Workload-Guided Partitioned Filtered Approximate Vector Searches

Evan Gebo[*]
University of Michigan
Ann Arbor, Michigan, USA
evangebo@umich.edu

Nolan Kuza[*]
University of Michigan
Ann Arbor, Michigan, USA
nkuza@umich.edu

Tomás García Lavanchy[*]
University of Michigan
Ann Arbor, Michigan, USA
tomasgl@umich.edu

Abstract

We present a greedy tree-based partitioning system using atomic predicates for a workload-aware vector database. We also designed a novel synthetic dataset to test filtered approximate vector searches which is more complex than the synthetic workloads used in existing work. We ran experiments on this workload to test our algorithm, and concluded that while it has the potential to outperform a greedy range-based partitioner in terms of recall and query times, more work is needed to fine tune our algorithm. Our implementation of the algorithm and synthetic dataset is available at https://github.com/Nolan1324/CSE-584-W25-ANN-Project.

1 Introduction

Many applications involve storing documents with both vector embeddings and structured metadata, for example documents with text embeddings and also publication region, or images with visual embeddings and a timestamp. Fast filtered document retrieval is important for tasks where quickly searching for similar documents that meet certain criteria is needed, for example when using retrieval augmented generation with large language models or searching over image databases with filters. However, most vector database systems are optimized primarily for vector-only searches, rather than for searching on both vector data and structured metadata. While some vector databases support filtering vectors based on metadata, the common approaches typically involve performing vector search and metadata filtering as two separate steps rather than integrating them together. In this paper, inspired by classical partitioning techniques in relational databases, we develop a greedy partitioning algorithm for structured vector searches. We then analyze how this algorithm compares to the standard greedy range-based partitioning algorithm.

2 Related work

[16] outlines the basic methods for performing vector filtering, which are illustrated in fig. 1. In the “filter first" method, the filter is applied first and then an approximate nearest neighbors (ANN) search is run on the resulting vectors. This is best when the filter selectivity is low, as running ANN search on a few vectors is cheap even without an index. The system could instead use the filter results to generate a bitmap and then traverse the vector index while ignoring vectors that are not in the map. In the “filter later" method, vector search is performed first, and then the results are filtered. The system will have to oversample when performing vector search, since some vectors will be filtered out afterwards. Some systems do this iteratively by searching for increasingly many

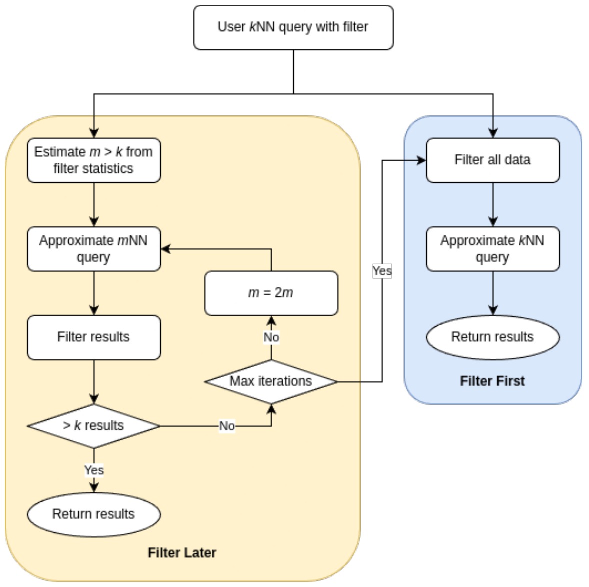


Figure 1: A flowchart illustrating the filter later approach (left) and the filter first approach (right) to filtered ANN queries.

batches of vectors from the index until there are enough that satisfy the filter.

One common extension to improve performance is to first partition the dataset, and then filter the partitions as a whole and only then search surviving partitions. This process is illustrated in fig. 2. There are several primary benefits of this approach. First, it improves distributed performance since each partition can be processed on a separate machine. Second, if a partition can be entirely excluded by the filter, then the system can skip vector search (the most expensive step of a filtered search) for that partition altogether. Finally, certain vector indices, for example HNSW, need to be stored in memory when used. This limits the size of the index by the working memory, so by partitioning and building an index per partition, each index can only be loaded when relevant, allowing us to work on larger datasets.

Consequently, most modern vector databases adopt a partitioning strategy, either explicitly exposed to users or hidden internally. For instance, Milvus hides the partitioning by default, organizing data into chunks called segments. Vectors are assigned to segments in an incremental manner, sealing the segment and then creating a new one once it grows too large. When a user requests an index build, Milvus constructs separate indices for each segment instead

[*]All authors contributed equally to this research.

of a single index. Milvus also supports a “cluster compaction" operation, which rearranges vectors in clusters based on ranges of a certain “cluster attribute" so that some clusters may be pruned during filtered searches. However, this process must be triggered *after* data insertion, is fairly opaque to the user, and only allows partitioning on one attribute purely based on data distribution information.

Milvus also supports higher level vector partitioning. The user can define their own partitions and decide which partitions to insert into and search from. It also supports an automatic mod-based partitioning scheme, which in the absence of any workload or distribution information is optimal for ensuring that the data is easily distributed across workers. However, it is not paticularly useful for pruning partitions on range-based attribute filters; for instance, a “greater than" attribute filter would still require searching all partitions under the modulo.

2.1 Relational partitioning

Partitioning is a well-studied problem in relational databases, with several established techniques. Our analysis focuses on horizontal partitioning, which groups rows rather than columns. This is appropriate for our application, where reducing the number of vectors searched can yield significant performance gains. Skipping entire partitions allows us to avoid unnecessary data loading, improving efficiency.

