ABSTRACT

Title of Thesis: Approximate Nearest Neighbor Search with Filters
Benjamin Landrum
Master of Science, 2024

Thesis Directed by: Laxman Dhulipala
Department of Computer Science

Approximate nearest neighbor search (ANNS) on high-dimensional vectors is a fundamental primitive used widely for search over neural embeddings of unstructured data. Prior work on ANNS has produced indices which provide fast and accurate search on datasets up to billions of points, but are not well suited to queries restricted to some subset of the original dataset.

Filtered ANNS is a formulation of the problem which adds metadata to points in the dataset which can be used to filter points at query time. This setting requires indexing a dataset in a metadata-aware way to support filtered queries. Filtered ANNS is important for applications such as product and image search, and necessary to give recently popular 'vector databases' functionality similar to more traditional tabular databases.

This work concerns two versions of the filtered ANNS problem. The most popular formulation in prior work associates points with boolean metadata in the form of labels and filters queries using a boolean predicate on these labels. In this setting, we present a novel index with state-of-the-art performance for queries with filters requiring either one label or both of a pair of labels which won a large benchmarking competition's track focused on the problem. We also

introduce a novel formulation of filtered ANNS called ‘window filtered’ ANNS, in which points are associated with a continuous metadata value (in practical use, this corresponds to a timestamp, measure of popularity, etc.), and queries are filtered to a range of metadata values. In addition to describing the problem, we present a practical and theoretically motivated index which handily outperforms baselines.

Approximate Nearest Neighbor Search with Filters

by

Benjamin Landrum

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2024

Advisory Committee:
Laxman Dhulipala, Chair/Advisor
Alan Sussman
Amol Deshpande

Dedication

To my friends and collaborators in Laxman's group, who I will dearly miss.

Acknowledgments

I would like to thank everyone with whom I've had technical conversations about ANNS, who all influenced my thinking on the topic in their own way.

Primarily, I'd like to thank my advisor, Laxman Dhulipala, who introduced me to the problem and has been an invaluable resource, collaborator, and friend throughout the process of this research.

This work would not have been possible without my collaborators at UMD and elsewhere. I would like to especially thank Magdalen Dobson Manohar, who was an uncommonly knowledgeable and accommodating mentor the entire time I have been working on ANNS. Without her work on ParlayANN and expertise, much of the work described here would have been impossible.

I would like to thank Mazin Karjikar, my mentee, for his contributions to the work in Chapter 3.

Shangdi Yu, Josh Engels, and their advisor Julian Shun were all essential to the work described in Chapter 4, and I would like to especially thank Josh for taking point and making it possible for this idea to go from conversation over lunch to submitted paper in less than 2 months during the busiest time of the year.

Lastly, I would like to thank my parents, who funded the more-or-less gap year in which I completed this MS.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

In recent years, the development of powerful neural models for generating vector embeddings of unstructured data has sparked significant interest in the problem of search over high-dimensional vectors. Armed with such embeddings, it's possible to find items in a dataset which are relevant to some query by identifying the nearest vectors to the embedding of that query. These embeddings have sparked a renaissance in a broad range of applications, from image search to recommendation. However, the datasets searched for these applications are so large, the dimensionalities of the vectors so high, and the latency requirements so strict, that naive methods for finding near neighbors of a vector exactly are entirely impractical.

This is the domain of *approximate nearest neighbor search* (ANNS). Instead of finding the exact nearest neighbors of a query vector in a dataset, it is often sufficient to find vectors which are almost as close as the true nearest neighbors without degrading user experience. This is due in part to the fact that the problem of similarity search over unstructured data is already rather fuzzily defined, such that any set of results acceptable to a user is not particularly sensitive to small perturbations, and because it can generally be assumed that would-be neighbors which are not in the top k desired are likely still close to the query, and therefore also meaningfully similar to it.

The major advantage of approximate search is its speed. Empirically, it is possible to do

approximate search on an indexed dataset with beyond 99.99% accuracy in orders of magnitude less time than it would take to search the same points exactly. [1] In practice, much lower accuracies are generally needed, which can be achieved further orders of magnitude faster. This advantage increases swiftly as the size of the dataset grows, an important property as vector search is increasingly deployed over billions [2, 3] or even trillions [4] of vectors.

While traditional ANNS is comparatively well studied and can be served efficiently at scale, the related problem of *filtered* ANNS, which constrains search to a subset of the data matching some condition on metadata associated with the vectors, is a new and active area of research. The core of what makes filtered ANNS difficult is the problem of building an index that is still effective when large sections of the data set are not considered valid neighbors for a query. In many vector database products (i.e. [5–7]), filters are considered a black box, arbitrarily including or excluding points in a manner which is entirely unpredictable at build time when an index is being constructed.

Such approaches leave valuable information on the table. Effectively leveraged, metadata can be a powerful tool for improving performance of filtered ANNS, facilitating indices which anticipate the needs of filtered queries and retain robust performance when answering them. However, in cases where the number of possible filters is polynomial or worse (such as searching for points which have both of a pair of labels when the number of possible labels is large), it is impractical to index the points associated with every possible filter ahead of time. This leads us to the question:

Can we build metadata-aware indices that effectively answer filtered queries when there are many possible filters?

In this work, we endeavor to show that the answer is emphatically **yes**. Not only are such indices feasible, they dominate alternatives with vastly improved throughput at the same recall.

1.1 Applications of ANNS

While ANNS can be applied broadly in any setting where it's desirable to find vectors similar to some query in a large dataset, in practice, essentially all applications use vectors which are representations of unstructured data. What follows is a brief overview of common techniques for generating such vectors and fundamental concepts convenient for understanding how ANNS solves real-world similarity search problems.

Metric Learning Metric learning is the problem of constructing vector representations of data so that distance comparisons between representations are a meaningful proxy for similarity between elements of the original dataset. This problem has been studied extensively since at least the early 2000s [8], and work on neural embeddings stable under small changes goes back to at least the early 1990s [9]. Learned embeddings facilitate representing all manner of similarity search and information retrieval problems as vector search. Prominent applications include embedding text [10], images [11], and graphs [12].

Contrastive learning [13, 14] serves as an illustrative example of a typical and popular model for metric learning. Given a dataset of pairs labeled to be similar, the task is to build a vector representation such that the elements of the pairs provided map to vectors which are closer together than the vectors for a randomly sampled pair. A neural network which maps the data referred to by the pairs onto the vector space is initialized, and then trained using the provided pairs. At each training step, gradient descent is used to update the weights of the neural network

to maximize the difference between the distance between elements of a provided 'positive' pair, and the distance between the elements of a randomly sampled 'negative' pair. This maximizes similarity of the vector representations of the elements labeled to be similar, and minimizes the similarity of elements which are not known to be similar.

Metric learning does not have to be restricted to pairs of items which can be embedded by a single model. A *joint embedding space* is a a common vector space which multiple embedding models output onto, such that the similarity of embeddings generated by different models can be compared and is still a meaningful measure of the similarity of the objects being embedded across domains. The most conspicuous example is OpenAI's CLIP [15] model for embedding images and captions in a joint embedding space as a primitive for text-conditioned image generation. Because the caption and image embedding models are trained to make captions embed similarly to the corresponding images while separating the embeddings of unrelated items, distance in the joint embedding space is a useful proxy for semantic similarity between text and images. As such, nearest neighbors of the embedding of a piece of text are likely similar pieces of text, or images with content related to that of the text.

This underlies the concept of 'two-towers' architectures in recommendation systems, where the user and item are embedded by separate models (the 'towers' in question) into a joint embedding space and recommendation is treated as ANNS, where the items embedded closest to a user's embedding are served to that user as recommendations. [16]

RAG Recent excitement about vector search has come in part from interest in Retrieval Augmented Generation (RAG), a method for augmenting large language model (LLM) based question answering systems with an external knowledge base. When a RAG-equipped system receives a

question to be answered, it embeds it, and queries a knowledge base for text which has a similar embedding to the question. The best matches are then added to the context window of the LLM, allowing it to ground its answers. [17] This approach, introduced in [18], improves the accuracy and interpretability of LLM-based agents, and is widely deployed in practice. The intermediate step of identifying pieces of text relevant to a query is ANNS in the embedding space, and has in part driven increased popularity of the 'vector databases' that provide ANNS indices as a service.

1.2 Outline

This work will provide background on the field of ANNS, and then present two projects in filtered ANNS. The remainder of this chapter will present background on and popular approaches to the vanilla ANNS problem, as relevant to the problem of filtered ANNS. Chapter 3 will present the IVF^2 index, which was developed by the author and collaborators at UMD and Carnegie Mellon for the NeurIPS'23 Practical Vector Search Challenge's 'Filter' track [19]. Chapter 4 will present work by the author and collaborators at UMD and MIT on the formulation of and a novel index for window filtered ANNS. This work is described in a paper which is at the time of this writing under submission at ICML, and should be cited thusly: [20].

