilities. NaviX stores the lower layer of G_H as a relationship table, which are stored in Kùzu as compressed sparse row-based graph structures on disk. During kNN search, all accesses to the lower layer happens through the buffer manager. There is also a higher layer that is also persisted and is cached in memory.

To motivate our approach to evaluating predicate-agnostic kNN queries, consider a graph database of `Chunk` nodes, `Person` nodes, and `Mention` edges. Suppose `Chunk` nodes store chunks of text documents and their embeddings in an `embedding` property. `Person` nodes have `name` properties and there is a `Mentions` edge from a `Chunk` node to a `Person` node if the text of the chunk mentions a person. Suppose that a user has created a NaviX index called `ChunkHNSWIndex` on the `embedding` properties of `Chunk` nodes. Consider the following Cypher query:

```
MATCH (a:Person)<-[m:Mentions]-(b:Chunk)
WHERE a.name = "Alice"
PROJECT GRAPH AliceChunks(b);
CALL QUERY_HNSW_INDEX(AliceChunks, 'ChunkHNSWIndex', k=100,
                      q=[0.1, 0.4, ..., 0.8])
RETURN b, _rank
```

The query asks for 100 nearest neighbors of v_Q [0.1, 0.4, ..., 0.8] only across `Chunks` that mention `Person` nodes with name "Alice". We refer to the part of the query that selects the subset S of vectors among which the kNNs must be found as the *selection subquery*. The selection subquery above consists of the MATCH, WHERE, and PROJECT GRAPH clauses. In this work, we aim to support arbitrary, i.e., predicate-agnostic, selection subqueries, which contrasts with several prior work that construct specialized indices that can answer kNN queries mixed with a priori-known predicates [12, 5, 60, 30, 65, 61] that are highly restricted in their structures, such as containing range queries on attributes with small domains.

There are two broad approaches to perform predicate-agnostic kNN search. *Prefiltering* approaches [38, 29, 55, 58] compute the subset S first and then pass S to the kNN search algorithm, which finds the kNNs of v_Q only within S. *Postfiltering* approaches [64, 63, 28] continuously find and stream vectors in the original V from nearest to furthest to v_Q, and check if each streamed vector is in S, until k such vectors are found. These approaches offer different tradeoffs about the time they spend on preprocessing vs on kNN search. NaviX adopts the prefiltering approach. The core challenge of prefiltering approaches is that the original index is constructed on V and not S. Therefore, the search algorithm is performed over nodes that are not in S, which can lead the search in wrong directions and the algorithm may even end up exploring fewer than k vectors in S.

We next study the problem of *how to design a prefiltering-based search algorithm* that is efficient when evaluating queries whose selection subqueries have two different properties: (i) they have different *selectivity levels*; and (ii) they generate S with different *correlations* to v_Q's close neighborhood in G_H [38]. In uncorrelated queries, vectors in S are uniformly selected from V, so the chances of nodes in S being v_Q's nearest neighbors in G_H is roughly the same as the "global" selectivity of S in V. In positively and negatively correlated queries, vectors in S are, respectively, more and less likely to be in v_Q's nearest neighbors in G_H. Naive HNSW search algorithm, which explores all selected and unselected neighbors of a candidate can be very inefficient because it can explore very few selected nodes, or lead the search in wrong directions away from selected regions in G_H.

We propose an adaptive algorithm that is based on observing the behaviors of a space

of heuristics that choose which nodes to explore, i.e., compute distances to and put into candidates priority queue, during kNN search. Our space is based on the observation that prefiltering approaches need to make two main decisions when exploring the neighborhood of each candidate vector c_{min}:

1. *How much of the neighborhood of c_{min} to explore:* Here two main choices are: (i) to explore only the 1st degree neighbors of c_{min} as in the original kNN search algorithm; and (ii) to explore higher degree neighbors, until some predefined number of nodes in S are explored. This has been proposed in some prior works [**?**] and often 2nd degree neighbors is enough.
2. *The order to explore 2nd degree neighbors:* If the algorithm explores 2nd degree neighbors of c_{min}, then the order in which they are explored is important. We identify two choices: (i) the *blind* order, which is used in prior solutions [38, 57], follows the arbitrary order in which the 1st degree neighbors are scanned; (ii) the *directed* approach, which we propose in this work, orders 1st degree neighbors according to their distance to v_Q and explores 2nd degree neighbors in this order. The directed approach has the advantage that it can better direct the search towards regions that are closer to v_Q. However, it also pays the upfront cost of computing the distances to all 1st degree neighbors of c_{min}.

We show that each heuristics has a range of different selectivity (i.e., $|S|/|V|$) ranges within which they outperform others. Specifically, at high degree selectivities the 1st degree heuristic, at medium selectivities, the directed heuristic, and at very low selectivities the blind heuristic outperform others. Since in a prefiltering approach, the kNN search algorithm knows the selectivity of S a priori, one can design a *global-adaptive* heuristic that picks an appropriate heuristic after S is computed to get the best of all worlds across this space of heuristics. We further improve the global-adaptive algorithm by using the (local) selectivity of the neighborhood of each candidate c_{min} to make even more nuanced adaptive decisions. We call this the *local-adaptive* heuristic and propose it as a robust prefiltering algorithm to evaluate predicate-agnostic kNN queries.

Aside from our design of NaviX and our local-adaptive predicate-agnostic search algorithm, we present the details of our implementation, highlighting the core components of the underlying GDBMS we were able to re-use, such as disk-based columnar storage structures, operators, node semimasks, and an *in-buffer manager distance computation* optimization that computes distance functions directly on vectors in buffer manager frames. We present extensive experiments evaluating NaviX against several baselines, including specialized vector databases, such as Weaviate [57] and Milvus [29], which implement the prefiltering approach and PGVectorScale [47] and VBase [64], which implement the post-filtering approach. We show that NaviX generally outperforms these baselines in terms of latency when the search algorithms are configured to achieve similar recalls. Therefore,

our work demonstrates that it is possible build a vector index inside an existing DBMS whose performance is competitive with specialized systems. Our experiments also allow us to explain the general pros and cons of pre- and postfiltering approaches on a wider set of selection subqueries than prior work. Specifically, our selection subqueries include not only simple predicate filters on vectors but also join queries. Our observations from our evaluations and implementation details can be of independent interest to readers.

Chapter 2

Background

We first describe some basic terminologies. Then, give a background on HNSW indices and the problem of predicate agnostic search. Then we cover the necessary background on Kùzu, focusing on the components of the system we used in our implementation of NaviX.

2.1 Basic Terminologies

1. Vector dataset: A vector dataset $V = v_0, v_1, ..., v_{n-1}$ has each element v_i as a vector $x_i = [x_{i0}, x_{i1}, ..., x_{id-1}]$ of d dimensions.

2. Distance: The distance between two vectors x and y in V, denoted as $dist(x, y)$, can be computed using several distance functions. The most commonly used among these is the COSINE distance.

3. kNNS: Given a vector set V and a query point v_q, k-nearest neighbor search ($kNNS(V, v_q, k)$) identifies a set of k vectors $K \subseteq V$ that have the smallest distances to v_q. The approximate kNNS relaxes this constraint, allowing vectors in K to approximately satisfy this condition. Unless explicitly stated otherwise, we use kNNS to refer to approximate kNNS in this thesis.

4. Recall: For a vector set V and query point v_q, let K be the result of exact kNNS and K' be the result of approximate kNNS. The $k@k'$ recall for query v_q is defined as $\frac{|K \cap K'|}{|K|}$. This is the most common metric used to measure the accuracy of the index [18, 26, 38, 58, 45].

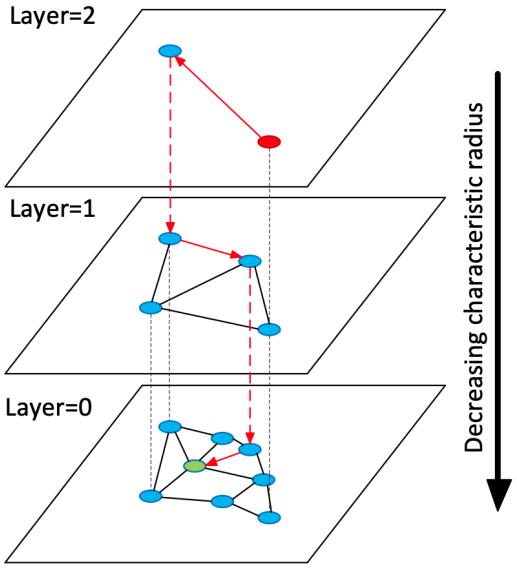


Figure 2.1: HNSW Index (replicated from reference [26]).

2.2 HNSW

HNSW [26] falls under the class of spatial indices called *approximate proximity graphs*. Similar to other spatial indices, these indices answer kNN queries for different values of k. Let V be the set of vectors to index and $dist(u, v)$ be a distance function in a vector space. In proximity graph indices, the index is a graph $G_H(V, E_H)$. Each vector $v \in V$ is represented as a node and an edge (u, v) represents closeness of u and v according to the distance function $dist$. Finding kNNs of a *query vector* v_q in proximity graphs involves a search algorithm in G.[1]

HNSW Construction: Figure 2.1 shows the overall structure of an HNSW index. HNSW indices have multiple levels. The lowest level contains all the vectors and the higher levels progressively contain fewer vectors. The construction algorithm is configured with a parameter M that determines the maximum number of neighbors of each vector, which for simplicity we assume is the same at each level. The index is constructed by inserting vectors one at a time. For each vector v, the algorithm first determines the maximum layer

[1]HNSW can be thought of as a relaxation of the *sa-tree* [33] proximity graph index, which is an exact kNN index. We highly recommend this for readers interested in the core ideas and intuitions behind HNSW.

```
Input: dataset V, efConstruction, M
Output: HNSW index graph G
v_e ← first element of V
G ← v_e
maxLevel ← randomLevel(v_e)
for each element v in V \ {v_e}:
    L(v) ← randomLevel(v)
    currEP ← v_e
    for (ly ∈ maxLeveltoL(v) + 1):
        currEP ← GreedySearch(v, currEP, 1, ly)
    for (ly ∈ min(maxLevel, L(v))to0):
        candidates ← SearchLayer(v, currEntry, efConstruction, ly)
        selectedNeighbors ← SelectNeighbors(v, candidates, M)
        // Prune if more than M neighbors are found
        if (|selectedNeighbors| > M):
            selectedNeighbors ← RNGShrink(selectedNeighbors, M)
        // Add forward edges and shrink
        AddEdgesAndShrink(G, v, selectedNeighbors, ly)
        for each neighbor n in selectedNeighbors:
            // Add backward edges and shrink
            AddEdgesAndShrink(G, n, v, ly)
    if L(v) > maxLevel:
        v_e ← v
        maxLevel ← L(v)
return G
```

Listing 2.1: HNSW index creation algorithm

(l) to insert v into. Then from the top-most level, it first finds an entry point v_{el} to level l by finding the closest vector to v. Then within level l it finds the $efConstruction$ (efC) NNs $w_1, ..., w_{efC}$ of v using the search algorithm we describe below (and starting from v_{el}). Then, if efC is greater than M, it prunes the found neighbors to M using an algorithm called Relative Neighborhood Graph (hereafter RNG) [49]. The key idea behind RNG is to remove the longest-distance edge in any triangular formation. This helps to create diverse connections i.e. both long- and short-range edges necessary for optimal navigation through the graph. After pruning, edges are inserted both from v to w_i and vice versa. If adding an edge to w_i increases w_i's neighbors to $b > M$, w_i's neighbors are pruned using the same algorithm [49]. Let $u_1, ..., u_b$ be the neighbors of w_i in increasing closeness to w_i. Each u_j is kept if it is closer to w_i than any of the previously kept $u_{t<j}$. Then v is inserted into each level below l in the same manner. Algorithm 2.1 shows the high-level pseudocode.

HNSW search algorithm performs a depth-first search-like traversal algorithm in each level of the index. Algorithm 2.2 shows the pseudocode of the search in a particular level.

```
Input: Query vector v_q, k, entry point v_e, candidate size efs
Output: k nearest vectors
min priority queue C ← {v_e} // candidates
max priority queue R ← {v_e} // results
visited set V ← {v_e}
while(C ≠ ∅):
    c_min ← Pop-Min(C)
    r_max ← |R|< efs ? Peek-Max(R) : ∞
    if (d(v_q, c_min) > d(v_q, r_max)): break; // convergence criterion
    for (n ∈ neighbours(c_min, l)): // neighbors in layer l
        if (n ∉ V):
            V ← V ∪ {n}
            if (|R|< efs OR d(v_q, n) < d(v_q, r_max)):
                Insert(C, n)
                Insert(R, n)
                if (|R|> efs): Pop-Max(R)
return closest k vectors in R
```

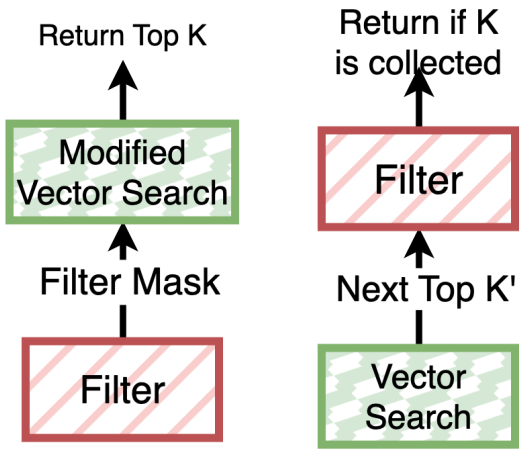
Listing 2.2: HNSW search algorithm in a partilar layer l.

The inputs are v_q, k, an entry vector v_e, and an efs value (explained momentarily). The output is the approximate k NNs of v_q in a particular level. The algorithm uses two priority queues: (i) a min priority queue of *candidates*, which represent vectors whose neighbors have not been explored; and (ii) a max priority queue of *results*, which store the efs closest vectors seen so far. At each iteration, the algorithm iteratively explores the closest candidates' neighbors (c_{min} in Algorithm 2.2) on line 10, starting from v_e. For each neighbor n, if n has not been visited already, $dist(v_q, n)$ is computed and put into the candidates queue. If $dist(v_q, n)$ improves the closest vectors seen so far, it is also put into results. The iterations stop when a c_{min} has a distance larger than the efs'th closest vector in results (line 9). The algorithm returns the k closest vectors in results.