Partitioning strategies can be classified into two categories: dataset-aware but workload-oblivious, and strategies that are aware of both dataset and workload. Unfortunately, determining the optimal partitioning scheme is NP-hard in both cases. A common dataset-aware but workload-oblivious heuristic is range or mod partitioning on a single attribute, and this approach is implemented in most relational and vector databases by default.

[2] is one such paper that develops a workload-aware partitioning system for relational databases through a genetic algorithm. By compiling all predicates used throughout the workload, and breaking down predicates until they cannot be broken down further (forming atomic predicates), one can create a baseline set of partitions on all predicates. The genetic algorithm employs the use of a cost estimation model to select the best partitioning scheme, and then mutates said set of partitions, merging them, for the next batch of testing. Our work takes inspiration from their use of atomic predicates, though we do not employ a genetic algorithm.

Another workload-aware approach is taken in [4], which reduces the database partitioning problem to a weighted hypergraph max-cut instances which can be efficiently approximated. It creates this graph by representing data points as vertices and the queries that access the data points as hyper-edges. The approximate cut solution corresponds to a partitioning scheme. An interesting quality is that this scheme optimizes data locality as well as query time, meaning that data that is likely to be accessed together (e.g. a customer and their address stored in separate tables) are stored together. However, in our use case the attribute table is generally completely un-normalized [16], so we don’t need to worry about this form of data locality.

2.2 Filtered ANN

While most works treat filtering and the ANN search as two separate stages there are some methods to enhance these approaches. For instance, [13, 16, 18] recognize the unique challenge from hybrid queries, as different plans may result in varying performance, and thus employ a cost-based model to dynamically choose whether to employ filter-first or filter-later based on filter selectivity and other factors.

Some works also design custom ANN graph [10, 15] indices instead of supporting any vector index as a black-box, which is the approach taken in [13, 16, 17]. The filtered DiskANN [6] is one instance of an ANN graph designed explicitly for filtered similarity searches. In essence, it works by considering the attributes during the index construction process. [19] develops indices that can be efficiently merged, which are useful in filter-first approaches by allowing you to use higher-quality indices formed from merged indices that survive the partition-level filter instead of the low-level indices originally built for each partition. However, these approaches that modify the underlying graph structure are difficult to integrate into well-established vector databases. Thus, our work aims to investigate solutions that do not require modifying the graph structure.

2.3 Greedy ranged-based partitioning

A simple partitioning approach applicable to both relational and vector databases is greedy ranged-based partitioning. For instance, Milvus uses this when performing cluster compaction. In this algorithm, we view the data as sorted on a single attribute. We then partition the attribute into ranges so that each partition contains a nearly equal number of data points. At search time, we simply prune away partitions whose ranges fall outside the search query. We use this algorithm as a foundation for the algorithm we develop in our work, and we also use it as a baseline for comparison in the results section.

3 Methodology

We designed a greedy tree-based partitioning heuristic that takes into account both the data and workload. We implemented the algorithm for use with the Milvus vector database, although it could be used for other vector databases as well.

3.1 Definitions

3.1.1 Dataset. In the vector database partitioning problem, our dataset consists of a set of vectors. Each vector also has a set of named, scalar attributes. In our work, we only consider integer attributes. For example, we may have two attributes named x and y. Then a vector v may have attribute values $x = 5$ and $y = 7$. Our partitioning algorithm will only consider the attribute data.

3.1.2 Workload. A typical vector search query consists of a query vector v and an integer k; the database attempts to retrieve the top-k nearest neighbors to v. A **filtered** vector search query additionally includes a filter predicate ϕ that the retrieved vectors must satisfy. For instance, a predicate could be $\phi(x, y) = x \geq 100 \wedge y < 500$. We represent our workload as a collection of these filter predicates.

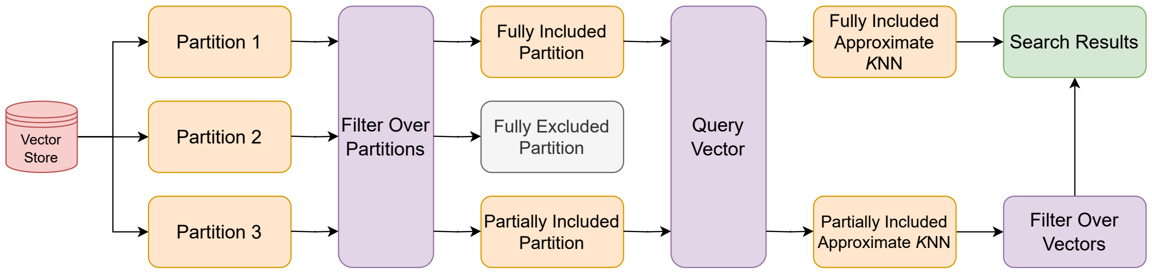


Figure 2: A flowchart illustrating the partitioning approach, where partitions are filtered individually and then passed to ANN algorithm for the vector search.

3.1.3 *Predicates.*

We define an *atomic predicate* as a unary predicate of the form $\alpha(x) = x \geq c$, where x is some attribute name and c is an integer constant. We can build compound predicates using $\wedge, \vee, \neg$ (and, or, not) logical operator; for instance, the filter $\phi = x \geq 100 \wedge \neg(y \geq 500)$ is built from two atomic predicates. Likewise, a compound filter can be decomposed its atomic predicates. Although this framework can technically represent equality predicates (=), we believe that equality predicates would require additional considerations in the algorithm and dedicate this to future work (section 6.3).

3.2 Design

At a high level, our partitioning algorithm extends on the basic range-based partitioning algorithm by taking both the dataset and workload into account. We first characterizes the workload by decomposing the filters into atomic predicates. Then, we perform a greedy tree-based heuristic based on these atomic predicates.