Chapter 2: Background & Preliminaries

2.1 Preliminaries

2.1.1 ANNS Paradigms

The term ANNS is used broadly to describe approximate vector search, but encompasses two distinct notions of 'near neighbors': k-nearest neighbors (k-NN) and ϵ-nearest neighbors (ϵ-NN).

k-NN search refers to a query which returns the k vectors in a dataset which are most similar to the query vector for some small $k \in \mathbb{N}$. This is the more popular version of the ANNS problem, and all unqualified uses of the term 'ANNS' in this work can be assumed to refer to approximate k-NN search.

ϵ-NN (sometimes ϵ ball-NN) search refers to a query which returns all vectors within a ball of radius ϵ centered on the query vector for some small distance ϵ. The applications where ϵ-NN is used are relatively limited due to the limitations of using the absolute value of a similarity measure provided by vector embeddings. While such embeddings provide that similarity in the vector space corresponds to some notion of similarity in the original data, the actual distances within the space are generally only meaningful in relative terms. However, ϵ-NN search is still useful in settings where the presence of points within an empirically determined ϵ is a valuable

signal. An example of this is duplicate detection, exemplified by Meta using it to identify posts of images known to violate policies while being robust to superficial edits [21].

The distance measure used for ANNS can also vary, but Euclidean distance and (negative) inner product are most common. When not specified, distances in this work are implied to be Euclidean.

2.2 ANNS Background

ANNS is a reasonably mature area of research, and as approaches for filtered ANNS build heavily on those for unfiltered vector search, it's valuable to understand broadly the popular approaches for ANNS from prior work.

2.2.1 Partitioning-based Approaches

Many popular approaches to ANNS indexing construct partitions of the vector space; the intuition behind such approaches being that a point's nearest neighbors are expected to fall in the same partition.

2.2.1.1 Locality Sensitive Hashing

For many years, research in high-dimensional ANNS was dominated by locality sensitive hashing (LSH) [22–28] and derived approaches. In the most basic formulation of LSH, B random hyperplanes are chosen which divide the dataset in such a way that neighbors of a point are on the same side of the hyperplane as that point with high probability. Sidedness queries on these hyperplanes are then used to determine a B-bit hash for any point, and points with the same hash

are stored together. At query time, the query vector is hashed using the same hyperplanes, and compared with points in the corresponding bucket.

LSH was long favored for the theoretical and practical efficiency of hashing as a means of localizing search to a bucket, and the theoretical guarantees it provides are stronger than any competing paradigms. However, highly optimized LSH implementations are not competitive in practice with newer heuristic approaches [3] to ANNS.

2.2.1.2 IVF

The IVF index is a generalization of the concept of an inverted file index to vectors. A classical inverted file index is a means for searching for documents which contain some word. Instead of naively traversing each document for usages of the word in question at query time, we precompute a list of documents containing each word that could be searched, and return said list to answer the corresponding queries. In the setting of vector search, we cluster the dataset to partition it, typically with k-means clustering, and store a representative point and list of assigned points for each cluster. At query time, we compare the query vector to the representative of each cluster, and then exhaustively search the points assigned to the nearest few clusters. The intuition behind the IVF index is that the neighbors of a point are likely to be assigned to the same cluster as that point, but as the dimensionality of the dataset increases, this assumption becomes increasingly weak. IVF is commonly combined with quantization methods for compressing vectors (see section 2.2.3). [29–33]

2.2.2 Graph-based Approaches

The current state of the art for billion-scale ANNS is dominated by graph-based approaches [3], which build a graph over the points in the dataset which empirically converges to near neighbors of a query point under beam search.

A beam search of size B is a generalization of a greedy search. Given a query x, we start at a start node s and "explore" s by adding neighbors of s to a queue. This queue consists of only the closest B points to x we have seen so far, explored or unexplored. We continually explore the closest unexplored node from the queue to x until all nodes in the queue are explored. By increasing B, the beam search explores more points and is more likely to find a better nearest neighbor of x.

2.2.2.1 HNSW

Hierarchical Navigable Small Worlds (HNSW) is a graph-based index which attempts to generalize the idea of a skip list [34] to vector search. The bottom level of the hierarchy contains every point in the dataset, and at each level the surviving points each have a constant probability of surviving into the next level. This process repeats until one point remains, which becomes the starting point for search at query time. At each level, we build a Navigable Small World (NSW) graph, which is defined as any graph in which a greedy traversal from one vertex to another is expected to take a number of steps polylogarithmic in the size of the graph. At query time, greedy search is performed to convergence at each level, and traversal of the next level down begins at the nearest neighbor found in the previous level. [35]

2.2.2.2 HCNNG

HCNNG is an approach for building empirically performant search graphs built on hierarchical clustering techniques. The hierarchical clustering used is generated by a procedure in which we sample two random points, split the dataset on the hyperplane equidistant to these points, and recurse on each side until a partition is smaller than some cutoff size. We then build a degree limited minimum spanning tree (MST) on the points in each partition. This process is repeated on the order of 30 times, and the graph ultimately used is the union of all the resulting MSTs. Each MST is empirically a connected and reasonably navigable graph over the points in the corresponding partition, and because the partitions generated by repeating the process will tend to overlap between iterations, the graph formed by the union of the MSTs tends to be globally navigable. [36]

2.2.2.3 Vamana

The Vamana search graph construction algorithm incrementally builds a graph which approximates the sparse neighborhood graph (SNG) [37], which is itself an approximation of the relative neighborhood graph (RNG) [38]. In the RNG, two points are connected by an edge if and only if there does not exist a third point closer to either point than they are to each other. The SNG is constructed by essentially running the Robust Prune procedure described in the following paragraph on every point, with the remainder of the dataset as the candidate list and $\alpha = 1$

The algorithm used to approximate the SNG neighborhood of a point in Vamana is called Robust Prune, and proceeds by considering each potential neighbor p in order of increasing distance from the point in question x. Let $\alpha \in \mathbb{R}^{\geq 1}$ be a parameter controlling the sparsity of the

graph, with larger values of α corresponding to sparser graphs, and values closer to 1 more nearly approximating the SNG. The nearest candidate is always added as a neighbor. Once a candidate p is added as a neighbor of x, we remove all further candidates which are not at least α times closer to x than they are to p. This procedure is repeated until the candidate list is exhausted. [2]

To construct the "slow-preproccessing" variant of Vamana, which is defined in [39] and provides for some theoretical guarantees about its performance used in Section 4.4, we perform a procedure which is equivalent to running Robust Prune on every point in the dataset with all other points as candidate neighbors, being distinct from the SNG due to the inclusion of α.

2.2.3 Quantization-based Approaches

Mostly orthogonal to the problem of constructing an index over a set of vectors is the problem of constructing compressed representations of the vectors which approximately preserve distances while improving cache utilization, memory footprint, and comparison time. Typically, quantization approaches are evaluated on their performance in an IVF index [29, 40, 41].

2.2.3.1 Product Quantization

Most popular approaches for quantization in ANNS are derived from Product Quantization (PQ), which represents vectors by clustering subspaces independently. When learning the representation, the vector's elements are divided into subspaces of consecutive dimensions. Each subspace is clustered with k-means, where $k = 2^b$. When encoding a vector, the dimensions corresponding to a given subspace are replaced with a b bit integer representing the index of the centroid in the clustering of the subspace nearest to that vector's components in the subspace.

When decoding a vector, we simply replace each centroid index with the centroid it refers to, constructing a vector of the same dimensionality as the original. [29]

2.3 Filtered ANNS

An important problem closely related to traditional ANNS is filtered ANNS, which associates metadata with the points in the dataset, and provides predicates at query time which restrict search to a subset of the data satisfying some condition on the metadata. The bulk of prior work on filtered ANNS considers metadata in the form of binary labels which can be assigned to points, and while we will introduce an alternative formulation of filtered ANNS restricting search to a range in a continuous metadata value in Chapter 4, the remainder of this section will describe the boolean label case discussed in prior work.

The two naive approaches to filtered ANNS are *prefiltering*, where the dataset is restricted to elements matching the filter before a spatial search over the remaining elements, and *postfiltering*, where results from an unfiltered search are restricted to those matching the filter.

While filtered ANNS is a core component of commercial vector databases such as Pinecone [42], Weaviate [43], QDrant [44], Milvus [45], and many more, there is relatively little existing academic work on filtered nearest neighbor search.

2.3.1 OR predicates

Two existing works provide algorithms tailored for the case of a single label or an OR of two or more labels. NHQ [46] supports single filter queries by modifying the distance function to treat points with shared labels as closer together, thus making them more likely to be returned

during a filtered search. FilteredDiskANN [47] modifies the DiskANN [2] graph index by building a separate Vamana graph for each label and then merging them into one graph. They control the degree of the final graph using a filter-aware pruning strategy that attempts to ensure that the subgraph corresponding to each label remains connected. They also modify the graph search routine to only visit vertices that satisfy one or more of the labels being searched for. They furthermore provide a (less performant) dynamic version of the algorithm supporting insertions and deletions that builds the final index using a filter-aware search and prune on each inserted point.