The full algorithm uses this subroutine at each level, starting from the top level index. At each level $i+1$ except the lowest level, it sets $k = 1$ and $efs = 1$ to find the closest neighbor v_{ei} of v_q, which is used as an entry point for the search at level i. At the very top level the search starts from a fixed fixed entry point v_e^*. At the lowest level, the algorithm finds k NNs using some efs value greater or equal to k. The efs parameter controls the trade-off between search accuracy and latency. The higher the efs value, the longer it will take for the search to converge, but since a larger candidate set is considered, the higher will be the recall.

In HNSW search algorithm the dominant cost is the distance computations, which require expensive floating point operations on vectors. The distance computations are

done when a neighbor w of a c_{min} is explored and put into the candidates queue. When we discuss modified versions of this search algorithm, we will focus on minimizing the distance calculations.



(a) Prefiltering (b) Postfiltering

Figure 2.2: Filtered Vector Search Strategies

2.3 Predicate Agnostic Filtered Search, Prefiltering & Postfiltering

We briefly discuss the challenge of predicate-agnostic vector search, which is the problem of finding kNNs of v_Q over an arbitrary subset S of the vectors, that are selected by a selection subquery Q_S. In this thesis we focus on the *prefiltering approach* (Figure 2.2a) to evaluate predicate-agnostic search. In this approach, the system first evaluates Q_S and computes S. Then, the entire S is passed to the HNSW search algorithm which finds the approximate k nearest neighbors of v_Q in S. This is the approach adopted by some other systems, such as Weaviate [56] and Acorn [38]. The core challenge of prefiltering is that the search is performed on G_H, which was constructed by iteratively connecting each vector v with their closest M neighbors in V and not S. As a result v's neighbors in G_H that are actually in S can be very sparse or v can even be disconnected from other vectors in S.

An alternative to prefiltering is *postfiltering approach* (Figure 2.2b), which runs the unmodified HNSW search algorithm on G but finds a growing number of $k' > k$ NNs to v_Q. Then, for each neighbor w found, it checks if w is in S until k such vectors are found.

Several systems, such as PGVectorScale [47] and VBase [64], choose this approach. We compare NaviX's performance against these systems in the evaluation section.

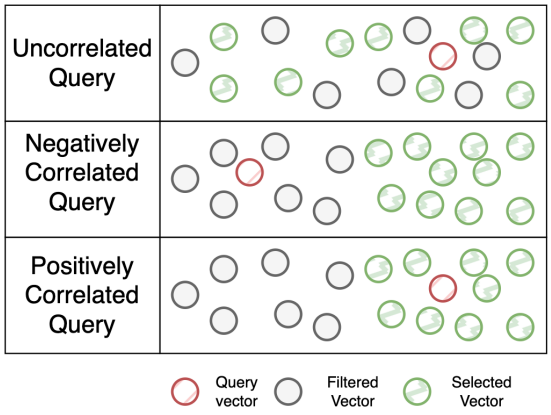


Figure 2.3: Pictorial depiction of different correlations S can have with v_Q's neighborhood. Edges between vectors in HNSW are omitted. (replicated from reference [38])

2.4 Query Correlation

Next, we explore query correlation, which significantly affects search performance and accuracy during the predicate-agnostic filtered search. Consider a dataset with vectors V and its filtered subset S. We can classify correlation in three cases: *uncorrelated*, where vectors in S are uniformly distributed relative to the query vector in high-dimensional space (with no clusters), *positively correlated*, where S forms a cluster containing the query vector, and *negatively correlated*, where the query vector is far from an S cluster in high-dimensional space. Figure 2.3 illustrates these three correlation cases.

Query correlation significantly impacts the search performance, particularly when the vector search algorithm is unaware of the filtering (i.e., in postfiltering case). For instance, with negatively correlated queries, the search algorithm must traverse more paths to reach the nearest neighbour target region within the filtered subset S.

In the evaluation section, we will demonstrate the limitations of existing approaches across different selectivities and query correlations, and show how NaviX performs more robustly and efficiently in comparison.

2.5 Kùzu Overview

Kùzu [9] is a GDBMS that adopts many of the architectural principles of analytical DBMSs, such as adopting disk-based columnar storage structures to store the node and relationship records, vectorized query processor [1], and morsel-driven multi-core parallelism. We refer readers to references [9, 14] for a detailed description of Kùzu's design. Here we provide background on the components that are needed to explain the design and implementation of NaviX. Other background is provided in Section 3 when describing different components.

2.5.1 Storage Structures

Node records in Kùzu are stored using a native design that is based on other columnar designs such as Parquet [37]. The storage of relationship records are stored in disk-based compressed sparse row (CSR) structures. This is a highly optimized persistent graph topology design and an example advantage of leveraging the exiting capabilities of GDBMSs to implement HNSW-based vector indices. Briefly Kùzu partitions the adjacency lists to *node groups*, which are 128K many adjacency lists. These lists form a unit and are compressed and stored consecutively on disk using the CSR format.

2.5.2 Node Semimasks

Query plans in Kùzu are composed into several subplans $SP_1, ..., SP_t$. Subplans form a directed acyclic graph and are executed one after another. Outputs of one subplan is consumed by the next subplan. Kùzu frequently uses sideways information passing (SIP) by passing *node semimasks* from one subplan SP_i to another SP_j. Node semimasks identify a subset of the nodes whose properties or relationships need to be scanned in SP_j. As we describe in Section 3.2, we use node semimasks to pass the results of selection subquery Q_S to the subplan that contains the vector search operator.

Chapter 3

System Design and Implementation

We next describe the design and implementation of NaviX. We first present index creation in Section 3.1 covering the query syntax, query plan pipeline, and storage structures used to create the index. Section 3.2 covers the vector search query syntax and query plan pipeline. Then Section 3.3 describes NaviX's search algorithm to perform predicate-agnostic vector search.

3.1 Index Creation

Index creation is executed as a standalone `CALL` function in Cypher. `CALL` functions are similar to the 'PRAGMA' functions in some RDBMSs. NaviX supports creating an index on node records. Suppose there is a node table with schema 'Chunk(id UINT64, docID UINT64, embedding FLOAT[1024])', which represent chunks, e.g., paragraphs, whose contents are embedded as 1024-dimensional vectors. Below is an example query creating an index called `ChunkHNSWIndex` on the `embedding` property of `Chunk` nodes.

```
CALL CREATE_HNSW_INDEX('ChunksHNSWIndex', 'Chunks', 'embedding', M_U)
```

NaviX is a 2-level HNSW implementation. M_U is the maximum degree of vectors in the lower level index. Note that being 2-level is not a strict design choice as NaviX can easily be extended to construct more levels. However, as observed by several prior works [32, 25] and as we show in our evaluation, 2 levels perform well on high-dimensional datasets. Figure 3.2 shows the plan that constructs the index on the `Chunk` nodes' `embedding` property. The first subplan, which consists of `Scan Node` and `HNSW Construct` operators, implements

an HNSW construction algorithm using morsel-driven parallelism [21], which is Kùzu's approach to executing plans in parallel.

Before the subplan starts, two in memory CSR data structures G_L (for lower level) and G_U (for upper level) are initiated. The sizes of these CSRs are respectively, n and $n * s$, where n is the number of nodes in the indexed node table and s is sampling rate for G_U (by default 5%). G_U and G_L are initiated with M_U and $M_L = M_U/2$ pre-allocated edges for each node. In morsel-driven parallelism, threads create copies of plans and in parallel grab small chunks of inputs called morsels from the leaf scan operator and execute the rest of the operators in parallel on each morsel until no morsels remain. In the first subplan, shown in Figure 3.2, each thread concurrently scans a morsel of vectors (2048 many) from disk and updates the shared CSR structures as follows. Let thread T_i be responsible for inserting vector v into the index. Given the latest state of $G_{L/U}$, T_i executes the HNSW search algorithm (Algorithm 2.2) to find the $M_{L/U}$ closest neighbors of v. This search happens in lock-free manner using C++ atomics and $G_{L/U}$ might be modified by other threads as the search algorithm runs. However HNSW is already an approximate index and our evaluations demonstrate that the index quality can tolerate this possible data race to benefit from better parallelism and we recommend this optimization to developers. Then edges to $M_{L/U}$ neighbors of v are inserted twice, pruning any adjacency lists if necessary. Unlike the search algorithm, writes are synchronized by acquiring locks on the adjacency lists of each vertex. We note that only G_U and G_L are kept in memory. All accesses to the vectors happen through the buffer manager (also in index searching), which is critical to increase the scalability of the index as vectors are often much larger than the actual HNSW index. For example, on our Wiki dataset, vectors take 63GB while NaviX only takes 7.8GB.

The second subplan persists G_L on disk. It scans the constructed in memory HNSW graph and connects it with Kùzu's existing `Node Group CSR Construct` operator. This operator takes in a node group (by default 128K) many node's relationships and converts them into Kùzu's disk-based CSR structure. G_U could also be persisted using Kùzu's existing disk-based structure. However, because it is intended to be very small, we persist it using a simpler format that writes the offsets and edges consecutively on disk.

Observe that as per our first design goal, we leverage many of the existing capabilities of the underlying GDBMS, including its storage structures, which are automatically compressed on disk by Kùzu, its automatic query parallelization mechanism, and many of its operators, such as scans, Node Masker (next subsection), and CSR Construct. We did not modify any of these core components. This is an important advantage that simplifies the engineering efforts of system developers. However, there were also components we modified, such as the transaction manager, for which we added a new record type to

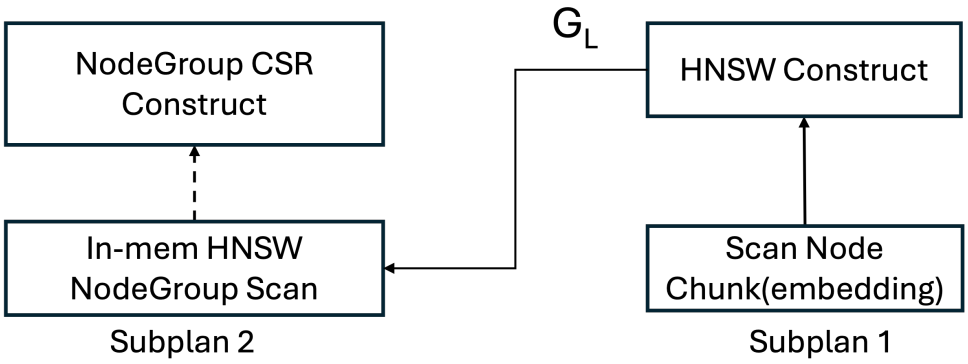


Figure 3.1: NaviX construct plan.

the write-ahead log of the system and the associated replay function to atomically serialize the lower layer of the index. We also added a new entry to the catalog to store metadata about the index, e.g., which CSRs on disk belong to the index's lower layer.

3.2 Index Search Query Syntax and Plan

To describe the index search query syntax, we replicate our query from Section **??**:

```
MATCH (a:Person)-[:Mentions]->(b:Chunk)
WHERE a.name = "Alice"
PROJECT GRAPH AliceNodes (b);
CALL QUERY_HNSW_INDEX(AliceGraph, 'ChunkHNSWIndex', k=100,
                      q=[0.1, 0.4, ..., 0.8])
RETURN q, b, _rank
```

Here, Q_S is the subquery specified by the MATCH and WHERE clauses and identify the `Chunk` nodes that are mentioned by a `Person` with name Alice. In Cypher, the PROJECT GRAPH clause is intended to run batch graph algorithms on a subgraph of nodes and edges, such as PageRank or connected components. We use it to specify only a subset of nodes (and not edges) that can be passed to QUERY_HNSW_INDEX function to find vectors within a subset of indexed nodes. Figure 3.2 shows the plan structure for evaluating predicate-agnostic search queries. The first subplan evaluates Q_S and uses Kùzu's `Node-Masker` operator to create a node semimask. This is then passed to the second subplan that starts with an `HNSW Search` operator that takes in the semimask and runs a modified HNSW algorithm (if the semi-mask is not null). All accesses to the

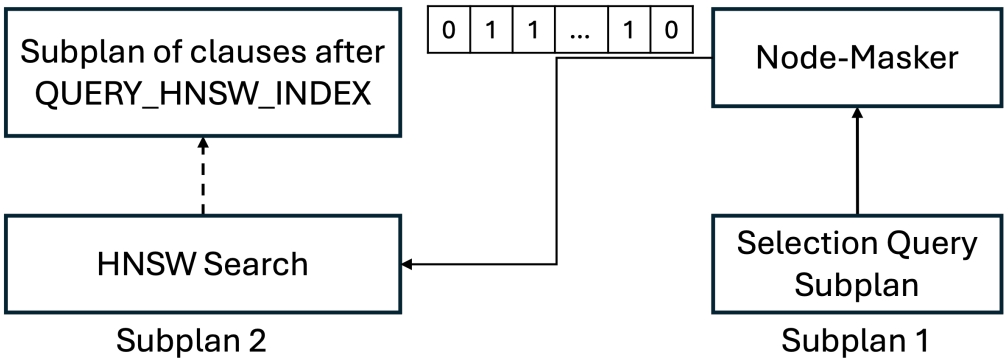


Figure 3.2: NaviX search plan.

persisted adjacency lists of vectors in the index and to the actual vectors happen through the buffer manager of the system.

3.2.1 In-Buffer Manager Distance Computations

We end this section with an optimization that may be of interest to system developers. As we discussed above, during both index construction and search, vectors are scanned through the buffer manager to improve the scalability of NaviX. Specifically, the algorithm scans the vectors of neighbors n_1, ..., n_ℓ of a candidate c_{min}, and runs distance $dist(n_1, v_Q)$. In the standard approach in DBMSs, each piece of data is first copied from the buffer manager's frames to an operator-local buffer. Then, the operator operates on this data. We observed that the copy operations are a performance bottleneck. To address this, we extended the storage manager interfaces of Kùzu to directly run a function on the data residing in buffer manager frames without performing any copies. Specifically, we pass a distance function to the buffer manager. The buffer manager finds and pins a frame, if necessary, scans the vector from disk to the frame, runs the function, and unpins the frame. Therefore the vectors never move to a local buffer of the HNSW search operator. In Section 4.6, we show that this optimization can improve vector search latencies by up to 1.6x. Finally, we used the SimSIMD [51] library to perform distance computations using SIMD instructions.