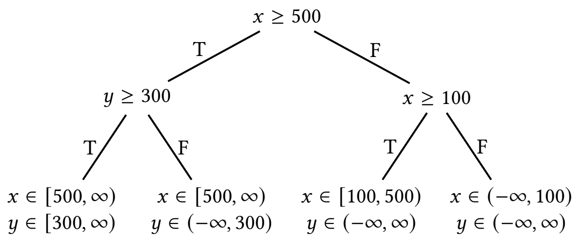
The resulting partitions are represented as a binary decision tree. Figure 3 provides a running example of such a tree. Each internal node of the tree consists of an atomic predicate $x \geq c$. When searching for which partition a vector lives in, we traverse to the left subtree if the predicate evalutes to T (true), and we traverse the right subtree if the predicate evalutes to F (false). The leaf nodes represent the partitions. For instance, in our running example, a vector with attributes $x = 700, y = 400$ would be found within the first leaf node.

We can also summarize each partition as a set of integer ranges in each attribute. In our running example, all vectors found in the first leaf node have attributes x, y that satisfy $x \in [500, \infty]$ **and** $y \in [300, \infty]$. We will use this range-based representation of the tree in later algorithms.

3.3 Workload characterization

The first step of the algorithm is to characterize the workload $\mathcal{W}$, summarized in algorithm 1. As previously defined, $\mathcal{W}$ consists of a set of predicts. We first change all atomic predicates in $\mathcal{W}$ to the canonical form $x \geq c$. This is because the two predicates $x \geq c$ and $x < c$ would result in identical subtrees in our partition tree; thus, we want to consider them as equivalent in later algorithms.

We then collect all the atomic predicates of the workload. We calculate how frequently each predicate occurs in the workload and



$x \geq 500$
T
F
$y \geq 300$
$x \geq 100$
T
F
T
F
$x \in [500, \infty)$ $y \in [300, \infty)$
$x \in [500, \infty)$ $y \in (-\infty, 300)$
$x \in [100, 500)$ $y \in (-\infty, \infty)$
$x \in (-\infty, 100)$ $y \in (-\infty, \infty)$

Figure 3: An example of a partition tree created from atomic predicates. Each leaf node represents a partition.

Algorithm 1 CHARACTERIZE-WORKLOAD($\mathcal{W}$)

```
Convert all predicates in $\mathcal{W}$ to the form $x \geq c$
Calculate atomic predicates $\{\phi_i\}_{i=1}^m$ from workload $\mathcal{W}$
Compute $f : \{\phi_i\}_{i=1}^m \to \mathbb{N}$, the frequency of each predicate
Sort atomic predicates $\phi_{\max} = \phi_1, \ldots, \phi_m = \phi_{\min}$ by $f$
return $\{\phi_i\}_{i=1}^m, f$
```

then sort them by frequency. This so that the later partitioning considers building partitions on the most frequent atomic predicates, which are likely most representative of the whole workload.

3.4 Partition tree generation

Once the sorted list of atomic predicates $\{\phi_i\}_{i=1}^m$ and their frequencies f are generated, we can build the partition tree, as summarized in algorithm 2. We first call BUILD-TREE with the entire dataset $\mathcal{D}$ and the full list of atomic predicates $\{\phi_i\}_{i=1}^m$. We then consider each atomic predicate until we find a suitable predicate to create a split on.

For each considered predicate ϕ_i, we check that its frequency $f(\phi_i)$ in the workload is above some lower bound f_{lower}. This is to avoid creating many partitions off of infrequently occuring predicates, as this could result in overfitting.

We then restrict the current dataset to $\mathcal{D}|_{\phi_i = \mathrm{T}}$, the entires that evaluate to T on the predicate. $s := \left|\mathcal{D}|_{\phi_i=\mathrm{T}}\right| / |\mathcal{D}|$ then measures how balanced this predicate would make the two splits. If $s_{\text{lower}} \leq s \leq 1 - s_{\text{lower}}$ (where s_{lower} is a hyperparameter), we continue. We want to avoid making the split unbalanced because having one very large partition and one very small partition could cause degenerate

Algorithm 2 BUILD-TREE($\mathcal{D}$, $\{\phi_i\}_{i=1}^m$, f)

```
for $i \in [m]$ do
    $s \leftarrow \left|\mathcal{D}|_{\phi_i=\mathrm{T}}\right| / |\mathcal{D}|$
    if $f(\phi_i) \geq f_{\mathrm{lower}}$ and $s_{\mathrm{lower}} \leq s \leq 1 - s_{\mathrm{lower}}$ then
        $C \leftarrow \{\phi_j\}_{j=i+1}^m$
        $\ell \leftarrow$ BUILD-TREE($\mathcal{D}|_{\phi_i=\mathrm{T}}$, $C$)
        $r \leftarrow$ BUILD-TREE($\mathcal{D}|_{\phi_i=\mathrm{F}}$, $C$)
        return $(\phi_i, \ell, r)$
    end if
end for
return partition leaf node
```

behavior of both the vector indices (small vector indices do not perform well) and the partition searching.

Once we have accepted the predicate ϕ_i, we can create an internal node by splitting on this predicate. We pass the subset of data that evaluates to T on ϕ_i, $\mathcal{D}|_{\phi_i=\mathrm{T}}$, to a recursive call of BUILD-TREE to construct the left subtree. Likewise, we pass the subset that evaluates to F on ϕ_i, $\mathcal{D}|_{\phi_i=\mathrm{F}}$, to another recursive call to construct the right subtree. In both recursive calls, we pass $\{\phi_j\}_{j=i+1}^m$ as the new atomic predicate list so that we do not reconsider predicates that we already considered at ancestor nodes.