2.3.2 AND predicates

AnalyticDB-V [5] supports arbitrary predicates, including AND queries, by retrieving all candidates that satisfy the predicate from a central database and then varying the search strategy based on the cardinality of the resulting set. AIRSHIP [48] supports AND queries by modifying the greedy search procedure of a graph index via careful selection of the starting point and a ”double queue“ approach to the greedy search that maintains separate queues for points that do and do not satisfy the filter predicate. CAPS [49] answers AND queries by building an IVF index over the dataset and then building a Huffman tree over the filters in each bucket. They search by first selecting buckets closer to the query point and then by using the Huffman trees to quickly identify points that satisfy the filter predicate. As demonstrated in their baseline for the NeurIPS23 Big ANN Benchmarks competition [19], the FAISS library [4, 50] can be used to support AND queries in an ad-hoc way, augmenting an IVF-based implementation with bitmaps to identify elements satisfying the filter predicates in each bucket.

Chapter 3: The IVF^2 Index

In this chapter, we present and evaluate the IVF^2 index, a novel approach to indexing vectors with boolean label metadata to serve filtered queries requiring one or two labels. Our contributions include:

- We characterize the typical distribution of real-world boolean labels
- We describe the IVF^2 index and motivate its design
- We ablate components of the index and discuss the impact of these ablations on its query performance, build time, and memory consumption
- We compare to a strong open-source baseline

3.1 Methods

3.1.1 Power-Law Distributed Labels

Labels which are 'power-law distributed' or more rigorously 'Zipfian', have frequencies distributed in such a way that for any cutoff x, and multiple α, if there are y labels with at least x occurrences, it's expected that there are $\frac{y}{\alpha}$ labels with at least αx occurrences. This pattern is common in real world data, being observed most famously in English-language word frequencies

[51], but also in the populations of cities [52], citation counts of scientific publications [53], and nonzero entries of sparse vector representations of text [54].

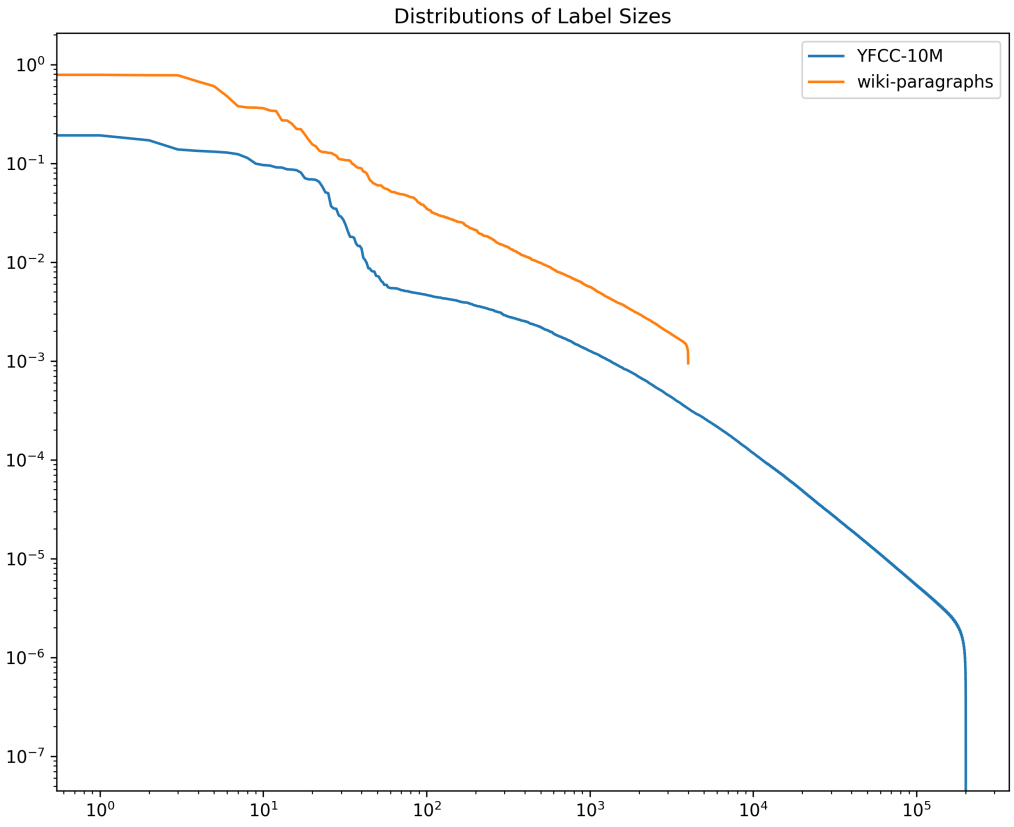


Figure 3.1: The cumulative density functions (CDFs) for two of our datasets with naturally derived labels, where a point (x, y) shows that a given dataset has x labels associated with at least y points.

Figure 3.1 shows that this pattern appears in both of our experimental datasets with natural labels, as a perfectly Zipfian distribution would appear linear on log-log axes. This is unsurprising, as both include labels derived from English text corpora: captions on photographs for YFCC-10M and the text of English language Wikipedia articles for wiki-paragraphs. YFCC-10M also includes labels corresponding to administrative regions where a photograph was taken, which also obey this pattern as a downstream effect of the distribution of city sizes.

This distribution of label frequencies motivated the design of the IVF^2 index. Because the

vast majority of labels in a dataset can be expected to be small and therefore efficient to search without indexing, we can reserve expensive indices for large labels.

Symbol	Description
n	Size of the full dataset
d	Dimensionality of the vectors in the dataset
f_i	A specific label in the dataset
$\|f_i\|$	Number of points associated with label f_i
C	Cutoff size between 'large' and 'small' labels
k	Number of clusters in IVF index
s	Target size of clusters in per-label IVF indices
$C_{\text{bitvector}}$	Cutoff for constructing a bit vector for large labels
$N_{\text{target points}}$	Target number of join candidates from a large label
C_{tiny}	Cutoff size for using bitvector join in AND queries

Table 3.1: Summary of notation for this chapter

3.1.2 Construction

We build an inverted file index over labels, where an index representing the points associated with label f_i is accessible in constant time from a pointer at index i in an array built for this lookup. For each f_i, the index built depends on the number of points $|f_i|$ associated with it. This allows us to reserve the construction of memory and build-time intensive datastructures for the largest labels.

This approach of independently indexing each label increases the minimal memory footprint by an amount proportional to the number of point-label connections present in the dataset, as the vectors being indexed are stored only once. We define a hyperparameter C representing the cutoff in size between 'large' and 'small' labels. In a setting where the memory footprint of the index is not a concern, C would ideally be the point at which more sophisticated indexing no longer offers a speedup over exhaustive search, but in practice C is the lowest value allowing the

resulting set of indices to fit in memory on the machine used for querying, which is significantly smaller for typical machines.

3.1.2.1 Small Labels

The power-law distribution of label frequencies entails that for reasonable values of C, the vast majority of labels f_i satisfy $|f_i| < C$ and are associated with relatively few points in the dataset. The points associated with small labels are indexed with a sorted array, which allows both minimal memory footprint and is convenient for the use of a linear time join between the points of two labels.

3.1.2.2 Large Labels

For f_i with $|f_i| > C$, we construct an index over the points in the large label f_i with three parts:

- An IVF index
- A Vamana search graph
- A dense vector of bits b_i of length n where $\forall j \in [n], b_i[j] = 1 \Leftrightarrow j \in f_i$

IVF Index We construct an IVF index over the points of f_i with k-means clustering [55] initialized by a hierarchical clustering splitting recursively on random hyperplanes. We store an array of associated indices and a centroid in $\mathbb{R}^d$ for each partition. A hyperparameter s is used to determine k for a given f_i, with $k = \lfloor \frac{|f_i|}{s} \rfloor$.

Search Graph In addition to the IVF index used for AND queries, we construct a Vamana search graph [2] over the points in f_i. We find that as $|f_i|$ increases, the max degree of a Vamana graph over the points associated with f_i should increase to preserve recall. To better fit graph construction to the size of the label in question, we define *weight classes*: disjoint size regimes above the cutoff with distinct build and search parameters optimized for datasets in that size regime.

Bit Vector We construct a bit vector encoding the membership of points in a large f_i for fast constant-time membership lookup. The memory footprint of this bit vector is linear in the size of the full dataset, and is the only component of the index which has superlinear memory footprint as the size of the full dataset n increases. As a result, to be more adaptable to large n, we define a separate cutoff $C_{\text{bitvector}}$, below which large labels do not construct a bit vector.