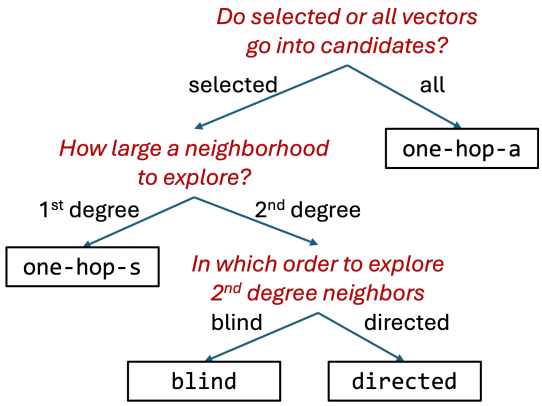


Figure 3.3: Space of heuristics in a modified search algorithm.

3.3 Adaptive-Local Search Algorithm

The primary advantage of prefiltering is that because it pays the upfront cost of computing Q_S entirely, it has full information about which vectors are in S. This information can be utilized during kNN search algorithm. In direct contrast, post-filtering approaches perform the search without any information about what is in S. We next discuss the question of how to use S to design a robust prefiltering algorithm that performs the search efficiently across different selectivity levels and correlation scenarios.

3.3.1 Design Space of Fixed Heuristics

We begin by outlining a set of fixed heuristic decisions a modified HNSW search algorithm can make and discuss the intuitions of the pros and cons of each. Then, informed with these analyses, we outline two adaptive search algorithms in Section 3.3.2. For reference, Figure 3.3 shows the decision tree that summarizes the space of heuristics we discuss. We emphasize that the core step of the original HNSW search algorithm is that at each iteration, the algorithm explores the *closest neighbors of a* c_{min} *in* V. Therefore we are interested in developing modified search algorithms that efficiently explore the *closest neighbors of a* c_{min} *in* S. Intuitively, heuristics that achieve this *primary goal* should perform the search more efficiently than others. For reference Figure 3.5 shows the heuristics we discuss (except the `onehop-a` heuristic).

The first decision is *if all or only selected vectors should be put into candidates queue*. The unmodified HNSW algorithm puts all vectors, which may be inefficient due to explor-

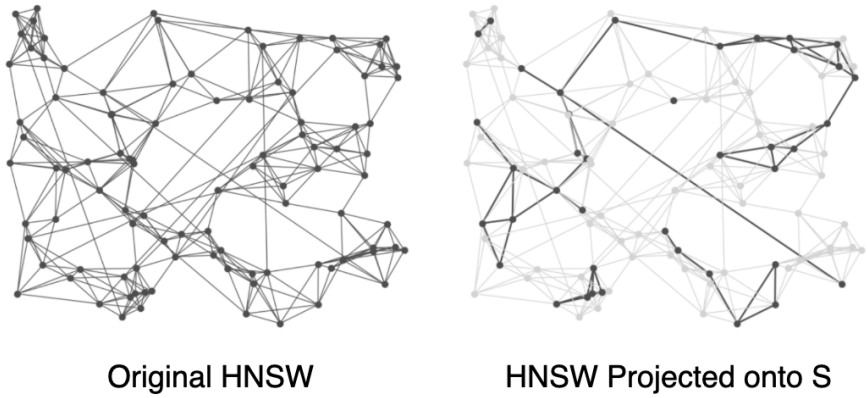


Figure 3.4: Very low selectivity S can disconnect HNSW.

ing unselected vectors. Further, exploring unselected vectors can misdirect the search away from regions that contain selected vectors, slowing down convergence. We refer to the original unmodified HNSW as the `onehop-a` heuristic (for one hop **a**ll vectors). An algorithm can instead decide to explore only selected vectors. We refer to this heuristic as `onehop-s`. Intuitively `onehop-s` would improve the convergence of `onehop-a` at high selectivity levels but creates a separate problem. Specifically, at low selectivity levels the *selected projection* of G_H, i.e., the subgraph that contains S and their edges, can be disconnected. Figure 3.4 shows this problem pictorially. Therefore, only exploring 1st degree selected neighbors of c_{min} can explore very few nodes in S, which lowers the recall of `onehop-s`.

The second decision is *how much of each candidate's neighborhood to explore in the original index G*. Here, to address the disconnected graph problem at lower selectivity levels, an algorithm can explore higher degree neighbors of each candidate. Specifically an algorithm can explore 2nd degree nodes, which we show is adequate in our evaluations. A version of this heuristic has been explored in ACORN [38]. Given a candidate c_{min} and its neighbors $n_1, ..., n_\ell$, ACORN's heuristic takes the first neighbor n_1 of c_{min} and explores the selected vectors among n_1 and n_1's neighbors $n_{11}, ..., n_{1k}$. This is then repeated for the second neighbor n_2 of c_{min}, until M many selected vectors are explored (or the algorithm exhausts all 2nd degree neighbors of c_{min}). We refer to this as the `blind` 2nd degree heuristic (`blind` for short). We note that in our evaluations, we implement and evaluate an improved version of this heuristic than the one used in reference [38]. Specifically, we first explore all 1st degree neighbors $n_1, ..., n_\ell$ and then start exploring 2nd degree neighbors. This slight version performs strictly better.

The third decision is *the order in which the 2nd degree neighbors are explored*. The

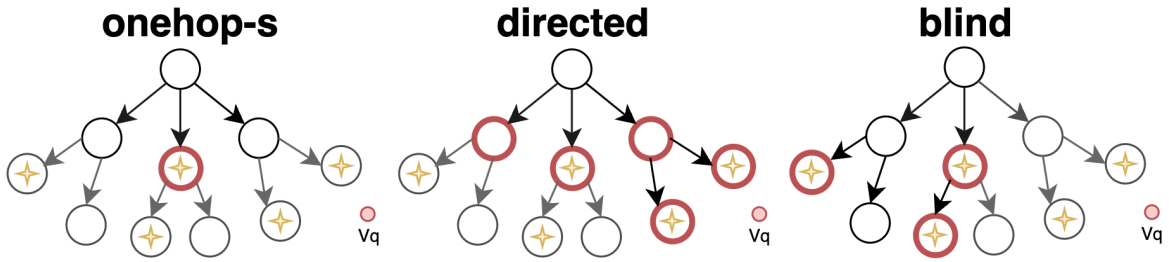


Figure 3.5: Pictorial descriptions of fixed heuristics. Vecors with + signs are selected. Patterns identify explored vectors. `onehop-s` only explores selected 1st degree vectors. `directed` explores all 1st degree but aims to explore the 2nd degree vectors that are closer to v_Q. Blind explores selected 1st degree vectors and an arbitrary subset of 2nd degree ones.

advantage of `blind` is that it does not require any distance calculations to implement. However, it is oblivious to how close each 2nd degree neighborhood it explores is to v_Q. In other words, it does not perform its exploration in a directed manner towards the regions that are closer to v_Q. Alternatively we propose exploring the neighbors of c_{min} in increasing order of their distance to v_Q. Let $n_1^*, ..., n_\ell^*$ be the neighbors of c_{min} ordered from closest to v_Q to furthest. We refer to the heuristic that explores the 2nd degree neighbors of c_{min} in this order as the `directed` 2nd degree heuristic (or `directed` for short). `directed` has the advantage of prioritizing the search towards regions that are closer to v_Q but incurs the cost of computing the distances to each neighbor of c_{min}.

Within this space, except for `onehop-a`, different heuristics have different advantages in different selectivity regions. At high selectivity levels, we expect `onehop-s`, which limits explorations to selected 1st degree vectors to work well because there is enough selected vectors in S that these heuristics should achieve the primary goal we articulated above. As selectivity decreases, because there is eventually not enough vectors in S, `onehop-s` should degrade in recall. Therefore, heuristics that explore 2nd degree neighbors, such as `blind` and `directed` should work better. Between these two, `directed` should converge faster if we focus on the number of candidates it explores. However, `directed` incurs the additional cost of computing the distance of v_Q to every neighbor (selected or unselected) of c_{min}. The benefits of `directed` can overweigh its overheads at medium selectivity levels. Eventually however, as selectivity decreases enough, `directed` cannot have an advantage over `blind`. This is because at low-enough selectivity levels, there is not enough selected nodes even in the 2nd degree neighbors of c_{min}. Therefore both `directed` and `blind` effectively explore the same set of selected vectors, yet `blind` does not incur the overhead

of computing the distances to unselected 1st degree vectors. In our evaluations, we will verify these expectations both in terms of the search latencies of these heuristics as well as by studying the number of distance computations they perform.

3.3.2 Adaptive Algorithms

We next describe two adaptive algorithms that utilize the selectivity information in S to pick the best of all worlds across `blind`, `directed`, and `onehop-s`, respectively, across low, medium, and high selectivity levels. We first pick an upper bound threshold $ub_{onehop-s}$ above which we pick `onehop-s`. We found 50% to be a safe choice here although in some of our evaluations slightly lower thresholds also work. To decide between `blind` and `directed`, we use the following formula. Recall that under low-enough selectivity, if there are less than M selected vectors in the 1st and 2nd degree neighbors of c_{min}, `directed` has no particular advantage over `blind`. So it is guaranteed to be suboptimal. Therefore, we try to estimate if we are in this safe region.

Specifically, we calculate the **e**stimated number of **s**elected **v**ectors (esv) in the 1st and 2nd degree neighbors of c_{min} with the formula: $esv = \sigma \times (M + 1) \times M$, Let σ be the selectivity of S (more on this later). The formula multiplies the selectivity with the total sizes of the 1st and 2nd degree neighbors of c_{min}. If esn is less than M, which is the maximum number of selected nodes `blind` explores, then `directed` cannot have an advantage over `blind` by calculating the distances to each 1st degree node. Therefore we default to blind. This safe region is conservative, so in our implementation we are more lenient and compare esn with M^*lf, where lf (3 by default) is a leniency factor. Figure 3.6 pictorially shows the regions where we hypothesize each fixed heuristic to work well and Figure 3.7 summarizes the rules we use to select between the fixed heuristics.

We can configure the above adaptive algorithms in two different ways. First, we can use the global selectivity of S as σ, $\sigma_g = |S|/|V|$. We refer to this version of the algorithm as `adaptive-global` (`adaptive-g` for short). As we demonstrate in our evaluations, this algorithm is able to capture the best of all worlds. However, we can improve this algorithm, and even any fixed-heuristic in their optimal regions, by choosing a possibly different heuristic for each candidate c_{min}. This is useful if S correlates with certain regions in G and is not a random subset of V. Figure 2.3 pictorially shows the three different possibilities of S being uncorrelated, positively, or negatively correlated with the region around v_Q. Especially in correlated scenarios, even though the global selectivity of S is low enough that we are in the optimal (global) selectivity region for the `blind` heuristic, if c_{min} is in a region that is more likely to contain selected vectors, it might be useful to choose `onehop-s` heuristic for c_{min}.

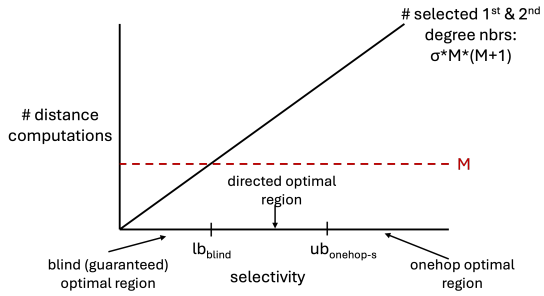


Figure 3.6: Selectivity regions where different fixed heuristics are optimal and the threshold formulas we use to switch between them.

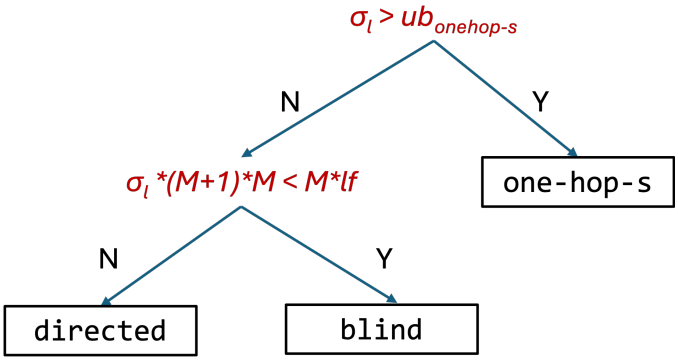


Figure 3.7: `adaptive-local`'s decision tree to pick a heuristic. `adaptive-g` uses the same tree using σ_g instead of σ_l.

We refer to the adaptive algorithm that uses the *local selectivity* of c_{min} to pick a heuristic as `adaptive-local`. Specifically, `adaptive-local` uses $\sigma_l = |S(nbrs(c_{min}))|/|nbrs(c_{min})$, where $nbrs$ (c_{min}) is the neighbors of c_{min} and $S(nbrs(c_{min}))$ is the selected neighbors of c_{min}. σ_l calculates the fraction of c_{min}'s neighbors that are selected. Note that σ_l can be calculated without any distance computations. In this thesis, we propose `adaptive-local` as a predicate-agnostic filtered search algorithm and demonstrate that it is efficient under both different selectivity levels and correlation scenarios.

Chapter 4

Evaluation

We next evaluate the performance of NaviX, which refers to our implementation that uses the `adaptive-local` search algorithm. Our experiments aim to answer two main questions. (1) How do the different prefiltering heuristics behave under different selectivities and query correlations? These experiments demonstrate the robustness and efficiency of the `adaptive-local` heuristic used by NaviX. (2) How does NaviX compare against other prefiltering and postfiltering approaches.

4.1 Experimental Setup

4.1.1 Baselines

We used the following baseline systems.

Kùzu configurations: We compare the performances of our own implementations of different heuristics in Kùzu. These are: (i) Kùzu-`onehop-s`; (ii) Kùzu-`blind`; (iii) Kùzu-`directed`; (iv) Kùzu-ag, which adopts the `adaptive-g` heuristic; and (v) NaviX, which adopts the `adaptive-local`heuristic.

Weaviate (v1.28.2) [57] and Milvus (v2.5.0) [29]: These systems serve as baselines of popular specialized vector databases. Weaviate also serves as baselines of an external implementation of some of the prefiltering heuristics we covered. Milvus instead performs a mix of pre- and postfiltering evaluation depending on the selectivity estimation.

PGVectorscale (v0.5.1) [47] and VBase [64]: These systems are two vector index implementations on Postgres that serve as baselines of vector indices that are implemented

Dataset	# Vectors	Dimension	Distance Fn
GIST [18]	1M	960	L2
Tiny [24]	5M	384	L2
Arxiv [40]	2.1M	384	Cosine
Wiki	15.4M	1024	Cosine

Table 4.1: Datasets

inside an existing DBMS. They also serve as baselines that implement postfiltering approaches.