If we exhaust the list of candidate predicates, instead of creating an internal node, we create a leaf node. In our implementation, we also create a leaf node if we have reached the maximum number of partitions or if the current partition is below some minimum partition size, which are both hyperparameters.

After all recursive calls terminate, the initial call to BUILD-TREE returns the root node $\mathcal{T}$ of the partition tree.

3.5 Summarizing partitions

Once we have generated the partition tree, we can locate which partition a given vector lies in by performing simple recursive search. However, for later algorithms, it will also be useful to summarize each partition in a more direct form. Namely, we summarize each partition as a mapping from each attribute name to an integer range.

To formalize this, let $N = \{x_1, \ldots, x_d\}$ denote the set of attribute names where $d := |N|$ denotes the number of attributes. Let $\overline{\mathbb{Z}} := \mathbb{Z} \cup \{\infty, -\infty\}$ denote the set of extended integers.

Each partition can be defined as a unique function p that maps each attribute name to a single integer range

$$p : N \to \{[a, b] \mid a, b \in \overline{\mathbb{Z}}, a \leq b\}.$$

Then partition p contains vector v with attribute values $(z_1, \ldots, z_d) \in \mathbb{Z}^d$ if and only if $z_i \in p(x_i)$ for all $i \in [d]$.

The goal is now to extract a collection of partition functions from the tree $\mathcal{T}$, as summarized by algorithm 3. We initialize the current partition function to $p(x) = (-\infty, \infty)$ for all $x \in N$. We then call GET-PARTITION-RANGES($\mathcal{T}$, p) to get the collection of partition functions. In each recursive call, p represents the current partition function for the current node of the tree. Once we reach a leaf node, we can simply return p as the partition function for that paticular leaf node.

If $\mathcal{T}$ is an internal node, we consider the node's atomic predict $x \geq c$. At this step, we only need to consider what p evaluates to

Algorithm 3 GET-PARTITION-RANGES($\mathcal{T}$, p)

```
if $\mathcal{T}$ is an internal node then
    $(x \geq c, \ell, r) := \mathcal{T}$
    $[a, b] := p(x)$
    $p_{\mathrm{T}}(y) := \begin{cases} [\max(a, c), b] & \text{if } y = x \\ p(y) & \text{otherwise} \end{cases}$
    $p_{\mathrm{F}}(y) := \begin{cases} [a, \min(b, c)] & \text{if } y = x \\ p(y) & \text{otherwise} \end{cases}$
    $L \leftarrow$ GET-PARTITION-RANGES($\ell$, $p_{\mathrm{T}}$)
    $R \leftarrow$ GET-PARTITION-RANGES($r$, $p_{\mathrm{F}}$)
    return $L \cup R$
else if $\mathcal{T}$ is leaf node then
    return $\{p\}$
end if
```

on x. Let $[a, b] := p(x)$ be the current range for x. If the predicate $x \geq c$ happens to be true, then the range of x gets restricted to $[c, b]$ if c is greater than a, and otherwise the range remains $[a, b]$. Thus, the range becomes $[\max(a, c), b]$. Similarly, if $x \geq c$ happens to be false, x gets restricted to $[a, \min(b, c)]$. We thus create two new functions p_{T} and p_{F} with these updated ranges and pass them to the left and right recursive calls respectively.

Once all recursive calls complete, we collect all the partition functions returned by the leaf nodes, giving us a full set of partition functions. For example, fig. 3 depicts the partition ranges at each leaf node.

3.6 Inference

Now that we have created the partitions, it is straightforward to identify which partition a single vector lies in. However, a more complex problem arises when a user makes a query containing a filter ϕ at inference time. In order to benefit from this partitioning system, we need to identify which partitions will never satisfy ϕ so that we can skip performing vector search on those partitions entirely.

To handle this, we consider the function representation of each partition derived in the previous section. For each partition p, we determine if the values in p will always, sometimes, or never satisfy ϕ. To represent these three situations, two-valued logic (T, F) is not enough. Thus, we apply three-valued logic (3VL), which consists of an additional "maybe" (M) truth value, which represents ambiguity between true and false. Truth tables on these values can be defined in a natural way (appendix A). For instance, $\neg\mathrm{M} = \mathrm{M}$, $\mathrm{T} \wedge \mathrm{M} = \mathrm{M}$, and $\mathrm{F} \vee \mathrm{M} = \mathrm{M}$. Our function PARTITION-MAY-SATISFY will return T, M, F when p always, maybe, or never satisfies ϕ, respectively.

Algorithm 4 effectively breaks ϕ down into an expression tree, evaluates p on the leaf nodes (atomic predicates) of the tree, and then applies the 3VL truth table rules to arrive at a final answer. For instance, suppose we are evaluating p on the atomic predicate $x \geq c$. $[a, b] := p(x)$ gives the range of values that x may lie in. If both $a \geq c$ and $b \geq c$, then $x \in [a, b]$ is *always* $\geq c$; thus, we return T. However, if $b \geq c$ but $a < c$, then x *may or may not* be $\geq c$; thus, we return M. Finally, if both $a < c$ and $b < c$, then x is *never* $\geq c$.