Materialized Join Often, two large labels will have an intersection with more than C points. In this case, we construct a search graph on the points in the intersection to accelerate queries constrained to the intersection of those labels.

3.1.3 Querying

Query behavior of the index is determined by the form of the query (single label vs. AND) and the size(s) of the labels referenced by the query.

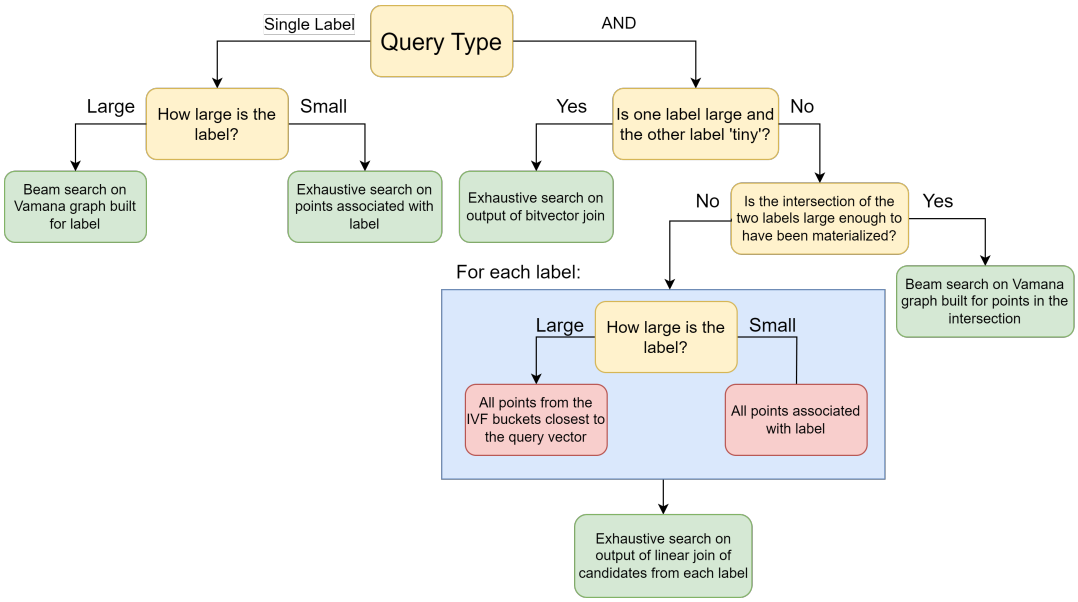


Figure 3.2: Flowchart illustrating the methods used to resolve queries.

3.1.3.1 Single-Label Filter

In the case where the filter on a query is only dependent on a single label, a query reduces trivially to a k-NN query on the index for that label.

Small Labels For f_i with $|f_i| < C$, we only store a sorted array of the indices of points associated with f_i. A k-NN query on such an array is simply an exhaustive search over the points referenced by this array of indices.

Large Labels For f_i with $|f_i| \geq C$, we have a Vamana search graph over the points associated with f_i built when constructing the index. When queried with beam search, this provides a SOTA k-NN index [3] over the points associated with f_i.

3.1.3.2 AND Filter

The AND filter case describes queries which restrict search to points which are in the intersection of f_i and f_j for some distinct i, j. With the exception of a special case where the bit vector of the larger filter is used to accelerate the computation of the intersection, candidates are found independently for each filter, and we exhaustively search the intersection of these sets.

Small Label Join Candidates Because we only store the indices of the points associated with small labels, the candidates returned for a small label in this case is the set of all points associated with the label, as we have no datastructure for efficiently restricting the candidates by proximity to the query, and want to restrict the candidates by performing the join before doing distance comparisons.

Large Label Join Candidates To collect join candidates from large labels, we leverage the assumption that the true nearest neighbors matching the predicate will be relatively close to the query vector among the points associated with the larger label. To find candidate points close to the query while minimizing distance comparisons, we compare the query vector to the centroids of the IVF index constructed over the label's points, and add indices of points in the nearest clusters to a sorted array until we have at least $N_{\text{target points}}$ candidates, a query parameter.

Bitvector Join In the case where an AND query is between a large label and an especially small label of size less than C_{tiny}, a query parameter, the intersection is computed exactly by checking the bitvector associated with the large label for each point associated with the small label. This has the advantages of avoiding the overhead of comparing to the centroids of the large label's

IVF index and providing exact results for the query.

Sorted Queries Batched queries with filters have the convenient property that an ordering which maximizes temporal locality can be computed far more easily than would be possible with vector-only queries. In order to leverage this, we lexicographically sort queries by the label(s) they constrain search to before processing them. In the parallel setting where we perform queries, this causes naive partitioning of queries into jobs in a work-stealing scheduler to group together queries using the same label-specific index in jobs which will begin in the work queue of the same core. In a serial setting or within a set of queries executed together on a given core, this has the advantage of allowing frequently accessed indices associated with a large filter to remain in L1 cache between queries.

3.2 Experimental Setup

We run construction and querying for experiments on a 2.10GHz 4 socket Intel Xeon machine with 96 cores and two way hyperthreading, 132 MiB L3 cache, and 1.47 TB of RAM. Querying is done with 8 cores to simulate the smaller machines which would typically be used to serve such an index, and construction with the full machine.

We performed hyperparameter tuning for our method with Optuna [56], using their Tree-structured Parzen Estimator (`TPESampler`) [57] and median pruner (`MedianPruner`) with default settings. Trials were run for recall cutoffs between 0.75 and 0.95 in steps of 0.05. The FAISS baseline used the configuration provided by the FAISS authors for the big-ann-benchmarks competition.

3.2.1 Datasets

Dataset	n	d	n_f	Max $\lvert f_i \rvert$	Mean $\lvert f_i \rvert$	Total Associations	80% Pareto	n_q	$n_\cap$
YFCC-10M	10M	192	200,386	3,386,745	540	108M	0.05	100,000	38,374
wiki-paragraphs	35M	768	4,000	32,583,414	285,687	1B	0.33	5,000	5,000
UQ-V	1M	256	7	500,254	428,571	3M	0.71	10,000	10,000
Audio	53K	192	7	26,778	22,880	160K	0.71	200	200
Crawl	2M	300	7	995,481	852,855	6M	0.71	10,000	10,000
SIFT1M	1M	128	7	500,254	428,571	3M	0.71	10,000	10,000
GIST1M	1M	960	7	500,254	428,571	3M	0.71	1,000	1,000
MSong	992K	420	7	496,374	425,259	3M	0.71	200	200
Enron	95K	1369	7	47,702	40,709	285K	0.71	200	200

Table 3.2: Dataset statistics. Total associations is the number of point-label associations in the dataset. 80% Pareto is the portion of labels responsible for 80% of all point-label associations in the dataset. n_q is the number of queries, and $n_\cap$ is the number of AND queries with a filter conditioning on two labels.

3.2.1.1 YFCC-10M

The YFCC-10M dataset consists of CLIP [15] embeddings of a 10 million image subset of the YFCC-100M dataset [58]. Labels are derived from metadata associated with the original images, including the year and country in which an image was taken, and keywords associated with the content of the image. Query vectors were generated by embedding held out elements of the dataset, and filters are 1-2 labels which must be present in points returned by the search.

3.2.1.2 wiki-paragraphs

The wiki-paragraphs dataset was constructed by embedding 35M paragraphs from English Wikipedia with the cohere.ai `multilingual-22-12` embedding model [59]. Labels were generated by taking the 4,000 most common words in the corpus and giving each element of the dataset a given word's label if the corresponding paragraph contains that word. Query vectors were generated by embedding paragraphs from Simple English Wikipedia, with filters generated by sampling at random two words from the top 4,000 and restricting search to paragraphs containing both of them.

3.2.1.3 NHQ Datasets

We also evaluate on a suite of smaller datasets for filtered ANNS compiled in [46]. The vectors for these datasets come from a variety of domains, and their labels are synthetic in a tabular form, where each point is associated with 3 labels, one per column. We modify the queries, which originally have filters conditioning on 3 labels, to use 2 of the 3 labels chosen at random.

3.3 Experiments

3.3.1 Comparisons against other algorithms

We compare against a FAISS baseline [60] tuned for our setting by the organizers of the big-ann-benchmarks competition [19]. We do not compare to NHQ [46], as the GitHub repo containing their code is no longer public, and a request for code to compare against was not answered.

Our performance relative to the FAISS baseline can be seen in Figure 3.3, which demonstrates that our full index and ablations thereof greatly outperform it.

3.3.2 Ablations

We run ablation studies to validate the design of the IVF^2 index. Each ablation removes a single component of the index and leaves the others intact.

- `no-material-join`: We remove the materialized joins indexing large intersections between labels. This is the form of the index which was used in the official submission to the `big-ann-benchmarks`

competition.