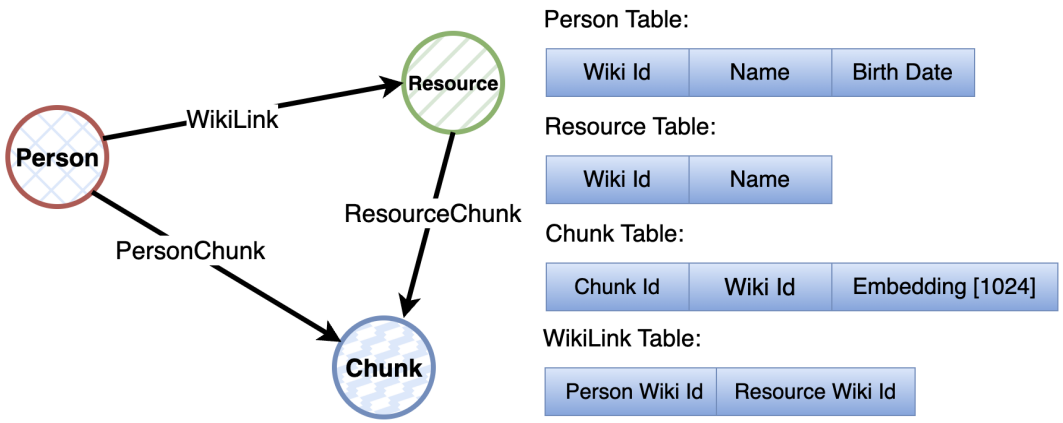
Except VBase, at very low selectivity levels (estimated or actual), all of these baselines have another heuristic, which is `brute-force`. This heuristic just computes vector search across ever possible vector in S and returns the kNNs with 100% accuracy. This will be relevant in our experiments in Section 4.4. Further, all of these baselines implement proximity-graph-based indices. Finally, we compared against Acorn [38], which also implements a proximity graph-based index but changes the index construction algorithm to make the graph denser unlike other baselines, which do not modify the original HNSW-based index. Despite using a denser graph, as we discuss in Section 5, we found Acorn to be less efficient in its search than NaviX. We note however we evaluate Acorn's `blind` heuristic both in Kùzu and Weaviate, which also adopts it in some selectivity ranges.

4.1.2 Datasets

We used the four datasets in Table 4.1.

GIST, Tiny, and Arxiv: GIST and Tiny are two datasets that contains embeddings of images that have been embedded using local GIST descriptors [34], which is a feature representation technique for images. GIST uses local INRIA Holidays images [17] and Tiny contains images from the TinyImages dataset [48]. GIST is one of the most popular datasets to evaluate vector indices [18, 10, 45]. Arxiv is a recent open-source dataset that embeds titles from the arXiv [3] paper dataset using the all-MiniLM-L6-v2 [50] text embedding model.

Wiki: The above datasets contain objects but no connections between objects. Therefore we can use them with predicate-agnostic queries where the selection subqueries contain simple filters but not joins. Moreover, these filters are uncorrelated with the query vectors (see Section 4.1.3). To extend our evaluation to selection subqueries with joins and different correlations, we prepared a new dataset from DBPedia latest version [8] and the



(a) LPG Schema (b) Relational Schema

Figure 4.1: Wiki Schema

Wikipedia dump [59]. DBPedia is an RDF graph of (subject, predicate, object) triples, such as (dbr:LionelMessi, dbo:wikiPageWikiLink, dbr:Camp_Nou), where subject and object "resources" have corresponding Wikipedia articles. The dataset schema is shown in Figure 4.1a. We describe it as a property graph database. Figure 4.1b also shows the corresponding relational schema.

- *Person(pID) nodes:* We modeled each resource with a dbo:birthPlace and dbo:birthDate predicate as a Person.
- *Resource(rID) nodes:* In the DBPedia graph, we do a 2-hop traversal around Persons along the dbo:wikiPageWikiLink predicates, which represent a link between the Wikipedia articles of resources. We model each resource we visit as a Resource node.
- *Chunk(cID, embd) nodes:* We chunked the Wikipedia articles of each Person and Resource node into chunks of 1028 tokens and created a Chunk node. We embedded the tokens of each chunk into 1024 dimensional vectors using the popular stella_en_400M-_v5 [42] model on Hugging Face [16] and stored it as an embedding property. Chunks are the objects that we index in a vector index in our baselines.
- *PersonChunk, ResourceChunk, and WikiLink relationships:* We connected each Person and Resource to the Chunk nodes of their corresponding Wikipedia articles, respectively, as PersonChunk and ResourceChunk relationships. We also added WikiLink relationships from Person nodes to their first degree Resource nodes.

Correlation	Uncorrelated									
Selectivity	**90%**	**75%**	**50%**	**40%**	**30%**	**20%**	**10%**	**5**	**3%**	**1%**
Wiki	0.98	0.98	0.98	0.98	0.99	0.98	1.00	1.02	1.12	1.12
Tiny	1.00	1.00	1.00	1.00	0.98	0.99	1.02	1.00	0.98	0.90
Arxiv	0.99	0.99	1.03	1.05	1.06	1.07	1.10	1.10	1.12	1.14
GIST	0.90	0.99	0.99	1.00	1.01	1.01	1.01	0.96	1.04	1.18

Table 4.2: Average correlation ratios of uncorrelated workload on each dataset.

Correlation	Negatively Correlated					Positively Correlated				
Selectivity	**22.9%**	**15%**	**9.9%**	**5.1%**	**1%**	**22.9%**	**15%**	**9.9%**	**5.1%**	**1%**
Wiki	0.055	0.050	0.055	0.064	0.037	2.65	2.90	2.64	2.57	2.90

Table 4.3: Average correlation ratios of for negatively and positively correlated workloads on Wiki dataset.

4.1.3 Query Workloads

Our workloads contain predicate-agnostic queries on our datasets. Each query Q consists of two main components: (i) Q_S is a selection subquery that selects a subset $|S|$ of the vectors from the entire indexed vectors V; and (ii) v_Q is a query vector. Q finds kNNs of v_Q within S. We refer to the global selectivity of Q_S as $\sigma = |S|/|V|$. In each Q, v_q can be "uncorrelated", "positively", or "negatively" correlated with S, which we define as follows. Let $\text{knn}_V^{v_Q}$ be the kNNs of v_Q in V. Let $\text{knn}_S^{v_Q} \subseteq \text{knn}_V^{v_Q}$ be the subset of these neighbors that are in S. We use $\sigma_{v_q} = \text{knn}_S^{v_Q}/\text{knn}_V^{v_Q}$ as a metric for the selectivity of v_Q's kNNs in V. Our correlation metric measures if σ_{v_q} and σ are correlated:

$$\text{ce} = \sigma_{v_q}/\sigma$$

If ce ≈ 1, then v_Q is uncorrelated with S because the close neighbors of v_Q in V are as likely to be in S as a random node in V. If ce $\gg 1$ (ce $\ll 1$), then v_Q is positively (negatively) correlated with S, as the neighbors of v_Q are more (less) likely to be in S than a random node.

Uncorrelated Workloads: For each dataset, our uncorrelated queries have a Q_S that contain a simple filter on the IDs of embedded objects. The structure of our uncorrelated queries in Cypher is as follows.

```
MATCH (c:Chunk)
WHERE c.cid < (MAX_CHUNK_ID * σ)
PROJECT GRAPH S(c);
CALL QUERY_HNSW_INDEX(S, HNSWIndex, v_Q, k)
RETURN c.cID;
```

We populate this template with different σ, v_Q, and k. For GIST, Tiny, and Arxiv, we randomly selected 50 queries from their own query sets. For Wiki, we generated 50 queries for uncorrelated and positively correlated cases as follows. We chose a random Person node p, sampled its chunks and chunks of connected Resource nodes, and used OpenAI's o1 model [36] to generate questions about p using these chunks as context. We provided chunks until reaching the 128K token limit. We embedded o1's question as v_Q also using stella_en_400M_v5. To get an uncorrelated S, we selected Chunks based solely on chunkIDs as in the above Cypher query, which filters chunks uniformly.

Wiki Positively Correlated Workload: We used the same v_Q's of the uncorrelated Wiki workload but changed Q_S to select a subset of Chunks that are from the Wikipedia articles of Person nodes, which we expect are more likely to be in the neighborhood of v_Q's than a random Chunk:

```
MATCH (p:Person)-[e:PersonChunk]->(c:Chunk)
WHERE p.birth_date >= {s_date} AND p.birth_date < {e_date}
PROJECT GRAPH S(c)
CALL QUERY_HNSW_INDEX(S, HNSWIndex, v_Q, k)
RETURN c.cID;
```

The selection subqueries are 1-hop queries that select a subset of Person nodes based on their birthdates and chunks that come from the Wikipedia articles of these people.

Wiki Negatively Correlated Workload: For Q_S, we use the 1-hop selection subqueries that select Chunks of Person nodes. For v_Q we prompt $o1$ to generate questions about non-people entities, such as cities, monuments, and companies.

For reference, Table 4.2 and 4.3 report the average correlations (ce) of these queries for different selectivity levels and correlated scenarios. Note that our selectivity levels go up to 100% in uncorrelated workloads as we can use predicates that select every object. For Wiki positively and negatively correlated workloads, selectivities are at most 22.9% because the 1-hop queries select Chunks that come from articles of Person nodes, which constitute 22.9% of all Chunks. In Section 4.4, where we compare prefiltering and postfiltering approaches, we also describe another 2-hop workload that we generated for Wiki.[1]

[1]See our repo [43] for questions generated by o1, our o1 prompts, and SQL versions of the selection

4.1.4 Evaluation Metric

For most experiments, we measure baseline latency on a dataset/workload under varying selectivity levels when searching for the k=100 nearest neighbors of v_Q. Since kNN is approximate, we target 95% recall and adjust the efs parameter to stay within 1% of it. If a baseline gives higher recall than 95% with a minimum possible efs then we match that for all other baselines. For each selectivity level, we use binary search on efs between 100 and 1000 per baseline to find the lowest value meeting this target. If a baseline fails to reach 95% recall even at efs 1000, we mark that point with a cross sign in our figures. VBase does not have any knobs to tune its recall, so we use it in its default configuration.

Systems	**GIST 1M**	**Arxiv 2.1M**	**Tiny 5M**	**Wiki 15.4M**
PGVectorScale	336.34	456.63	3153.52	5932.81
VBase	54.84	71.32	294.12	942.78
Weaviate	19.14	22.80	57.44	187.65
Milvus	19.85	23.66	50.15	182.42
NaviX	**5.13**	**7.09**	**18.25**	**55.22**

Table 4.4: Indexing Time (mins). All systems use 32 threads except for PGVectorScale and VBase, which construct indices single threaded.

4.1.5 Index Configurations

We built the HNSW indices with maximum connection M parameter set to 32 in upper layers and 64 in lowest layer. and efC set to 200 across all systems. We used Milvus in its single segment/partitioned configuration as every other baseline and NaviX also implements single partitioned HNSW implementations. Table 4.4 reports the index-building times when using 32 threads on our hardware (see below), except PGVectorScale and VBase which have single-threaded index construction implementation. Although our focus is not on index construction time, NaviX's construction times are always faster than baselines.

4.1.6 Other Experimental Setup Details

In Milvus, Weaviate and VBase, indexes and vectors are kept in memory. In NaviX and PGVectorScale, indexes and vectors are stored on disk and accessed through a buffer

subqueries we used.

manager. Each Q set contains 50 different v_Q and all numbers in our experiments are averages across these queries. For each query Q, we first warm up the buffer manager by running Q. Then we run each query 5 times and report the average latency of Q in our figures. Unless otherwise specified, all latency numbers measure end-to-end latency, i.e., including both the time to execute the selection subquery and kNN search. We ran our evaluation on a single x86 machine with 180 GB of memory and 32 v-CPUs using an Intel(R) Xeon(R) Platinum 8260 processor.

Our system is implemented in c++ and is available at: https://github.com/gaurav8297/kuzu

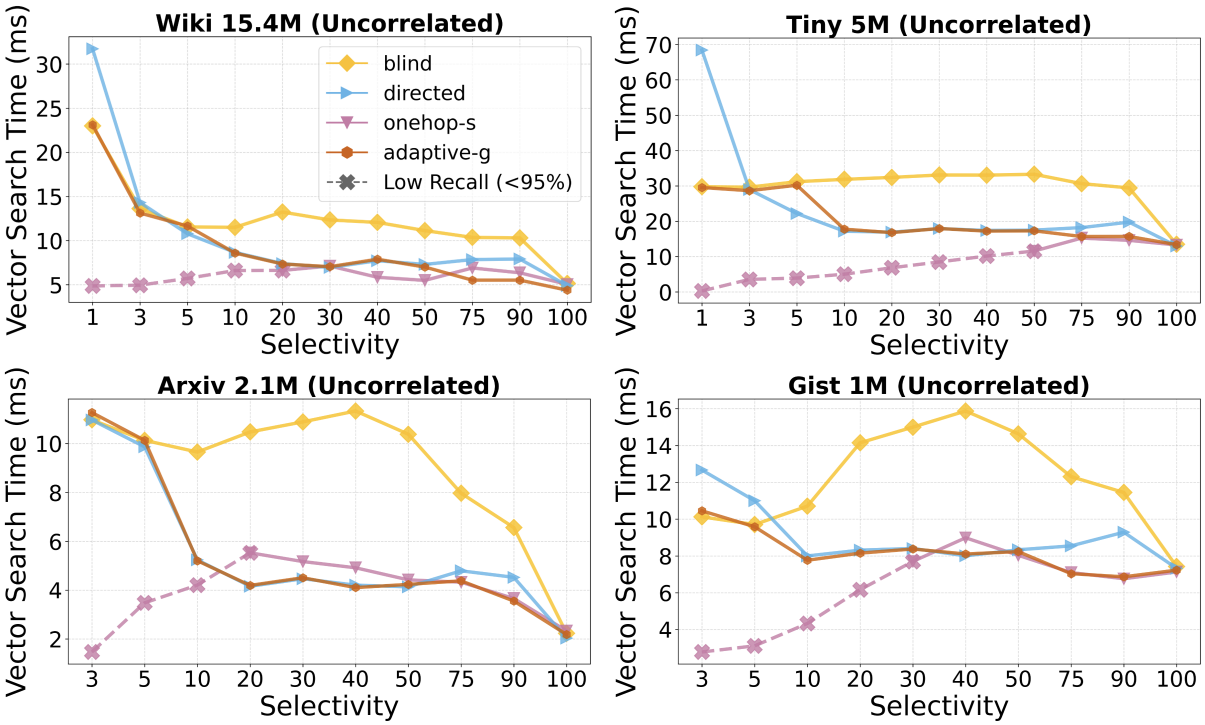


Figure 4.2: Vector search time vs Selectivity for different heuristics within 95% to 95.5% recall

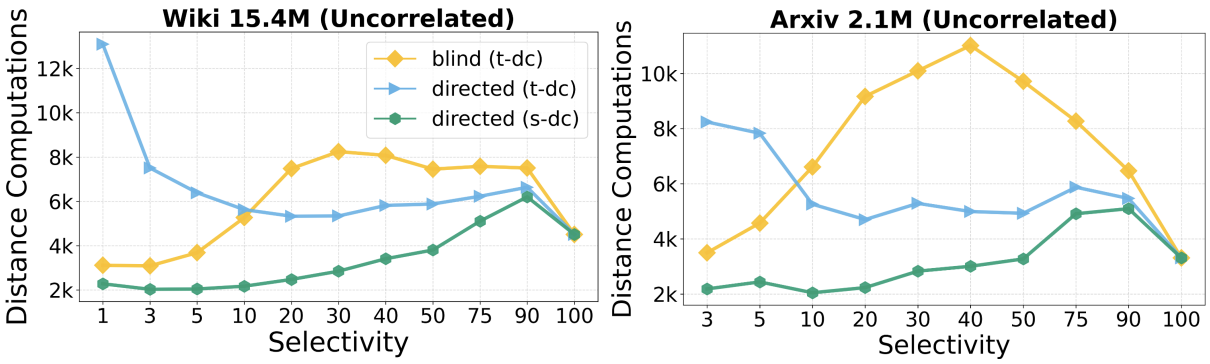


Figure 4.3: `t-dc` vs `s-dc` metrics of `blind` and `directed`.