Algorithm 4 PARTITION-MAY-SATISFY(ϕ, p)

```
if Atomic($x, \geq, c$) := $\phi$ then
    $[a, b] := p(x)$
    if $b \geq c$ then
        if $a \geq c$ then
            return T
        else
            return M
        end if
    else
        return F
    end if
else if And($\phi_1, \dots, \phi_m$) := $\phi$ then
    return $\bigwedge_{i \in [m]}$ PARTITION-MAY-SATISFY($\phi_i$, $f$)
else if Or($\phi_1, \dots, \phi_m$) := $\phi$ then
    return $\bigvee_{i \in [m]}$ PARTITION-MAY-SATISFY($\phi_i$, $f$)
else if Not($\phi'$) := $\phi$ then
    return $\neg$ PARTITION-MAY-SATISFY($\phi'$, $f$)
end if
```

As another example, suppose we have two attributes x, y and are evaluating the partition function $p(x) = [500, \infty], p(y) = [300, \infty]$ on the predicate $\phi = (x \geq 400 \wedge \neg(x \geq 800)) \vee \neg(y \geq 200)$. Then

$$\begin{aligned} \phi(p) &= (x \geq 400 \wedge \neg(x \geq 800)) \vee \neg(y \geq 200) \\ &= (\mathrm{T} \wedge \neg(\mathrm{M})) \vee \neg(\mathrm{T}) \\ &= (\mathrm{T} \wedge \mathrm{M}) \vee \mathrm{F} = \mathrm{M} \vee \mathrm{F} = \mathrm{M} \end{aligned}$$

After determining the truth value h of p on ϕ, we can safely skip the partition if $h = \mathrm{F}$. However, if $h = \mathrm{T}$ or $h = \mathrm{M}$, we must still search the partition.

Interestingly, if $h = \mathrm{T}$, then since ϕ always evaluates to true on the partition, we can ignore filtering entirely when performing vector search on that partition. This requires the vector database to support this, and we currently delegate this feature to future work (section 6.2).

3.7 Implementation

We implemented our partitioning system in Python to be used in tandem with the Milvus vector database partitioning API. Users can call functions on their raw data to characterize their workload and generate the partition tree. They can then create these partitions within Milvus and insert vectors in the correct partitions according to the true. At inference time, they can feed an expression tree of their filter to a function to determine which partitions must be searched. These partition names are then included in the vector search call to Milvus to prune away unnecessary partitions in the search.

While we implemented our algorithm at the user level, we do not believe it has significant overhead as opposed to a system-level implementation. However, there are certain optimizations that we have not implemented that are only possible at the system-level, which are addressed further in section 7.

4 Experiments

All experiments run on the Milvus docker installation hosted on an Ubuntu 24.04.2 LTS x86 machine with an AMD Ryzen 9 7900x3D 5.660GHz CPU, no GPU, 32 GB of RAM, and an internal NVME SSD. The experiments are available for download on our public GitHub. We also use the IVF vector index [11] in all of our trials. We used this index instead of alternatives such as HNSW [10] since in preliminary experiments with HNSW indices, we almost always obtained a perfect recall. We decided that IVF indices would give more insightful results on how performance changes with partition sizes, since it doesn't achieve as high of recall scores (though this comes at the cost of build time).

4.1 Synthetic dataset

Unfortunately, there are not many open source labeled vector datasets of sufficient size. Most research, including the results in [6, 13, 14, 16, 17] use either proprietary datasets or open source vector-only datasets augmented with synthetic attributes using the approach described in [13]. The Qdrant DBMS has developed a standardized set of benchmarks [1], however there are three issues with this current benchmark which lead us to believe it is not suitable for our application: (1) ideally each dataset would have many millions of vectors, since many indexing techniques exhibit high fixed costs, but Qdrant's datasets were much smaller; (2) our system currently only works with numeric attributes but some of Qdrant's attributes were categorical, meaning they were not enougnh numeric attributes for our algorithm to use; (3) while this will be discussed more in the next section, even the datasets from actual data sources are still arbitrarily filtered, and the workloads that correspond to them are still synthetic, so there is not much benefit in using their dataset as opposed to synthetically generating our own dataset.

For this project, we use the SIFT1M [7] and SIFT1B [8] datasets augmented with an a set of four i.i.d. `uint16` attribute $a_1, a_2, a_3, a_4 \sim \text{Uniform}\{0, \dots, 999\}$. While this is an unrealistic assumption, it has two major benefits: (1) it allows us to easily model arbitrary filters, e.g. the canonical filter with 25% selectivity is $\phi(x) = (x < 250)$, and (2) since this is the approach used by Milvus [13], Qdrant [1], and current ANN research to synthetically generate datasets, it makes our results more directly comparable.

We found using a subset of 10-million datapoints from the SIFT1B dataset ran well on our hardware. This consisted of 10-million 128-dimensional `uint8` vectors, so unless otherwise specified this is the experiment setting used. The SIFT1B dataset also consists of 10,000 test datapoints which were used to measure the recall of our algorithm.

4.2 Synthetic workflow

While there has not been much work on workload-aware filtered vector queries, there is a large body of literature on classical OLAP queries. This research, for example [2, 4, 9], often use variants of the TPC-H benchmark [3], e.g. the Star Schema Benchmark [12]. However, we do not believe this fits our use case for two main reasons.

First, much of the complexity of the TPC-H and related benchmarks come from the large quantity of joins and aggregations that need to be done by the queries [3]. However, our work does not modify how this is carried out (since we assume the attribute table is un-normalized, as is typical in vector database applications), so the main complexities of TPC-H are not relevant for us. Second, the data in the TPC-H benchmark is uniformly sampled similarly to our construction above [3], and the filters are generated by substituting random values into template queries. This is something we can easily replicate ourselves, with the added benefit of having full control over the filter distribution of the workload, which allows us to compare the performance against any theoretical guarantees.