- `no-sorted-queries`: We do not sort the queries to improve locality between consecutive queries. This is the best case in the setting where queries are streamed instead of being presented in batches.

- `no-weight-classes`: We do not use the weight classes optimization fitting graph construction parameters to finer-grained size regimes, building all graphs with the parameters corresponding to the largest weight class.

- `no-bitvector` We do not use bitvectors to accelerate AND queries between large and small labels.

3.3.3 Query Performance

In the same manner as the big-ann-benchmarks competition, we evaluate QPS and recall in terms of throughput for a batch of queries, and target optimal QPS at 90% recall.

Figure 3.3 provides a representative example comparison of QPS and recall for the ablations on the YFCC-10M dataset.

`no-weight-classes` Relative to the full index, the ablation removing the lower weight classes has higher recall, which comes at the expense of increased memory footprint and build time. To show that the full index would be more competitive if using the same amount of memory, we have `high-mem`, a complete IVF^2 index with parameters adjusted to make the index size and build time comparable to (but, for the sake of strict correctness, slightly better than) the `no-weight-classes` ablation with 2.1% faster build and 2.5% less memory consumption.

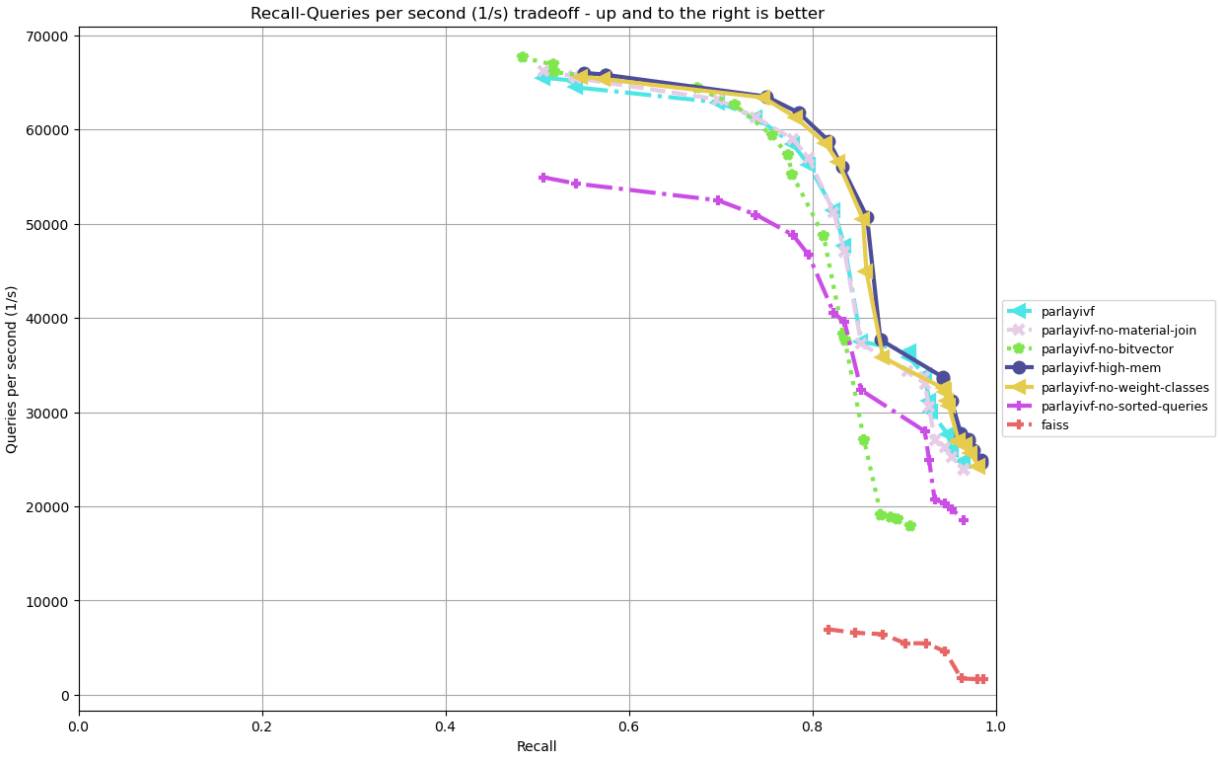


Figure 3.3: Relative query performance of ablations on YFCC-10M dataset. Faiss baseline included for comparison.

The uniformly slightly better performance of the `high-mem` configuration demonstrates that the improved performance of the `no-weight-classes` ablation over the baseline comes at a disproportionate cost in terms of index size and build time.

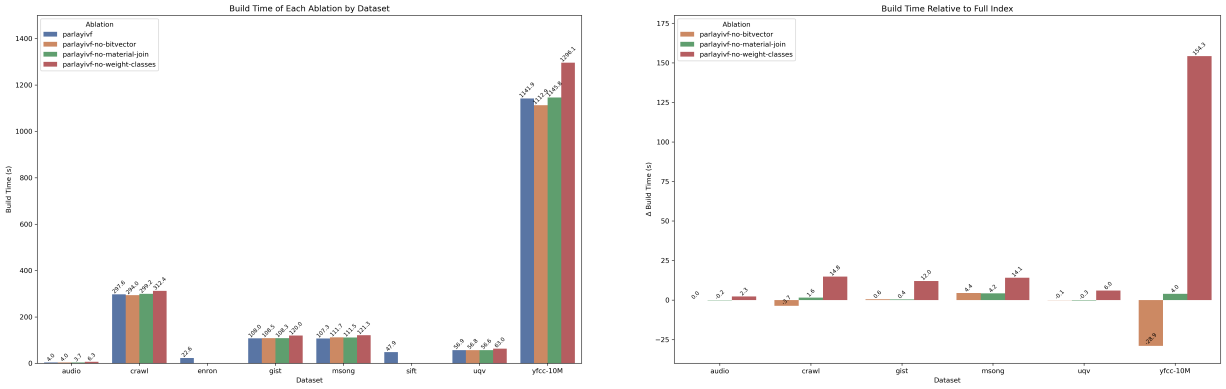
`no-sorted-queries` Not sorting the queries for improved locality uniformly lowers the throughput of the index, and the significant improvement in QPS validates that the net impact of the sort is positive.

`no-material-join` The removal of materialized joins from the index has an effect on QPS around 85% recall. Note in Table 3.2 that a minority of queries for the YFCC-10M dataset are AND queries, such that the effect of this ablation would be greater on another dataset with more large AND queries that use the materialized join graphs.

`no-bitvector` The removal of the bitvector has a significant impact on both QPS and recall in the high-recall regime, as the bitvector is both faster than the join it is used in place of, and retrieves perfectly the points relevant to an AND query, boosting recall over the inaccuracy introduced by the IVF index in AND queries between large and small labels.

3.3.4 Build Time

Reasonable build times are important for reducing the cost of constructing and updating indices.



(a) Plot of index construction times for different ablations.

(b) Plot of index construction times relative to the full index for different ablations.

Figure 3.4: Build times of ablations on different datasets.

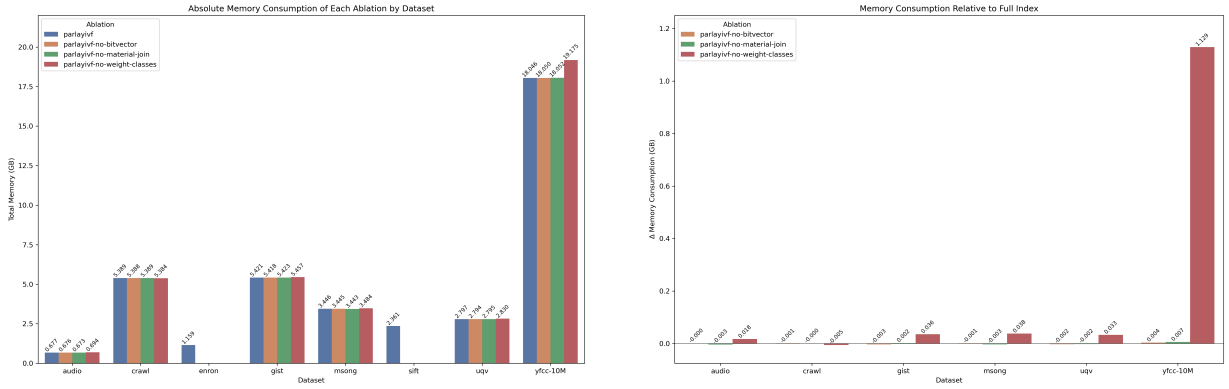
`no-weight-classes` The larger graphs constructed when removing the smaller weight classes are naturally slower to build, as they're higher quality than the sparser graphs which would otherwise be constructed over the smaller labels.

`no-material-join` Removing the materialized joins naturally improves build times as it avoids the construction of those additional graphs over the points in the materialized intersections.