4.2 Prefiltering Fixed Heuristics and Adaptive-G

We begin by studying the behaviors of different fixed prefiltering heuristics and `adaptive-g`. We used each dataset and their uncorrelated workloads. We extend these with NaviX, Milvus, and Weaviate in later sections. Since each of the Kùzu configurations have the same prefiltering cost for evaluating the selection subqueries, we only report vector search times. Figure 4.2 presents our results.

First, as we hypothesized in Section 3.3, across all datasets, `onehop-s` consistently has lower latency than other heuristics as it always limits its distance calculations to only the selected vectors in first degree neighbors of candidates. However, its recall drops significantly below 50% selectivity for Tiny5M and 30% for Wiki (Figure 4.2). Therefore, above 50% selectivity, `onehop-s` is the most performant of the fixed heuristics across all datasets.

Second, observe that at very high selectivity levels, `blind` and `directed` perform very similarly. This is because at very high selectivity levels, say 100%, since each first degree neighbor is selected, both directed and blind explore only the first degree neighbors. This is why their behavior is also similar to `onehop-s`. Then as selectivities decrease, `directed` starts outperforming `blind`. Once the selectivities are low enough, `blind` starts outperforming `directed`.

This behavior is expected and to explain it better, we measured two further metrics. First, we measure the number of distance calculations these heuristics perform across the selected vectors (`s-dc`). This number is equivalent to the number of vectors a heuristic puts into the candidates priority queue. We take this as a measure of how *effective* the search

is. Assuming different heuristics achieve similar recalls, more "effective" search heuristics explore a smaller part of selected projection of the HNSW graph and converge quicker. Second, we measure the total distance calculations a heuristic makes (`t-dc`). For `blind`, `t-dc` is always equal to `s-dc`. However, for `directed`, `t-dc` can be larger than `s-dc` as `directed` computes distances to unselected vectors in the 1st degree neighbors of candidates. We use the difference of `t-dc` and `s-dc` as a measure of *`directed`'s overhead*. Figures 4.3 show the `s-dc` numbers for `blind` and `directed` and `t-dc` numbers for `directed`.

As selectivity levels decrease, `directed`'s overhead only increase because a larger fraction of the 1st degree neighbors of each candidate are unselected, so computing the distances to them increase overheads. Further, irrespective of the selectivity level, `directed` performs the search as effectively or better than `blind`. However, `directed`'s effectiveness edge over `blind` is low at very high and very low selectivity levels because at these levels they perform very similar explorations. As we already observed above, at very high selectivity levels these heuristics perform very similar searches. At very low selectivity levels, since very few vectors are selected, both heuristics again behave similarly because both explore every 2nd degree neighbor. That is also why at very low selectivity levels, `blind` outperforms `directed`in latency, because `directed`'s overhead are highest and it does not have an important advantage in terms of its search effectiveness. `directed` however has an edge over `blind` at medium selectivity ranges of 50% to 5% where its search edge is larger and overheads are low enough. That is why at roughly these ranges we see `directed` to outperform `blind` by upto 2x in latency. Overall, since `onehop-s`'s recall degenerates below 50% selectivity, `directed` is the most performant fixed heuristic. Below and at 5% `blind` outperforms both `directed` and `onehop-s`. This matches the regions we outlined in Figure 3.6.

Next, observe that, except for a few small regions, `adaptive-g` is able to capture the best of all fixed heuristics in almost all ranges, indicating that using global selectivity information is enough to capture the best of all worlds in these benchmarks. The only exception is within ranges below but close to 50% on some datasets. This is because we choose 50% as a safe threshold in `adaptive-g` to switch to `onehop-s`, although in some datasets even between 30% and 50%, `onehop-s`'s recall is high enough.

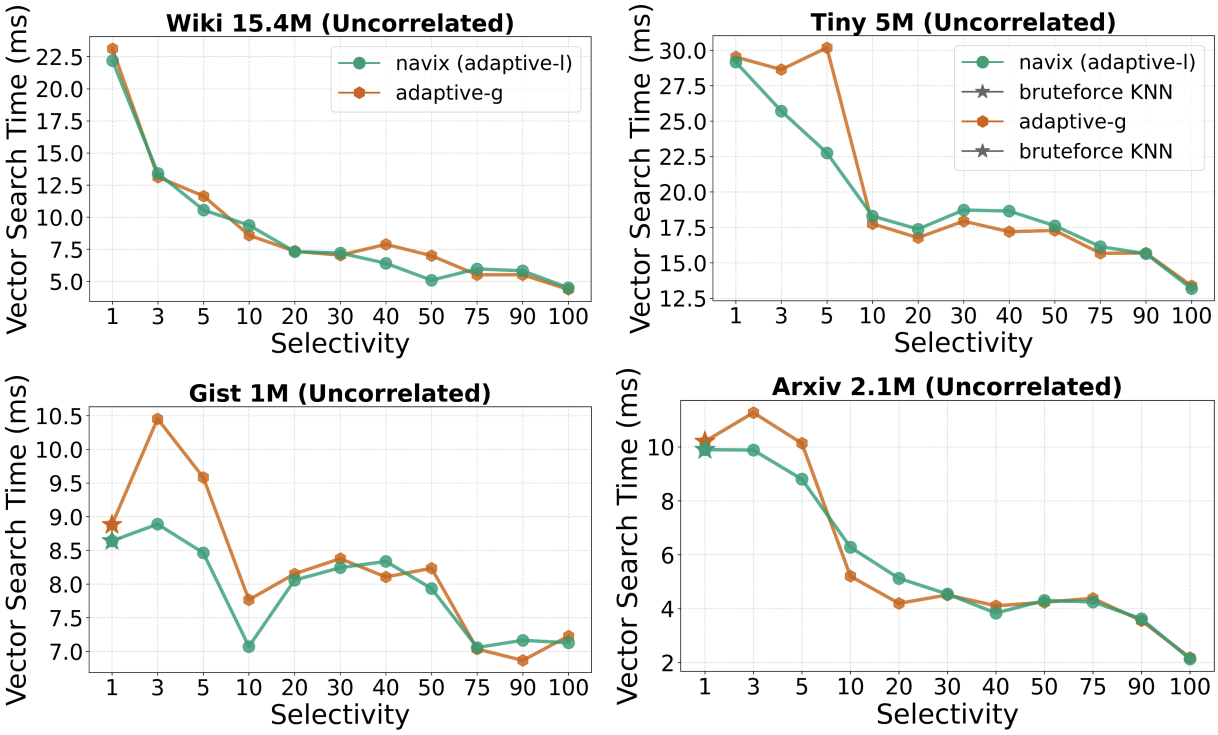


Figure 4.4: Vector search time vs selectivity for NaviX and `adaptive-g` for Wiki, Tiny, Arxiv, and GIST uncorrelated workloads.

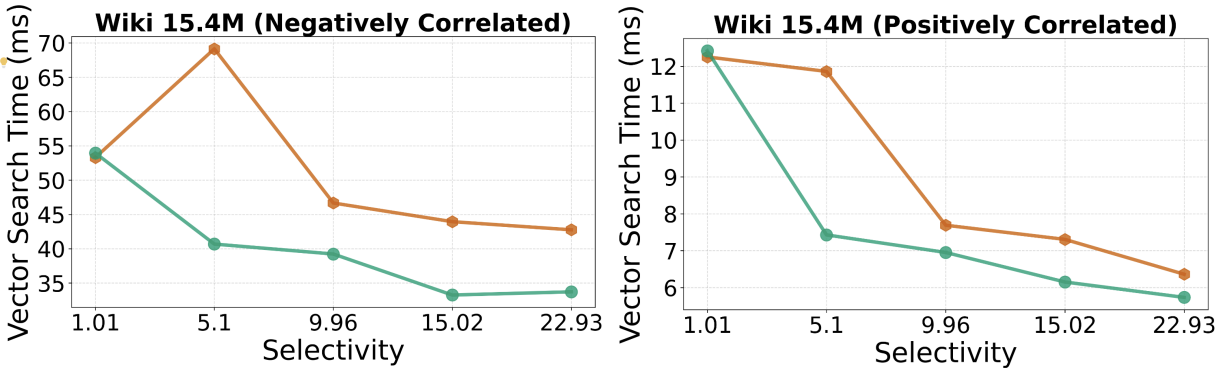


Figure 4.5: Vector search time vs selectivity for NaviX and `adaptive-g` for Wiki correlated workloads.

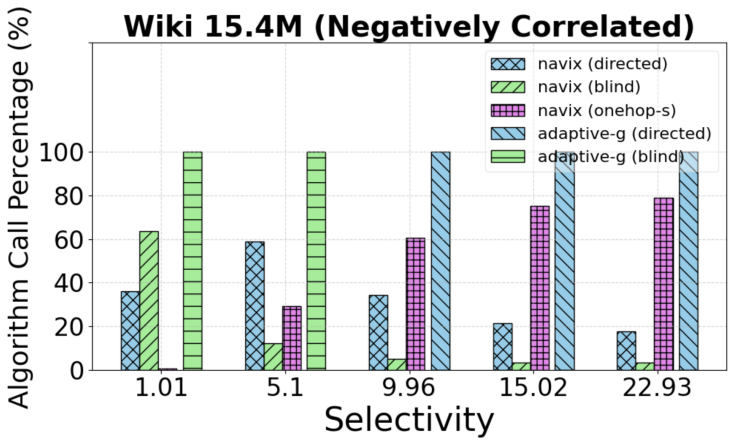


Figure 4.6: Wiki Negatively Correlated heuristics call distribution.

4.3 Adaptive-G and NaviX

In our next set of experiments we focus on comparing `adaptive-g` and NaviX (adaptive-l). Our goal is to show that NaviX is a strict improvement over `adaptive-g` and the advantage of NaviX is especially visible when queries are correlated with the vectors S that Q_S selects. We used the Wiki dataset and each of its workloads and measured the vector search times of `adaptive-g` and NaviX. We also used the uncorrelated workloads of our datasets. Figure 4.4 presents our uncorrelated workload results and Figure 4.5 presents our negatively and positively correlated workload results.

Observe that in all the uncorrelated workloads, `adaptive-g` and NaviX perform very similarly except at lower selectivities for Tiny and GIST, where NaviX slightly outperforms `adaptive-g` by up to 30%. This performance advantage occurs because NaviX can make better decisions when switching between different heuristics based on the local selectivity of the index traversal region.

In the negatively and positively correlated scenarios, the local selectivity of candidate nodes during search can be different than the global selectivity of S. Therefore we see `adaptive-g` and NaviX start making different decisions in terms of the fixed heuristics that they pick, which separates their behavior. Overall we observe that NaviX does better decisions and outperforms `adaptive-g` consistently, in some cases up to a factor of 1.7x. For example in the negatively correlated benchmark at 5% selectivity level, the average vector search latency of `adaptive-g` is 69.13 ms while that of NaviX is 40.67 ms.

To verify that NaviX and `adaptive-g` make different choices in terms of the heuristics they pick, we calculated the distribution of each heuristic they pick for different workloads. Figure 4.6 shows these distributions for the Wiki negatively correlated workload. The figure reports what fraction of times NaviX and `adaptive-g` picks each heuristic when exploring a candidate's neighborhoods for each selectivity level. Observe that since `adaptive-g`'s choice is based on the global selectivity, at each selectivity level, it commits to using one heuristic. Instead, NaviX makes more nuanced decisions. For example, in 22.9% selectivity, while `adaptive-g` commits to `directed`, NaviX picks `onehop-s` 80% of the time, indicating that it has performed the majority of the search in a region that contain vectors with very high local selectivity.