Therefore, we decided to synthetically generate a dataset. While we use a similar approach to [1, 13] we generate a more complex workload since we believe that will give a more equitable comparison between algorithms. The ideal workload we seek to model is one where there are a few attributes that are roughly equally used, but some are used noticably mora than others in filters. Additionally, each attribute should have some number of filters that can use the attribute. To accomplish this, we used the following construction of a synthetic workload. First, we constructed the dataset as detailed in section 4.1. Next, we define the following discrete exponential distribution:

$$\mathrm{DExp}(n \mid \lambda, N) = \frac{\exp(-\lambda n)}{\sum_{i=1}^{N} \exp(-\lambda i)}.$$

We now detail the probabilities used to construct each query. In essence, for each attribute we sample a λ hyperparameter from a normal distribution to use in the exponential distribution. We do this to provide some variance across trials, and we chose what we estimated to be rational choices of these distributions.

Let us first consider sampling atomic attributes. An atomic predicate $\phi(a_i \leq t)$ consists of three components: an attribute a_i, an operator, and a threshold t. Without loss of generality, we can assume that the operator is always $\leq$ (since we uniformly sample the attributes). For the threshold, following the synthetic workload in [1] we use percentiles $\{10\%, \ldots, 90\%\}$. For each attribute a_i, we randomly shuffle the percentiles to an order σ_i, in order to choose the thresholds with different exponential decaying probabilities. To sample the attribute, we first sample the parameters $\lambda_a, \lambda_1, \ldots, \lambda_4 \sim \mathcal{N}(0.25, 0.1)$ to define out workload. We then model

$$\begin{aligned}\Pr(\phi(a_i \leq t)) &= \Pr(t \mid a_i, \lambda_i) \Pr(a_i \mid \lambda_a) \\ &= \mathrm{DExp}(\sigma_i(t) \mid \lambda_i, N = 9)\, \mathrm{DExp}(i \mid \lambda_a, N = 4).\end{aligned}$$

Now we consider sampling filters. There are 4 types of filters that we sample uniformly at random: no filter, CNF filter, DNF filter, and singular filter. No filter is clear (just a standard vector search), and singular filter is just a single atomic predicate sampled using the procedure described above. CNF filters are the intersection of several atomic predicates, and likewise DNF filters are their union. To sample each of these, we first sample how many atomic predicates the CNF/DNF filter is composed of, and then sample an i.i.d. atomic predicate and combine them into the final CNF/DNF filter. We sample the number of atomic predicates in a CNF/DNF filter as $n \sim \mathrm{DExp}(\lambda_n, N = 4)$ where $\lambda_n \sim \mathcal{N}(0.25, 0.1)$. While this sampling procedure is more complex than that in [1, 13] we believe it's a rough approximation of more complex practical workloads.

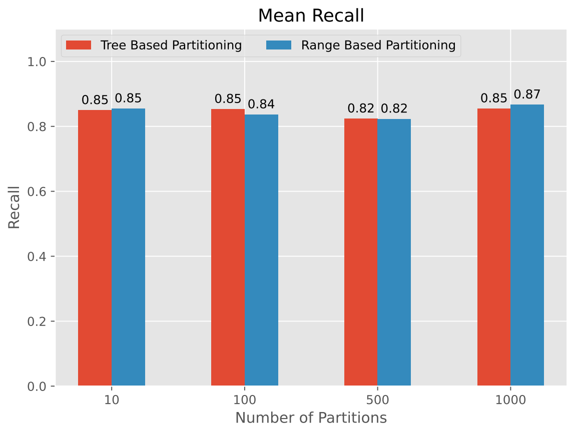


Figure 4: A comparison of our tree-based algorithm and the classical range-based algorithm in terms of mean recall over SIFT10M.

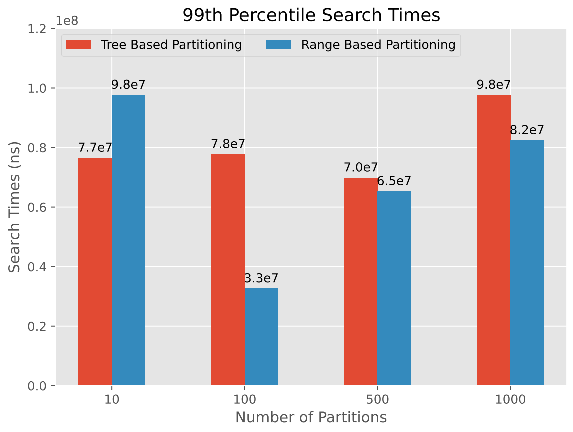


Figure 5: A comparison of our tree-based algorithm and the classical range-based algorithm in terms of the 99th percentile search times in nanosconds over SIFT10M.

5 Results

We ran experiments using the datasets and workloads described in section 4 comparing the number of partitions against search time and recall, and we compared our algorithm's performance against the performance of the classical greedy range-based partitioning system. We used a 5M subset and a 10M subset of the SIFT dataset, augmented as described in section 4. The results in figs. 4 to 5 are averaged over three trials on different synthetic datasets. We chose to measure the 99th percentile search times instead of the mean search times since that is the approach taken in [5], however we measure the mean recall to be consistent with other work.

First, note that across both trials there is no significant difference between the recall of the two methods. Importantly, this shows that changing the number of partitions did not have a significant impact on performance on the IVF index. In non-vector databases, this

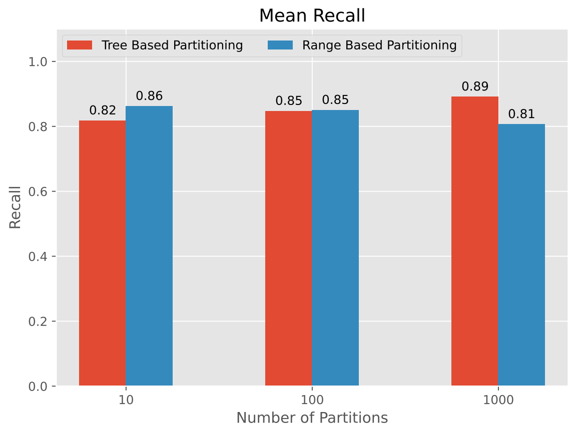


Figure 6: A comparison of our tree-based algorithm and the classical range-based algorithm in terms of mean recall over SIFT5M.