`no-bitvector` The removal of the bitvector saves some construction time because this component of the index requires initializing a large chunk of memory for each label it applies to and setting a bit for each point associated with the label. Skipping this construction step therefore saves time.

3.3.5 Memory Footprint

Memory-resident vector indices such as IVF[2] are constrained in size by the memory of the machine they're served on. As a result, in real-world applications, efficient use of memory is important to enable serving indices on economical hardware.



(a) Plot of index memory footprints for different ablations.

(b) Index memory consumption of ablations relative to full index

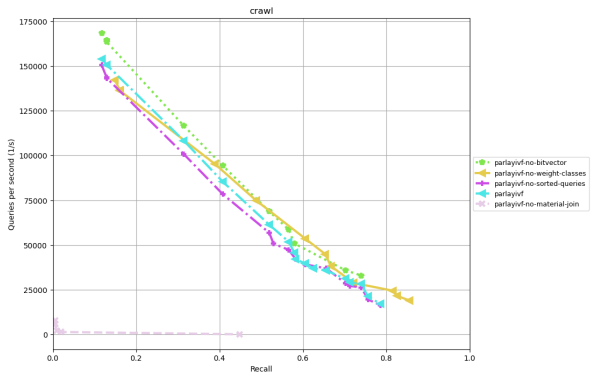
Figure 3.5: Memory footprints of ablations on different datsets.

`no-weight-classes` Note how `no-weight-classes` has a proportionally higher memory footprint than the native index, caused by the additional size of graphs which would have been constructed with lower degree bounds corresponding to lower weight classes.

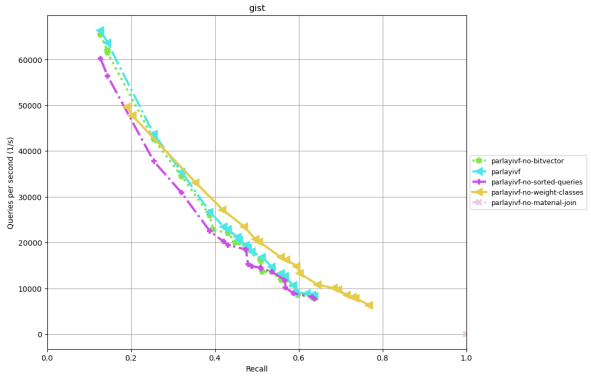
`no-material-join` Not materializing large intersections has a significant effect on the footprint of the index, as there are many such intersections and graphs are comparatively expensive to store. For this reason, it's advisable to disable this feature when queries are expected

to contain few ANDs on large labels or memory is highly constrained, but section 3.3.3 details the improved query performance it provides.

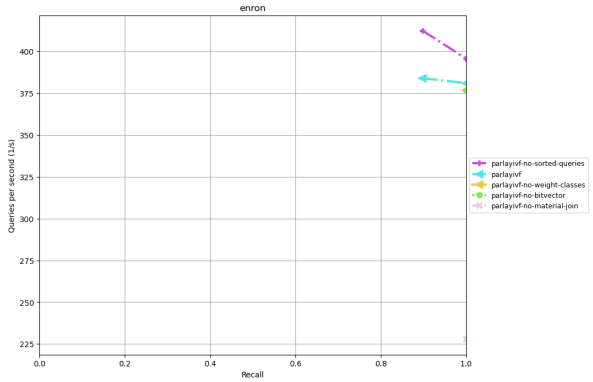
`no-bitvector` The removal of bitvectors for representing membership causes a marginal decrease in memory footprint, as we no longer store said bitvectors. The change in index size is small because the bitvectors are dwarfed in size by the graphs.



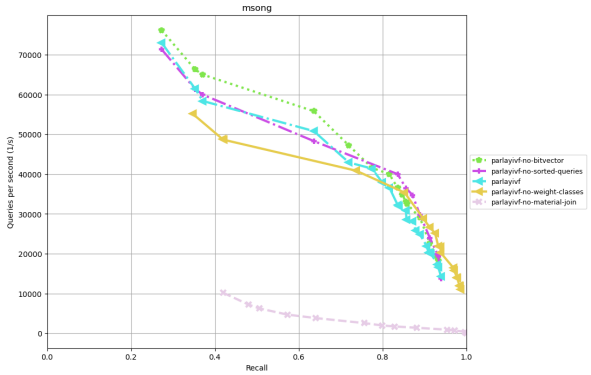
(a) Crawl dataset



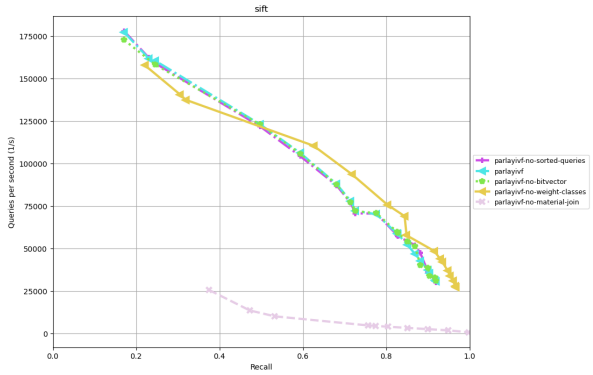
(b) GIST dataset



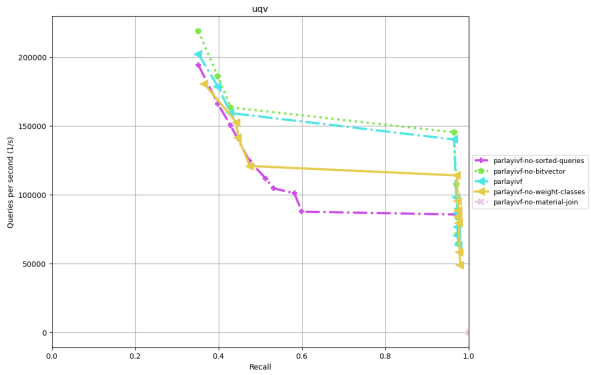
(c) Enron dataset



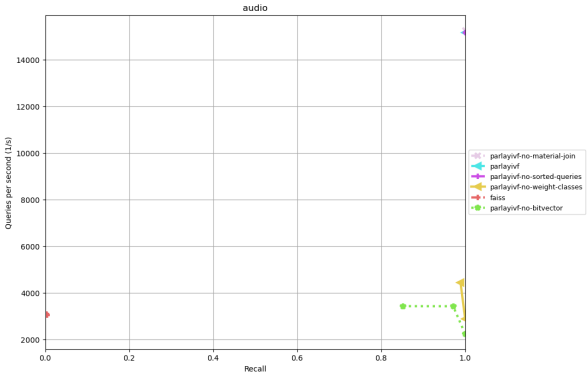
(d) Msong dataset



(e) SIFT dataset



(f) UQV dataset



(g) Audio dataset

Figure 3.6: QPS vs. Recall of ablations on various datasets

Chapter 4: Window Filtered ANNS

In this chapter, we are primarily interested in a difficult generalization of the ANNS problem that we term ***Window Search*** (see Definition 4.2.3). Window search is similar to traditional ANNS, except queries are accompanied by a "window filter", and the goal is to find the nearest neighbor to the query that has a label value within the window represented by the filter. This problem has a large number of immediate applications. For instance, in document and image search, each item may be accompanied by a timestamp, and the user may wish to filter to an arbitrary time range (e.g. they may wish to search for hiking pictures, but only from last summer, or forum posts about a bug, but only in the days after a new version was released). Another application is product catalogs, where a user may wish to filter search results by cost. Finally, as mentioned in Section 1.1, we note that a large class of emerging applications is large language model retrieval augmented generation, where by storing and retrieving information LLMs can reduce hallucinations and improve their reasoning [17]; window search may be critical when the LLM needs to recall something stored on a certain date or range of dates.

Although this problem has many motivating examples, there is a dearth of papers examining it in the literature. Some vector databases analyze window search-like problem instances as an additional feature of their system, but this analysis is typically secondary to their main approach and too slow for large-scale real-world systems; as far as we are aware, we are the first to propose,

analyze, and experiment with a non-trivial solution to the window search problem.

Our contributions include the following:

1. We formalize the c-approximate window search problem and propose and test the first non-trivial solution that uses the unique nature of numeric label based filters.
2. We design multiple new algorithms for window search, including a modular tree-based framework and a label-space partitioning approach.
3. We prove our tree-based framework solves window search and give runtime bounds for the general case and for a specific instantiation with DiskANN [2]. We also analyze optimal partitioning strategies for the label-space partitioning approach.
4. We benchmark our methods against strong baselines and vector databases, achieving up to a $75\times$ speedup on real world embeddings and adversarially constructed data.