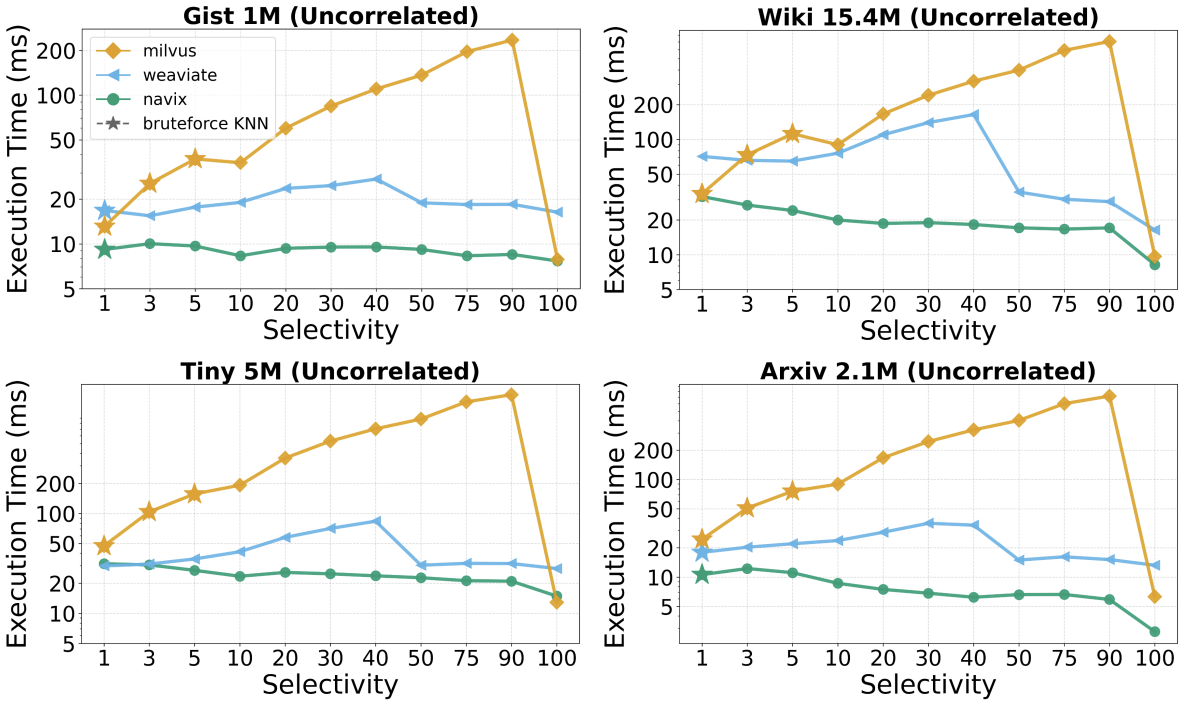


Figure 4.7: Execution Time vs Selectivity for prefiltering baselines with >95% recall and within 1% of each other

4.4 Comparison Against External Baselines

We next compare NaviX against our external baselines Milvus, Weaviate, PGVectorScale, and VBase. We first use all of our uncorrelated workloads on all systems. Then we used PGVectorScale and VBase on our correlated workloads. We omitted Weaviate and Milvus in correlated workloads because these queries contain joins, which these systems do no support. We measure end-to-end latency in these experiments. Figure 4.7 and 4.8 shows our results.

4.4.1 Comparisons Against Weaviate and Milvus

Weaviate adapts between two search heuristics. Above 40% threshold they use the `onehop-a` heuristic which includes all nodes whether selected or not as candidates for the traversal. Below and at 40%, they use the simpler version of `blind` from Acorn [38] that is configured to explore between 32 and 8×32 many vectors in the 2nd degree. This explains the latency hikes at 40%. Milvus is overall the slowest implementation in our experiments. Milvus does `onehop-a` heuristic at 100% and `bruteforce` below and at 5% selectivity, where we start observing latency drops. It performs postfiltering at the higher selectivities and `onehop-a` based prefiltering at the medium to lower selectivities [54]. Observe further that NaviX outperforms both of these systems. Specifically, when Weaviate switches to `blind` heuristic at 40% selectivity, the difference is quite drastic, 18.28 ms for NaviX compared to 164.19 ms for Weaviate on Wiki dataset. There are necessarily many differences across these different systems but we attribute the performance differences partly due to NaviX's better choice of search heuristics and partly to its fast zero-copy distance computations. Importantly, these experiments indicate that vector indices inside DBMSs can be competitive with specialized vector databases.

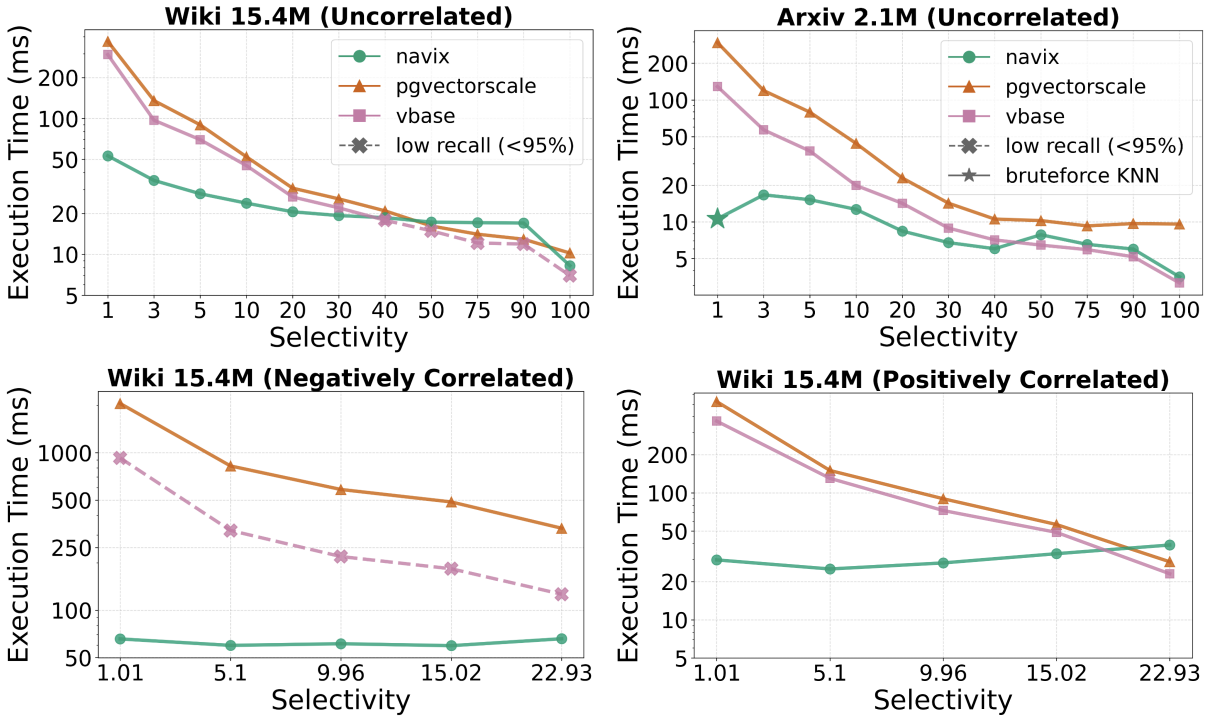


Figure 4.8: Execution Time vs Selectivity for postfiltering baselines with >95% recall and within 1% of each other

4.4.2 Comparisons Against External Postfiltering Baselines

We begin by studying the behavior of these systems in Wiki and Axriv uncorrelated workloads. Our results are shown in Figure 4.8. However, for Tiny and GIST, both postfiltering systems were unable to reach 95% target recall. PGVectorScale achieves 39.65% and 39.39%, whereas VBase achieves 74.09% and 81.26% for Tiny and GIST workloads respectively. Therefore, we omit these experiments. We can break the cost of postfiltering approaches into three main components:

- *Preprocessing cost*, if any, to prepare any intermediate tables that are needed to check if a streamed vector is in S or not.
- *Vector search cost* of streaming nearest neighbors from closest to furthest to k. This is determined by the likelihood of vectors in the nearest neighbors of v_Q being in S. The major factor that determines this is the selectivity level.
- *Verification cost* of checking if a streamed vector is in S or not.

In our uncorrelated workloads, observe that at high-selectivity levels, postfiltering approaches have an advantage over prefiltering appraoches. This is the ideal case for postfiltering: there is no preprocessing cost and the verification cost is cheap, a lookup to run a simple predicate on the object ID of the vector tuple. At these selectivity levels, prefiltering approaches, specifically NaviX pays the upfront cost of running the predicate on all vectors. Therefore postfiltering systems outperform NaviX above 50% selectivity. However, NaviX slightly outperforms PGVectorScale for Axriv dataset even above 50% selectivities. This is because the Arxiv dataset is smaller, thus the majority of the cost is due to vector search time rather than prefiltering time.

As the selectivity decreases, vector search cost increases, as more tuples need to be streamed, so cumulative verification cost also increases. Therefore the performance of postfiltering systems degrades significantly. For example, PGVectorScale's latency numbers degrade from about 10.22 ms to 334.42 ms for the Wiki uncorrelated workload. A similar degradation also appears in prefiltering approach, e.g., NaviX but to a lesser extent because vector search is aware of filtering which helps it to only consider selected nodes to converge faster. Thus, NaviX latency only degrades from 8.26 ms to 53.14 ms.

We next used our Wiki correlated workloads. Our results are shown in Figure 4.8. Although our correlated queries have joins of Person and Chunks, PGVectorScale and VBase plan still do not perform any preprocessing. Their plans stream a Chunk tuple `v(cID)` from a vector search operator. Then an IndexNestedLoopJoin (INLJ) operator joins v with PersonChunk on `cID`. If a tuple `pc(pID,cID)` is found, a predicate on the `pID` is executed. The primary difference is that now there is a more expensive verification cost. However, the general patterns exist. As expected, queries are broadly easier on positively correlated scenarios than negatively correlated scenarios for all systems, i.e., latencies generally go down. For example PGVectorScale's runtimes in the 1% to 22.9% range are 331.38 ms to 2052.44 ms and 28.67 ms to 524.54 ms respectively, in the negatively and positively correlated cases. NaviX is more robust to selectivity changes and its latency numbers are between 71.86 ms to 65.7 ms and 38.86 ms to 29.66 ms.

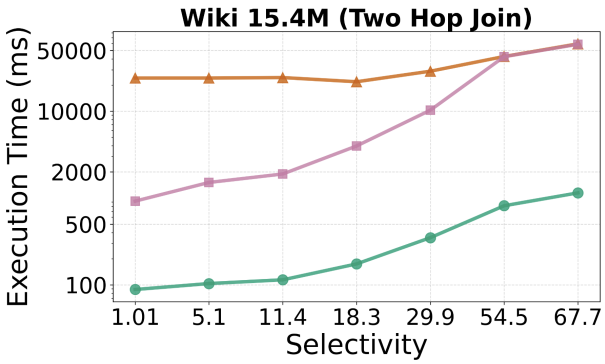


Figure 4.9: Execution Time vs Selectivity for two-hop join postfiltering baselines with >95% recall and within 1% of each other

We also extended our experiments with an additional 2-hop join benchmark on Wiki that represents a graph RAG application [62, 20] use case. In this workload selection subqueries are of the following structure:

```
MATCH (p:persons)-[pr:PersonResource]->(r:Resource)
      (r)-[rc:ResourceChun k]->(c:Chunk)
WHERE p.birth_date >= {s_date} AND p.birth_date < {e_date}
PROJECT GRAPH S(c)
CALL QUERY_HNSW_INDEX(S, HNSWIndex, v_Q, k)
RETURN c.cID
```

To write an equivalent SQL version, we create a temporary table using SQL's WITH statement. Unlike our previous experiments, PGVectorScale and VBase generate plans that have preprocessing costs. Specifically, they first materialize the 2-hop join in a temporary table `Tmp`. Then, for each streamed vector v, they check if v joins with any tuple in `Tmp` in a NestedLoopJoin operator. For vector queries, we used our previous queries in the correlated workloads. Our results are shown in Figure 4.9. First, these queries are harder than prior queries because the selection subquery is harder. Therefore each system's performance degrades. Also observe that every system's curve is now an upward curve. This is because as selectivity levels increase, even though vector search gets faster, the preprocessing cost for both post and prefiltering systems increase. This is because the size of the 2-hop join that needs to be computed increases, which offsets the costs.

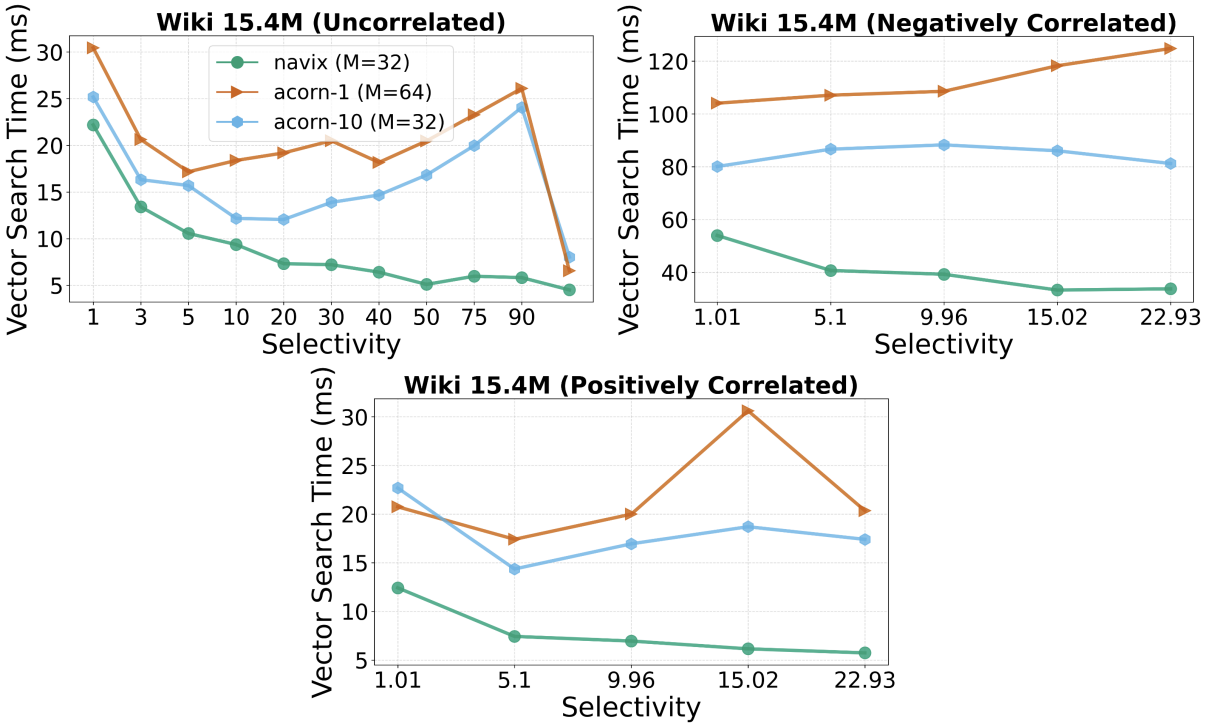


Figure 4.10: Vector search time (ms) vs selectivity for NaviX and Acorn-δ.