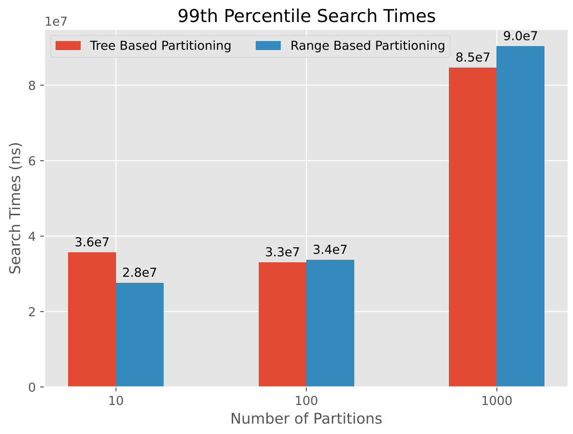


Figure 7: A comparison of our tree-based algorithm and the classical range-based algorithm in terms of the 99th percentile search times in nanosconds over SIFT5M.

is obvious, however ANN searches have the potential to change performance depending on partition size.

However, the relationship with search times is not so clear. When testing over 5 million datapoints, the search times are roughly comparable, however when testing over 10 million datapoints, the tree based partitioning method performs far worse than the range-based algorithm. This difference is extremely noticeable when considering the 100 partition case, which seems to be the optimal number of partitions for the range-based algorithm. There are two reasons we think this is happening. First, we hypothesis that our algorithm is more susceptible to changes in partition sizes than the range-based approach. Second, we think this effect has been exacerbated by poor hyperparameter tuning for our algorithm. We suspect that with more time to tune parameters such as minimum selectivity, the tree-based approach would have performed much better.

In summary, while these results show that our algorithm does not outperform the range-based approach, we think that with better hyperparameter tuning its performance could improve to match that of the range-based algorithm.

6 Limitations

While our experiments demonstrate the viability of a workload-aware tree-based partitioning algorithm, there are several limitations of our results. As explained in section 4.1, we use a synthetically labeled dataset and a synthetic workload, meaning our results may not generalize to real-world datasets and workloads. Additionally, other papers have run their experiments on datasets with up one billion vectors, whereas due to hardware limitations, we used 10 million vectors. Thus, we remain skeptical as to the quality of our results.

6.1 Hyper-parameter tuning

Unlike the range-based partitioning algorithm, our tree-based method has three hyper-parameters. However, due to time constraints, we were unable to fully tune these. These hyper-parameters include partition size, the filter selectivity threshold, and the filter frequency threshold. We suspect that with more hyperparameter tuning on a wider variety of datasets, our performance would increase.

6.2 Implementation

Another limitation is due to our implementation. Due to time constraints, we currently only support numeric attributes with range-based predicates, so we cannot run our algorithm on datasets or workloads with categorical features. Additionally, for ease of use we implemented our algorithm at the user-level rather than directly integrating with Milvus. As mentioned in section 3.6, if the truth value of p on ϕ is true, then we can ignore filtering entirely on that partition when performing our vector search. However, we cannot implement this logic without integrating with Milvus directly.

6.3 Algorithm

Our algorithm as a whole also has some limitations, mainly due to how it evaluates and uses atomic predicates.

First, the process of splitting filters into atomic predicates and sorting the atomic predicates by frequency inherently loses some information about the relationship between the predicates, for example of ϕ_1 frequently co-occurs with ϕ_2. We think that by incorporating this co-occurrence information we could calculate better splits.

Second, in the case of many similar predicates (e.g. $x \leq 49.9$, $x \leq 50$, and $x \leq 50.1$) each predicate is considered individually. However, this may be suboptimal splits depending on the frequency of each predicate. It can also be difficult to calculate the frequency of each predicate for real-valued queries, since it's possible that all three of these predicates could select the same (or almost the same) data. An ideal algorithm would incorporate some method of merging similar predicates such as these.

Additionally, our current approach doesn't generalize well to categorical features. While it is possible to support categorical attributes in our current algorithm, we believe that our partitioning scheme wouldn't be efficient. Once our algorithm selects a predicate to partition on, the tree is split such that one side satisfies the predicates and the other does not. However, it specifically relies on the binary split of values in an attribute being less than a given value which is undefined on categorical variables. This approach doesn't work well for categorical features, since if we have a large number of possible variables our algorithm can currently only consider atomic predicates where each possible value appears alone. This would lead to large trees that don't exploit the co-occurrences of the features like we do with numerical ranges.

7 Future work

Building on the findings of our research, the limitations of our current algorithm and experiment, and initial goals left unreached, there are several avenues for future work that we discuss. Some of these areas for research aim to simply further improve the performance of our algorithm, while others serve to introduce new functionality into the algorithm or expand upon the use cases for our partitioning.

7.1 Further testing

As mentioned in the 4 and 6 sections of this paper, we had difficulties in testing our algorithm on a dataset and workload representative of real world data and queries. While we may not have access to proprietary information, we leave proper evaluation of our algorithm as future work for those with such access. We also did not have the ability to run our algorithm across a dataset with 1B vectors, nor were we able to integrate our partitioning method into the system-level code of Milvus.

Furthermore, due to time constraints, we were not able to tune the hyper-parameters that our algorithm relies on to a satisfactory extend. As such, our results do not necessarily represent the upper-bound of increase in performance that our algorithm may provide. While more complete testing would be helpful to better understand the performance of our partitioning scheme in the real world, we maintain that our preliminary results still show the potential of our tree-based partitioning.