4.1 Related Work

Segment Trees. Segment trees (and the closely related Fenwick tree [61]) are tree data structures built over an array that recursively sub-divide the array to obtain a balanced binary tree [62]. By storing appropriate augmented values at the internal nodes of this tree, these data structures can be used to support a variety of queries over arbitrary intervals in the array, e.g., computing the maximum value in any given query interval $[l, r]$. Segment and Fenwick trees can be generalized to higher fanout trees, i.e., B-ary segment or Fenwick trees that have a fanout of B and a height of $\lceil \log_B n \rceil$ [63]. Our work adapts these tree structures to the window search problem by designing a similar data structure that stores ANNS indices at internal tree nodes. [64] used the Fenwick tree for filtered search within a clustering context, but their work only considers a prefix interval

Symbol	Definition
D	Vectors to index
N	$\lvert D\rvert$, the cardinality of D
V	Metric space D is in, e.g., R^n
q	Query vector $q \in V$
(a, b)	Window filter; see Definition 4.2.2
c	Approximation factor for window search
$\texttt{dist}_V$	Distance function between points in D
d	Running time to evaluate $\texttt{dist}_V$
A	Arbitrary c-ANN algorithm, e.g., DiskANN
A_q	Query time of A
β	Split factor for a β-WST; see Algorithm 1
α	Pruning parameter for DiskANN
δ	Doubling dimension
R	Set of closed integer ranges in $[1, \ldots, N]$
$\mathsf{blowup}(R)$	Max ratio of superset $\in R$ over range length
$\mathsf{cost}(R)$	Sum of lengths of ranges in R

Table 4.1: Notation used in this chapter.

($[0, r]$), and they used kd-trees, which are designed for low-dimensional data.

4.2 Definitions

This section lays out definitions for the main problem that we study: window search. Notation for the next three sections can be found in Table 4.1.

Definition 4.2.1. [Labeled Dataset] Consider a metric space V with distance function $\texttt{dist}_V$. Given a label function $\ell : V \to \mathbb{R}$ and a set $D \subset V$, we define a ***labeled dataset*** to be the pair (D, ℓ).

Definition 4.2.2. [Window Filtered Dataset] Consider a labeled dataset (D, ℓ). We define a ***window filter*** to be an open interval (a, b) with $a, b \in \mathbb{R}$, and we define a ***window filtered dataset*** to be $D_{(a,b)} = \{x \in D \mid \ell(x) \in (a, b)\}$.

Definition 4.2.3. [Window Search] Given a labeled dataset (D, ℓ), we define a ***query*** to be a vector $q \in V$ and a window filter (a, b), and we define the ***window filtered nearest neighbor*** to

be $q^* = \arg\min_{x \in D_{(a,b)}} \texttt{dist}_V(x, q)$.

Definition 4.2.4. [Approximate Window Search] Finally, given a labeled dataset (D, ℓ), we define c***-approximate window search*** to be the task of constructing a data structure that takes in a query $q \in V$ with window filter (a, b) and returns a point $y \in D_{(a,b)}$ such that $\texttt{dist}_V(q, y) \leq c \cdot \texttt{dist}_V(q, q^*)$, or $\emptyset$ if $D_{(a,b)} = \emptyset$.

4.3 Window Search Algorithms

In this section, we describe algorithms to solve the window search problem. In Section 4.3.1, we examine two naive baselines for solving window search; in Section 4.3.2, we introduce a new data structure, the β-WST, and design an algorithm to query it; and in Section 4.3.3, we examine additional algorithms for solving window search.

We note that with a fixed window filter (a, b), a reasonable approach to solving window search is simply to index $D_{(a,b)}$ using an off-the-shelf ANNS algorithm and then query it with each q. Thus, we are interested in the more challenging problem where we receive queries with *arbitrary* window filters.

4.3.1 Naive Baselines

Prefiltering is a naive baseline that works by sorting all of the points by label ahead of time. Given a query x with window filter (a, b), we do binary search on the sorted labels to find the start and end of the range that meet the filter constraints, and then find the distance between x and every point in the range and return the closest point.

Postfiltering [65, 66] is a second baseline that works by first building an index on all of

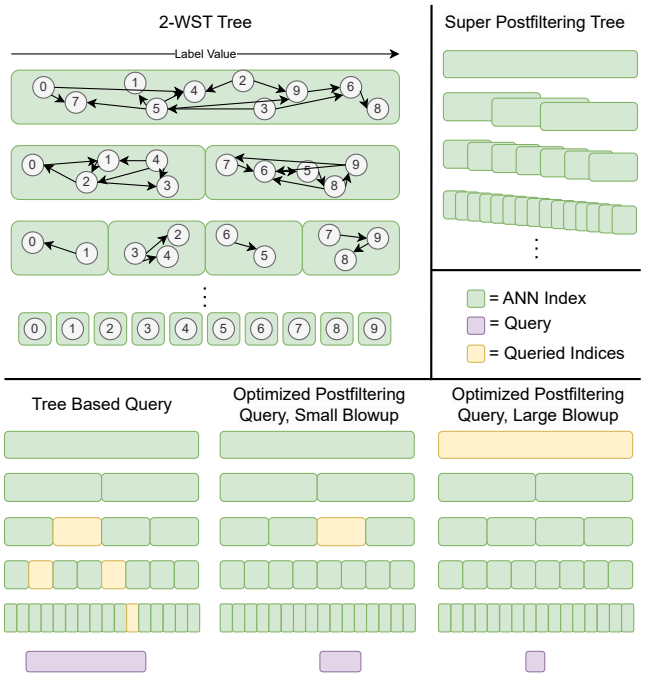


Figure 4.1: Top left is a 2-WST. Each index contains a recursive partition of the entire dataset D, indexed into a graph for fast ANN (see Algorithm 1). Top right shows the structure of an example label space partitioning method that ensures no optimized postfiltering query will have a large blowup (see Theorem 4.4.6). Bottom shows different query methods; from left to right: a tree-based query as in Algorithm 2, an optimized postfiltering query with a small blowup (see Definition 4.4.4), and an optimized postfiltering query with a large blowup (see Definition 4.4.4).

D using an ANN algorithm A. To do a window search, we query A for $k = 1$ point, and then continually double k until at least one point is returned that has a label within (a, b), and then we return the closest of these points. We additionally define a hyperparameter called final_multiply; if this is greater than 1, then we perform a final additional search with a final_multiply times larger value of k.

4.3.2 β-Window Search Tree

We now propose a data structure that we call a **β-Window Search Tree, or β-WST**. This data structure with $\beta = 2$ is illustrated in the top left of Figure 4.1 and the accompanying query method is illustrated in the bottom left. The overall idea to construct a β-WST is to split D into

Algorithm 1 BuildTree$(A, \beta, (D, \ell))$

1: **Input:** Dataset D with points $x_1, \ldots, x_{|D|}$ sorted by ℓ, branching factor β, ANNS algorithm A.
2: **Output:** β-Window Search Tree of D
3: **if** $|D| < \beta$ **then**
4: **return** $(\texttt{NULL}, \texttt{NULL}, D)$
5: $\texttt{index} \leftarrow A(D)$
6: $\texttt{sizes}[1, \ldots, \beta - 1] \leftarrow \lceil |D|/\beta \rceil$
7: $\texttt{sizes}[\beta] \leftarrow |D| - (\beta - 1) \cdot \lceil |D|/\beta \rceil$
8: $\texttt{children}[1, \ldots, \beta] \leftarrow \texttt{NULL}$
9: **for** $i \leftarrow 1$ **to** β **do in parallel**
10: $\texttt{start} \leftarrow (i - 1) \cdot \lceil |D|/\beta \rceil + 1$
11: $\texttt{end} \leftarrow \texttt{start} + \texttt{sizes}[i]$
12: $\texttt{children}[i] \leftarrow$ BuildTree$(A, \beta, (D_{(x_{\texttt{start}}, x_{\texttt{end}-1})}, \ell))$
13: **return** $(\texttt{index}, (\texttt{children}, \texttt{sizes}), D)$

β subsets, one corresponding to each child node, construct an instance of A at each node, and recurse on each child. We continue this process until the subset size is less than β, in which case we just store the points directly.

More formally, let $x_1, \ldots, x_N$ be the points of D sorted by ℓ, such that $\ell(x_1) \leq \ell(x_2) \ldots \leq \ell(x_n)$. With a slight abuse of notation, we define the $\arg\min$ of an empty set to be the empty set. A β-WST works as follows:

Index Construction. A β-WST T can be constructed as shown in Algorithm 1. In the base case, if the dataset D is small, we do not construct any tree node (Lines 3–4). Otherwise, we construct an ANN index of D (Line 5). On lines 6–7 we define the sizes for splitting D into β partitions, all of which are equally sized except the last, which may be smaller. On line 8 we initialize the children nodes. We then loop through every partition in parallel (Lines 9–12) and recursively call `BuildTree` on the set of points (sorted by label) corresponding to the start and end of the partition. Finally, on line 13 we return the constructed tree, which is a tuple consisting of an ANN index built on D, the result of `BuildTree` called on each child with the size of each child, and the point set D.