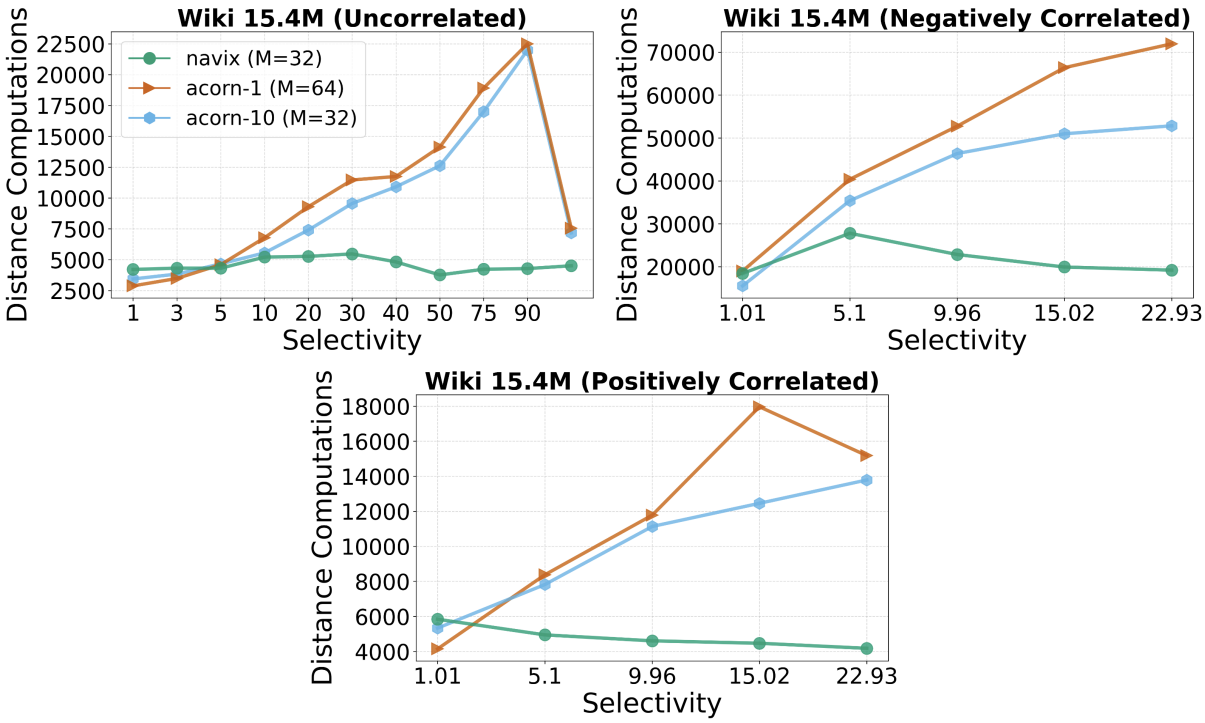


Figure 4.11: Distance Computations vs selectivity for NaviX and Acorn-δ.

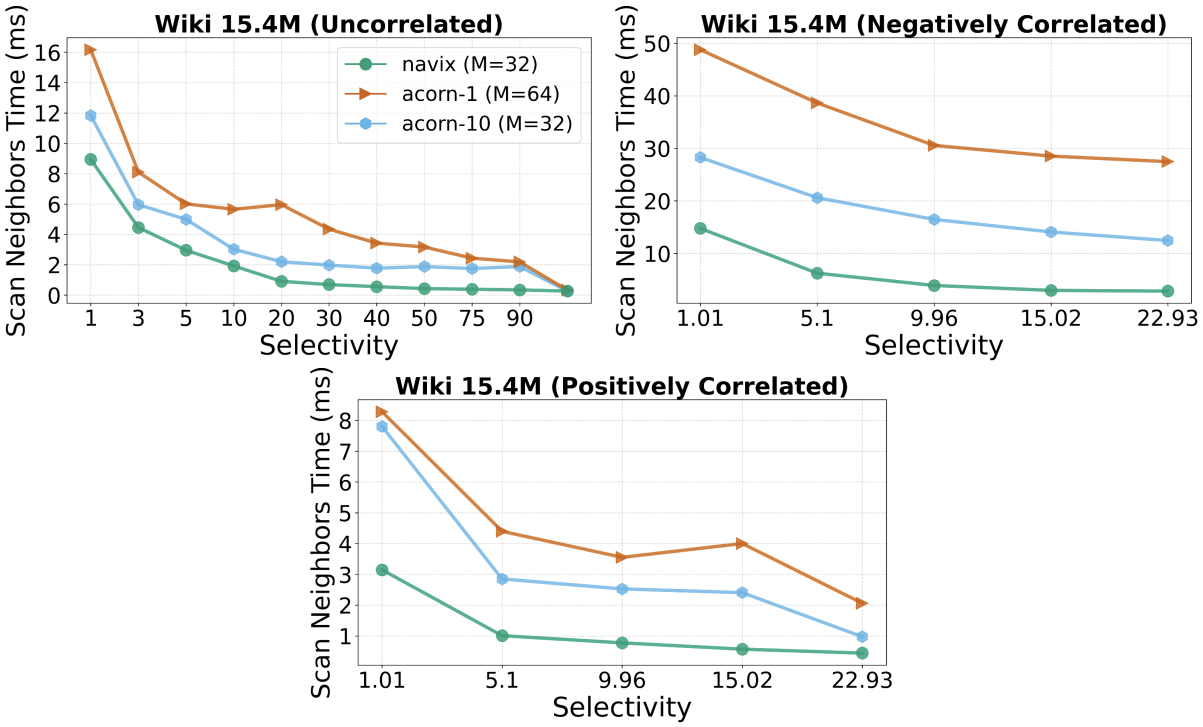


Figure 4.12: Time (ms) spent on scanning neighbors vs selectivity for NaviX and Acorn-δ.

4.5 Evaluation of Navix and Acorn:

Systems	Indexing Time (mins)	Avg Index Degree
NaviX(M=32)	55.22	31.9
Acorn-1 (M=32)	17.15	38.2 (∼1.2x)
Acorn-1 (M=64)	44.89	74.50 (∼2.3x)
Acorn-10 (M=32)	222.06	135.74 (∼4.2x)

Table 4.5: Indexing Time (mins) and Avg Index Graph Degree for all configurations of Acorn-δ and NaviX used in this evaluation.

This evaluation compares NaviX (`adaptive-local`) implementation with Acorn-δ on Wiki uncorrelated and correlated workloads. We have implemented Acorn's index in Kùzu using their version of the `blind` heuristic to replicate the design in reference [38]. This

ensures that we compare search performance and index size of these algorithms and indices within the same system implementation. We only compare the vector search time since prefiltering time is similar across both configurations of Kùzu. Figure 4.10 demonstrates our results and Table 4.5 demonstrates the corresponding indexing time and average index graph degree.

Recall from Section 5 that Acorn-δ uses a different index than NaviX and the other baselines we used in this thesis. Specifically, Acorn-δ changes the original index by removing (for $\delta = 1$) or modifying (for $\delta > 1$) the pruning routine during index construction, which makes the graph denser for any fixed M value. We compared three configurations of Acorn-δ: δ=1 with M=32, δ=1 with M=64, and δ=10 with M=32 against NaviX (M=32, efC=200).

We found that removing the pruning routine in Acorn-1, despite making the HNSW graph denser (by 1.2x), actually degrades recall for the same value of M (i.e., 32) if pruning is used. This indicates the pruning [49] routine is essential for creating diverse edges within the HNSW graph. Despite using a denser graph, Acorn failed to reach our target recall, which is why we excluded this configuration from the vector search time evaluation in Figure 4.10. In this configuration, Acorn-1 (M=32) reaches 87.53%, 83.32%, and 91.75% recall for k=100 average across all selectivities on uncorrelated, negatively, and positively correlated workloads respectively. Therefore, we increased Acorn's density by increasing M to 64 for $\delta = 1$ (2.3x denser) and increasing δ to 10 with M same as 32 (4.24x denser) to match our target recall.

As shown in Figure 4.10. Acorn-10 outperforms Acorn-1, which aligns with the observations from the Acorn paper [38]. NaviX consistently outperforms both Acorn configurations across all benchmarks. In the uncorrelated workload at 100% selectivity, all systems perform similarly. However, starting from 90% selectivity, Acorn's simpler version of the `blind` heuristic activates, exploring second-degree neighbors one at a time without considering which ones are closer to the query. The NaviX `adaptive-local` search algorithm is a strict imporovement over this approach, as mentioned in Section 4.2 and 4.3. At low vector selectivities, we found that both Acorn configurations perform fewer distance computations as shown in Figure 4.11, yet despite this they have higher latencies. To understand this better we profiled the executions and found that Acorn implementations now spend significant time performing traversals on the actual graph, because their graphs are denser and contain much more edges. By traversal, we are referring to scanning neighbors of nodes and not on the distance computations, for which we used the term "exploration" throughout this thesis. Figure 4.12 demonstrates this by showing the amount of time spent on get neighbor calls to scan neighbors of nodes in NaviX and Acorn configurations. For example, Acorn-1 spent 16.16 ms at 1% selectivity, while NaviX spent only 8.94 ms. Similar behaviour is

also visible in negative and positive correlated scenarios.

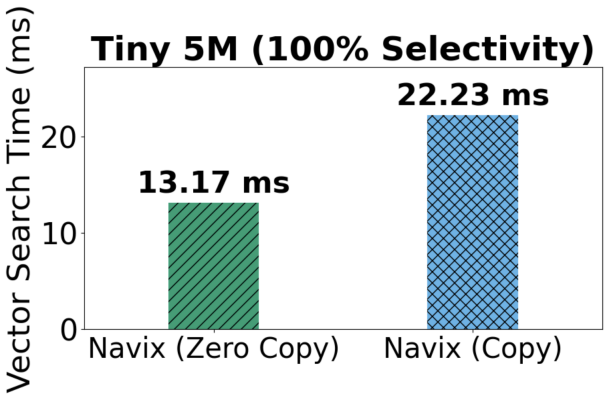


Figure 4.13: Vector Search Time with and without zero-copy optimization for 95% recall

4.6 Evaluation of In-BM Distance Calculations:

In this evaluation, we do an ablation study on how much performance benefits our in-bm distance calculation optimization provides. As shown in Figure 4.13, we used the Tiny dataset at 100% selectivity. We turned the in-bm distance calculation optimization off in NaviXẆe refer to this configuration as NaviX-copy. Figure 4.13 shows our results. We see that the in-bm optimization improves the performance of NaviX by 1.6x, demonstrating its importance for DBMSs that use a BM to scan their vectors into their vector search operators.

Chapter 5

Related Work

Our work is related to prior work from several areas:

5.1 Approximate Nearest Neighbours Indices

Existing indices that are used for approximate kNNS queries can be broadly categorized into two: (i) clustering-based indices [13, 46, 6, 18, 11]; and (ii) graph-based indices [10, 45, 35, 26]. NaviX adopts HNSW [26], which is a graph-based index. Overall, graph-based indices are empirically shown to be superior in terms of recall and performance compared to clustering-based approaches [4]. We now briefly go through these two categories and their limitations in detail.

5.1.1 Clustering Based Indices

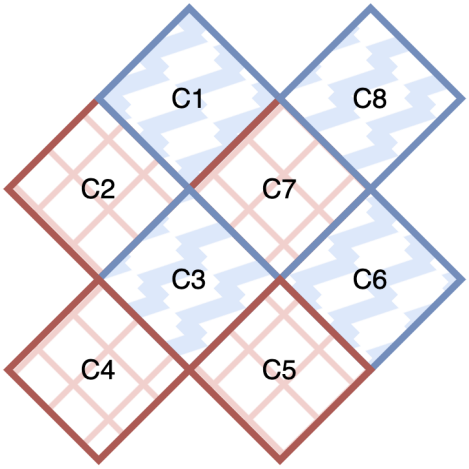


Figure 5.1: Pictorial representation of clustering-based indices where Ci represents a cluster in n-dimensional space

As shown in Figure 5.1, clustering-based indices partition the n-dimensional space into distinct regions. Given a query vector v_Q, the system searches one or more of these partitions to find its k-nearest neighbors (kNNs). IVF [18, 28] is one of the most commonly used clustering-based indices. During index building, it uses K-means to identify a set of centroid vectors and their corresponding clusters that partition the space. During the search, it performs a linear search over the centroids to find the $nProbes$ nearest centroids, then conducts a linear search within those centroid clusters to find the kNNs. The $nProbes$ configuration is needed because the nearest neighbor and v_q may lie on the boundary of the closest cluster and thus can be part of another cluster. The advantage of IVF compared to graph-based methods is its fast index-building time since K-means is the only computationally intensive step. However, IVF performs significantly slower in terms of both search latency and recall [4]. This is because in IVF each cluster is relatively small (around 10k-30k vectors) thus, achieving high recall requires searching through many clusters which impacts performance.

There are many variants of the IVF index. IVF-HNSW [28] is an index where the centroids are further indexed using HNSW, which improves the efficiency of finding the closest centroids. SOAR [46] is another IVF-style index that improves search performance by duplicating vectors at cluster boundaries to nearby clusters. This approach allows for a lower $nProbes$ value while maintaining the same recall. However, it requires more

storage space and makes update operations more complex. Another challenge with the IVF-based techniques is skewness due to the change in data distribution, some clusters become significantly larger than others, impacting search performance. This necessitates regular reindexing to handle evolving data distributions.

Locality-sensitive hashing [11] is another clustering-based index that uses a hash function designed to generate similar hashes for vectors that are close to each other. This hash function effectively helps to create a set of clusters. However, due to its poor search performance and recall [4], it is not widely adopted.

5.1.2 Graph-Based Indices

As explained in Section 2.2, Graph-based indices utilize a directed graph structure where vertices represent vectors in the dataset and edges indicate relationships between vectors. In this structure, denoted as $G = (V, E)$, each vertex $v_i \in V$ corresponds to a dataset vector, while directed edges $e_j \in E$ represent vector proximity. The relationship between any two vertices (u, v) is characterized by their distance measure $dist(u, v)$, which determines whether the connecting edge is classified as short or long [15]. This organizational structure is referred to as a proximity graph.

The search process in all proximity graph based indices follows a similar greedy approach i.e. search starts from a random entry point, and greedily moves toward the query vector by exploring the neighborhood.