7.2 Kernel Density Estimation

Another method that could improve partitioning quality is Kernel Density Estimation (KDE). In essence, KDE is a statistical method to estimate the underlying distribution of a sampled dataset. Currently, our algorithm uses the frequencies of atomic predicates to create an appropriate set of partitions. However, as explained in section 6.3 this can fail given similar numeric predicates, for example with thresholds of 49.9, 50, and 50.1. KDE would allow us to estimate the continuous distribution that the thresholds were drawn from, and use the peaks of that distribution as a new atomic predicate with a frequency proportional to the KDE distribution. While this new predicate would not be a "true" atomic predicate in that it could be decomposed further, it would allow the model to handle similar thresholds by treating them as the same predicate, which would lead to partitions that are, on average, more optimal over that range.

7.3 Online partitioning

One of the original goals for our project was exploring the possibility of online repartitioning based off of changing datasets and workload statistics. Unfortunately we did not have the time to implement this, and thus we leave it as an avenue for further work. The first case, in which our dataset is changing over time, is fairly simple and inexpensive to implement under the assumption that we are only adding data instead of removing data (which we think is reasonable for many applications, especially RAG applications). As new pieces of data are gathered, we can simply add them to the appropriate partition using either our algorithm or techniques described in [9]. This allows for users to continuously query the dataset even as new pieces of data are inputted, however it doesn't address the complexity involved in repartitioning to support drastic dataset changes, as the greedy algorithm from [9] can struggle to handle large distribution shifts.

Adjusting our partitioning scheme as the workload changes more complex. Unlike with a dynamic dataset, it is difficult to dynamically modify the tree while preserving performance. One approach would be to track statistics on the workload, and once some critical point is met repartition the dataset. However, this would be very expensive, so generally isn't feasible. Another approach would be to adapt the greedy method form [9], however this would require some future work to assess feasibility.

8 Conclusion

In summary, we have explored several filtered approximate k-nearest neighbor search algorithms, and attempted to implement an algorithm inspired by classical relational partitioning methods in the Milvus vector database. While our results did not significantly outperform the standard range-based partitioning algorithm in use by most modern vector databases, we do think that exploring better hyperparameter tuning on a larger variety of datasets could improve performance and is an interesting direction of research.

References

[1] [n. d.]. Vector Database Benchmarks. https://qdrant.tech/benchmarks/#filtered-search-benchmark

[2] Nino Arsov, Goran Velinov, Aleksandar S. Dimovski, Bojana Koteska, Dragan Sahpaski, and Margina Kon-Popovska. 2019. Prediction of Horizontal Data Partitioning Through Query Execution Cost Estimation. *arXiv e-prints*, Article arXiv:1911.11725 (Nov. 2019), arXiv:1911.11725 pages. doi:10.48550/arXiv.1911.11725 arXiv:1911.11725 [cs.LG]

[3] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 61–76.

[4] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 48–57. doi:10.14778/1920841.1920853

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. (2007). https://www.amazon.science/publications/dynamo-amazons-highly-available-key-value-store

[6] 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. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 3406–3416. doi:10.1145/3543507.3583552

[7] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine*

Intelligence 33, 1 (2011), 117–128. doi:10.1109/TPAMI.2010.57

[8] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 861–864. doi:10.1109/ICASSP.2011.5946540

[9] Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fabio Porto, and Patrick Valduriez. 2012. Dynamic Workload-Based Partitioning for Large-Scale Databases. In *Database and Expert Systems Applications*, Stephen W. Liddle, Klaus-Dieter Schewe, A. Min Tjoa, and Xiaofang Zhou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–190.

[10] Yu. A. Malkov and D. A. Yashunin. 2018. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. arXiv:1603.09320 [cs.DS] https://arxiv.org/abs/1603.09320

[11] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Umar Farooq Minhas, Jeffery Pound, Cedric Renggli, Nima Reyhani, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. 2024. Incremental IVF Index Maintenance for Streaming Vector Search. arXiv:2411.00970 [cs.DB] https://arxiv.org/abs/2411.00970

[12] Pat O'neil, Betty O'neil, and Xuedong Chen. 2009. The Star Schema Benchmark (SSB). (01 2009).

[13] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2614–2627. doi:10.1145/3448016.3457550

[14] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2022. Navigable Proximity Graph-Driven Native Hybrid Queries with Structured and Unstructured Constraints. arXiv:2203.13601 [cs.DB] https://arxiv.org/abs/2203.13601

[15] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.* 14, 11 (July 2021), 1964–1978. doi:10.14778/3476249.3476255

[16] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3152–3165. doi:10.14778/3415478.3415541

[17] Wei Wu, Junlin He, Yu Qiao, Guoheng Fu, Li Liu, and Jin Yu. 2022. HQANN: Efficient and Robust Similarity Search for Hybrid Queries with Structured and Unstructured Constraints. arXiv:2207.07940 [cs.DB] https://arxiv.org/abs/2207.07940

[18] 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. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 377–395. https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi

[19] Wan-Lei Zhao, Hui Wang, Peng-Cheng Lin, and Chong-Wah Ngo. 2022. On the Merge of k-NN Graph. *IEEE Transactions on Big Data* 8, 6 (Dec 2022), 1496–1510. doi:10.1109/TBDATA.2021.3101517

A Truth tables for three-valued logic

p	$\neg p$
T	F
M	M
F	T

p	q	$p \wedge q$
T	T	T
T	M	M
T	F	F
M	T	M
M	M	M
M	F	F
F	T	F
F	M	F
F	F	F

p	q	$p \vee q$
T	T	T
T	M	T
T	F	T
M	T	T
M	M	M
M	F	M
F	T	T
F	M	M
F	F	F