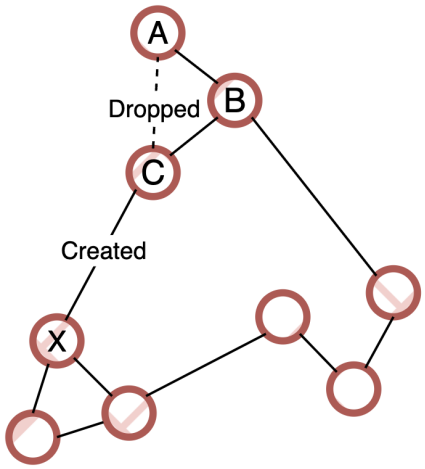


Figure 5.2: Proximity graph where edge (C,A) is dropped and (C,X) is created due to the RNG property

For optimal navigation i.e. to prevent the search from getting stuck in local minima, a graph-based index needs a proper balance of long- and short-range edges. Long-range edges allow efficient traversal from the initial point to the target region, while short-range edges enable precise convergence and accuracy within that region. Edge pruning helps achieve this balance. The Relative Neighborhood Graph (RNG) strategy [49] offers a systematic method for edge selection in proximity graphs. This strategy is used in many graph-based indices including HNSW, Vamana [45], and RNN-Descent [35]. In RNG, an edge between vertices $(u, v) \in V$ with distance function $dist(u, v)$ remains only if no other vertex $u' \in V$ can prune it. Specifically, u' triggers edge pruning when it is closer to both vertices than they are to each other, i.e., when $dist(u, u') < dist(u, v)$ and $dist(v, u') < dist(u, v)$. In essence, this strategy eliminates the longest edge within any triangular configuration in the graph. Figure 5.2 represents a case where the edge between A and C is dropped because it's the longest edge in the ABC triangle. This leads to the creation of a new long-range edge between C and X since C hasn't reached its maximum neighbors limit and X is the next closest neighbor.

Graph-based indices further fall into two categories based on their construction algorithm: incremental ingestion-based construction, such as HNSW [26] and refinement-based construction, such as Vamana from DiskANN [45], RNN-Descent [35], and NSG [10]. In the refinement-based construction technique, the index starts with a random initial graph connecting vectors, then creates a proximity graph through multiple refinement iterations which gets applied to the entire index. While, these indices sometimes build faster than incremental-based methods, their search performance is comparable or inferior to HNSW. We used HNSW in this thesis as it is the most widely adopted graph-based technique [29, 57, 19, 64]. However, our design for predicate-agnostic search is orthogonal to how the graph-based indices are constructed and can easily be extended to any other proximity graph-based index.

5.2 Predicate-agnostic Vector Search

Several prior DBMSs, vector databases, or vector search systems are capable of evaluating predicate-agnostic vector search queries. These systems adopt pre- or postfiltering or a mix of these approaches. They also adopt graph or clustering-based indices, or a mix of these. We covered Weaviate, Milvus, PGVectorScale, and VBase in prior sections. We cover Acorn and AnalyticDB-V here.

Closest to our work is Acorn-δ [38], which supports predicate-agnostic vector search on an HNSW-like index. As we showed in this thesis, Acorn's `blind` heuristic is a good

choice in very low selectivity levels but is suboptimal in medium and high selectivity levels compared, respectively, to `directed` and `onehop-s` heuristics. Acorn-δ also uses a different index than NaviX and the other baselines we used in this thesis. Specifically, Acorn-δ changes the original index by removing (for $\delta = 1$) or modifying (for $\delta > 1$) the pruning routine during index construction, which makes the graph denser for any fixed M value. As shown in Section 4.5 where we compared NaviX against Acorn with $\delta = 1$ and $\delta = 10$ implemented in Kùzu. We found that removing the pruning routine in Acorn-1, despite making the HNSW graph denser (by 1.2x) in fact degrades recall for the same value of M if pruning is used. This indicates the pruning routine is essential for creating diverse edges within the HNSW graph. Therefore despite using a denser graph, we found Acorn to not reach our target recall. We also increased Acorn's density by increasing M for $\delta = 1$ (2x denser) and by increasing δ to 10 (4.24x denser) to match our recall target. In these configurations, we found Acorn to perform fewer distance computations than NaviX but only in very low-selectivity scenarios. However, we found NaviX to still be more performant in vector search time across all selectivity levels because it uses a smaller index and traverses fewer number of edges.

AnalyticDB-V [58] is a distributed analytical RDBMS that implements a mix of HNSW index and a compressed clustering-based index called *Voronoi Graph Product Quantization* (VGPQ) index. The primary index is VGPQ. The newly inserted vectors are first inserted into an HNSW index and periodically merged into the VGPQ index. Since the primary index VGPQ, the HNSW search is not the bottleneck in predicate-agnostic search queries. On the VGPQ index, AnalyticDB-V performs a mix of post- and prefiltering approach based on the estimated selectivity of the selection subquery.

5.3 Specialized indices for predicates on low cardinality attributes

Several prior works have developed graph-based indices that can evaluate *predicate-aware* vector search queries. These approaches assume a set of predicates, or the structures of these predicates, are known apriori and construct indices whose edges contain extra edges that can be used to evaluate these predicates. Filtered-DiskANN [12] proposes methods that can support equality predicates on attributes that contain up to 1000 values. NHQ [5] and HQANN [60] are approaches that encode the attribute to filter on directly into the vector as a new dimension and use this during index construction to create extra edges between similar attribute vectors. This method works on small cardinality attributes and on simple filtering operations. HQI[30] partitions the index based on attributes whose

cardinality can be at most 20. This allows searching within a one or a set of partitions to evaluate equality or range queries on this attribute domain.

Finally, SeRF[65] and iRangeGraph[61] support arbitrary range filtering queries on an attribute in the context of graph-based vector indices. At a high level, both approaches construct segmented sub-indices for different ranges and merge them together to support arbitrary ranges, with the goal of maximizing overlaps across each sub-index to minimize the merged index. However, since these approaches require constructing many sub-indices, they are significantly slower during index construction.

Overall, these specialized indices are generally highly limited in terms of the predicates or the structures of their selection subqueries. For example, they are not designed to support selection subqueries that includes join or filters on string-based predicates.

5.4 Native Vector Indices in DBMS

We covered PGVectorScale and VBase in prior sections. Several other DBMSs or DBMS extensions also support vector indices. PGVector [19] and PASE [63] are vector index extension of PostgresSQL. Finally DuckDB and Neo4j also support vector indices. DuckDB uses the USearch [52] library and Neo4j uses Apache Lucene [7]. Unlike our approach, which integrates a native vector index, these systems use separate indices and do not support predicate agnostic filtered search.

Chapter 6

Conclusion

We presented the design and implementation of NaviX, and demonstrated the robustness and efficiency of the `adaptive-local` algorithm we adopted to perform predicate-agnostic vector search. NaviX utilizes the selectivity information in the selected subset S to decide through simple rules which exploration heuristic to pick during the search. To capture possible correlations of S with a query vector v_Q, NaviX uses the local selectivity of the neighbors of each candidate vector it explores during the search. Our results show that NaviX is efficient and robust and is competitive with or outperforms both existing pre-filtering and postfiltering based systems under a variety of query workloads that contain different selectivities, correlations, as well as preprocessing costs.

References

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends® in Databases*, 5(3), 2012.

[2] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array. In *CCGrid*, 2022.

[3] arXiv. Arxiv Dataset. https://info.arxiv.org/help/bulk_data/index.html.

[4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.*, 87(C), 2020.

[5] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. *SIGMOD*, 2(6), 2024.

[6] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: highly-efficient billion-scale approximate nearest neighbor search. In *NeurIPS*, 2021.

[7] Doug Cutting. Apache Lucene. https://lucene.apache.org/.

[8] DBpedia. DBpedia Latest Dataset. https://databus.dbpedia.org/dbpedia/collections/latest-core, jul 2024.

[9] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system. In *CIDR*, 2023.

[10] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *PVLDB*, 12(5), 2019.

[11] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *VLDB*, 1999.

[12] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *ACM Web Conference*, 2023.

[13] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *ICML*, 2020.

[14] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. Columnar Storage and List-based Processing for Graph Database Management Systems. *PVLDB*, 14(11), 2021.

[15] Nico Hezel, Kai Uwe Barthel, Konstantin Schall, and Klaus Jung. An exploration graph with continuous refinement for efficient multimedia retrieval. In *ICMR*, 2024.

[16] HuggingFace. MTEB Embedding Leaderboard. https://huggingface.co/spaces/mteb/leaderboard.

[17] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Hamming Embedding and Weak Geometric Consistency for Large Scale Image Search. In *ECCV*, 2008.

[18] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE PAMI*, 33(1), 2011.

[19] Andrew Kane. PGVector. https://github.com/pgvector/pgvector.

[20] LangChain. Enhancing RAG-based application accuracy by constructing and leveraging knowledge graphs. https://blog.langchain.dev/enhancing-rag-based-applications-accuracy-by-constructing-and-leveraging-knowledge-graphs/, mar 2024.

[21] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[22] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[23] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *NeurIPS*, 2020.

[24] Jinfeng Li, Xiao Yan, Jian Zhang, An Xu, James Cheng, Jie Liu, Kelvin K. W. Ng, and Ti-chung Cheng. A General and Efficient Querying Method for Learning to Hash. In *SIGMOD*, 2018.

[25] Peng-Cheng Lin and Wan-Lei Zhao. Graph based Nearest Neighbor Search: Promises and Failures, 2019.

[26] Yu A. Malkov and D. A. Yashunin. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE PAMI*, 42(4), 2020.

[27] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces. In *SISAP*, 2012.

[28] Meta. Faiss. https://github.com/facebookresearch/faiss, 2023.

[29] Milvus. Milvus v2.5.0. https://milvus.io/, dec 2024.

[30] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F. Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. High-Throughput Vector Similarity Search in Knowledge Graphs. *SIGMOD*, 1(2), 2023.

[31] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F. Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. High-Throughput Vector Similarity Search in Knowledge Graphs. *SIGMOD*, 1(2), 2023.

[32] Blaise Munyampirwa, Vihan Lakshman, and Benjamin Coleman. Down with the Hierarchy: The'H'in HNSW Stands for "Hubs". *arXiv preprint arXiv:2412.01940*, 2024.

[33] Gonzalo Navarro. Searching in metric spaces by spatial approximation. *VLDB*, 11(1), 2002.

[34] Aude Oliva and Antonio Torralba. Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope. *Int. J. Comput. Vision*, 42(3), 2001.

[35] Naoki Ono and Yusuke Matsui. Relative NN-Descent: A Fast Index Construction for Graph-Based Approximate Nearest Neighbor Search. In *ACM MM*, 2023.

[36] OpenAI. OpenAI O1 Large Language Model. https://openai.com/o1/.

[37] Apache Parquet. Parquet Documentation. https://parquet.apache.org/docs/.

[38] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. In *SIGMOD*, 2024.

[39] Mykhailo Poliakov and Nadiya Shvai. Multi-Meta-RAG: Improving RAG for Multi-Hop Queries using Database Filtering with LLM-Extracted Metadata, 2024.

[40] Qdrant. ANN Filtered Dataset. https://github.com/qdrant/ann-filtering-benchmark-datasets.

[41] Bhaskarjit Sarmah, Dhagash Mehta, Benika Hall, Rohan Rao, Sunil Patel, and Stefano Pasquali. HybridRAG: Integrating Knowledge Graphs and Vector Retrieval Augmented Generation for Efficient Information Extraction. In *ICAIF*, New York, NY, USA, 2024. Association for Computing Machinery.

[42] Nova Search. stella_en_400M_v5 Embedding Model. https://huggingface.co/NovaSearch/stella_en_400M_v5, 2024.

[43] Gaurav Sehgal and Semih Salihoğlu. Kuzu Navix Repo. https://github.com/gaurav8297/kuzu.

[44] Gaurav Sehgal and Semih Salihoğlu. Navix Longer Version Paper. https://cs.uwaterloo.ca/~ssalihog/papers/navix-tr.pdf.

[45] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. DiskANN: fast accurate billion-point nearest neighbor search on a single node. In *NeurIPS*, 2019.

[46] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. SOAR: improved indexing for approximate nearest neighbor search. In *NeurIPS*, 2023.

[47] Timescale. PGVectorScale v0.5.1. https://github.com/timescale/pgvectorscale, dec 2024.

[48] Antonio Torralba, Rob Fergus, and William T. Freeman. 80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition. *IEEE PAMI*, 30(11), 2008.

[49] Godfried T. Toussaint. The Relative Neighbourhood Graph of a Finite Planar Set. *Pattern Recognition*, 12(4), 1980.

[50] Sentence Transformers. all-MiniLM-L6-v2 Embedding Model. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2.

[51] Ash Vardanian. SimSIMD. https://github.com/ashvardanian/SimSIMD.

[52] Ash Vardanian. USearch. https://github.com/unum-cloud/usearch.

[53] Andrei Vasnetsov. Filtrable HNSW - Qdrant. https://qdrant.tech/articles/filtrable-hnsw.

[54] 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. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD*, 2021.

[55] Weaviate. Filtered Vector Search | Weaviate - vector database. https://weaviate.io/developers/weaviate/concepts/filtering.

[56] Weaviate. How we speed up filtered vector search with ACORN. https://weaviate.io/blog/speed-up-filtered-vector-search, nov 2024.

[57] Weaviate. Weaviate v1.28.2. https://weaviate.io/, dec 2024.

[58] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. AnalyticDB-V: A hybrid analytical engine towards query fusion for structured and unstructured data. *PVLDB*, 13(12), 2020.

[59] WikiMedia. WikiMedia enwiki Dump. https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2, jul 2024.

[60] Wei Wu, Junlin He, Yu Qiao, Guoheng Fu, Li Liu, and Jin Yu. HQANN: Efficient and Robust Similarity Search for Hybrid Queries with Structured and Unstructured Constraints. In *CIKM*, 2022.

[61] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. irangegraph: Improvising range-dedicated graphs for range-filtering nearest neighbor search. *SIGMOD*, 2(6), 2024.

[62] Zhentao Xu, Mark Jerome Cruz, Matthew Guevara, Tie Wang, Manasi Deshpande, Xiaofeng Wang, and Zheng Li. Retrieval-Augmented Generation with Knowledge Graphs for Customer Service Question Answering. In *ACM SIGIR*, 2024.

[63] Wen Yang, Tao Li, Gai Fang, and Hong Wei. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *SIGMOD*, 2020.

[64] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *OSDI*, 2023.

[65] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. Serf: Segment graph for range-filtering approximate nearest neighbor search. *SIGMOD*, 2(1), 2024.