Querying the Index. We query a β-WST using Algorithm 2. The input is a β-WST T as

built by Algorithm 1, a query point q , and a window filter (a, b). At a high level, Algorithm 2 recurses through the tree constructed by Algorithm 1 and queries instances of A that union together to equal the entire filtered dataset. If the dataset is small and the index is a leaf node `NULL`, we do a brute force search over D (Lines 3–4). If the window filter (a, b) covers all points, we query the ANN `index` and return the result (Lines 5–6). Otherwise, D has some points that do not have a label in (a, b), so we loop over each child (Line 8) and recurse into it if some of the points in the child meet the query's window filter constraint. Finally, for each one of these children that we recurse into we add the returned points to a candidate list $\mathcal{L}_{\text{cand}}$, and return the closest point from $\mathcal{L}_{\text{cand}}$ on line 13.

4.3.3 Additional Query Methods

In addition to Algorithm 2 above, we examine a number of additional query methods for window search that come with various trade offs.

- OptimizedPostfiltering uses the index built by Algorithm 1 but uses a novel query algorithm. Given a query x with window filter (a, b), OptimizedPostfiltering finds the smallest subset S of D corresponding to an index we built $I = A(S)$ where $D_{(a,b)} \subset S$, and then queries that index using the same procedure as described in Postfiltering. A "small blowup" query is one in which the smallest subset S is not that much larger than $D_{(a,b)}$, whereas a "large blowup" query is one where S is much larger than $D_{(a,b)}$. These small and large blowup queries are shown on the bottom right of Figure 4.1. Blowup factor is defined formally in Definition 4.4.4.
- ThreeSplit also uses the index built by Algorithm 1 and a novel querying algorithm. Similar to Algorithm 2, a query initially finds the highest level where any partition at all is entirely

Algorithm 2 Query$(T, q, (a, b))$

Input: β-WST $T = (\texttt{index}, (\texttt{children}, \texttt{sizes}), D)$, with points $x_1, \ldots, x_{|D|} \in D$ sorted by ℓ, query q, window filter (a, b).
Output: Approximate window-filtered nearest neighbor y, or $\emptyset$ if no points in D meet window filter constraint.
if $\texttt{index} = \texttt{NULL}$ **then**
    return $\arg\min_{y \in D_{(a,b)}} \texttt{dist}_V(q, y)$
if $(\ell(x_1), \ell(x_{|D|})) \subset (a, b)$ **then**
    return $\texttt{index}(q)$
$\texttt{start} \leftarrow 1$, $\mathcal{L}_{\text{cand}} \leftarrow \emptyset$
for $i \leftarrow 1$ **to** β **do**
    $\texttt{end} \leftarrow \texttt{start} + \texttt{sizes}[i]$
    if $\ell(x_{\texttt{start}}), \ell(x_{\texttt{end}-1}) \cap (a, b) \neq \emptyset$ **then**
        $\mathcal{L}_{\text{cand}} \leftarrow \mathcal{L}_{\text{cand}} \cup \text{Query}(\texttt{children}[i], q, (a, b))$
    $\texttt{start} \leftarrow \texttt{end}$
return $\arg\min_{y \in \mathcal{L}_{\text{cand}}} \texttt{dist}_V(q, y)$

contained in the window filter, and then does a query on every one of these partitions. Instead of recursing further down the tree, however, ThreeSplit then does an OptimizedPostfiltering call on each of the remaining label subranges on each side of the middle "covered" label portion. Because we fill in the middle first, we are guaranteed that there can be no "large blowup" case like in OptimizedPostfiltering.

- SuperPostfiltering is the same as OptimizedPostfiltering except that it operates on an arbitrary set of indexed subsets of D and not necessarily the ones constructed by Algorithm 1. One example of such a data structure is analyzed in Theorem 4.4.6 and visualized in the top right of Figure 4.1.

4.4 Theoretical Analysis

4.4.1 Analysis of Querying a β-WST (Algorithm 2)

In our analysis in this section, we assume without loss of generality that N is a power of β. Removing this assumption would lead to more floor and ceiling operators in Theorem 4.4.1 and

leave our other results unchanged. In the interest of space, we defer proofs to the Appendix.

Theorem 4.4.1. *If A can build an index that answers c-ANN queries on an arbitrary size m subset of D with query time $O(A_q(D, m))$, and a distance computation in V takes d work, then Algorithm 2 solves the c-approximate window search problem with running time*

$$O\left(\beta \log_\beta(N) d + \beta \sum_{j=0}^{\log_\beta N} A_q(D, N \cdot \beta^{-j})\right).$$

Proof. There are two components to proving Theorem 4.4.1. First, we must show that Algorithm 2 solves the c-approximate window search problem, and then we must show that it solves it in the given running time.

Algorithm 2 solves the c-approximate window search problem

First, we establish correctness and completeness of the evaluated points, i.e., that every point returned has a valid label, and that all points that meet the valid label are "evaluated" on either Line 4 or Line 6.

For correctness, note that if a point is returned on Line 4, by Definition 4.2.2 it has a label value in (a, b). Similarly, since a point returned on Line 6 is in D and by the if statement we know that all points in D have a label in (a, b), a point returned on Line 6 has a label value in (a, b). Any point returned by Line 13 is an $\arg\min$ over points returned in one of these two cases, so we are guaranteed that the overall point y returned has $\ell(y) \in (a, b)$.

For completeness, first consider some call to Algorithm 2 with $T = (index, (children, sizes), S)$. By assumption, N is a power of β, so we will proceed inductively over $|S|$ equal to powers of β. Let us first consider any S such that $|S| = 1$. If $x \in S_{(a,b)}$, then it will be evaluated on Line 4.

Now we assume that for $|S| = \beta^n$, if $x \in S$, then a call to $Query$ with the tree corresponding to S will evaluate x. For all sets S of size β^{n+1} that contain some x, if Line 4 or Line 6 is executed, then we evaluate x. Otherwise, by construction (Line 12 in Algorithm 1) the children subsets S_i completely partition S, so x is in some S_i with $|S_i| = \beta^n$, and so by our inductive hypothesis x will be evaluated when we call query on $children[i]$.

We now show that a c-approximate window-filtered nearest neighbor is returned for some (possible recursive) call to $Query$. Because of our correctness guarantee, at some point q^* will be evaluated on Line 4 or Line 6. If q^* is evaluated on Line 4, then because q^* is the closest point to q in all of $D_{(a,b)}$, it will also be the closest point to q in $S_{(a,b)} \subset D_{(a,b)}$, so it will get returned by the $\arg\min$ (and q^* is trivially a c-approximate window filtered nearest neighbor). If q^* is evaluated on Line 6, then by the guarantee of the c-ANN algorithm A, some point y will be returned that is a c-ANN to q on $S_{(a,b)}$. Because q^* is also in $S_{(a,b)}$, this implies that $\texttt{dist}_V(q, y) \leq c \cdot \texttt{dist}_V(q, q^*)$, so y is also a c-approximate window filtered nearest neighbor.

Finally, we show that if any instance of a call to $Query$ finds a c-approximate window filtered nearest neighbor, then the overall algorithm will return a c-approximate window filtered nearest neighbor. Consider the case that a valid c-approximate window filtered nearest neighbor y is returned by Line 4 or Line 6. If this is not a top-level call to $Query$, then $Query$ was called on Line 11, so the point y' that gets returned will also be evaluated in the $\arg\min$ on Line 13, and a point y' will be returned from Line 13 that is in $D_{(a,b)}$ (by our correctness result) and has $d(q, y') \leq d(q, y) \leq c \cdot d(q, q^*)$. Thus by transitivity, y' is also a c-approximate window filtered nearest neighbor for q, and inductively the point y'' that gets returned by the original top-level $Query$ call will be a c-approximate window filtered nearest neighbor.

Algorithm 2 running time

We will examine each level of the tree built by Algorithm 1 as Algorithm 2 traverses it, i.e., the nodes with $|S| = N$, $|S| = N/\beta$, $|S| = N/\beta^2, \ldots, |S| = \beta, |S| = 1$ (the nodes with $|S| = 1$ are just the $index = NULL$ case).

At a high level, this analysis is similar to B-ary segment or Fenwick trees [63], which have $O(\beta \log_\beta N)$ query time and query at most $O(\beta)$ indices per level. The overall idea for our analysis is to show that Algorithm 2 will run an ANN search on at most $2\beta - 2$ indices per level.

First, we note that our algorithm has a "one time evaluation guarantee": if we execute an ANN search (Line 6) or an exact search (Line 4) on some subset S, then we did not execute an ANN search or exact search on any parent of S (since then we never would have reached it recursively), so every point in S (and therefore every point in $D_{(a,b)}$) is evaluated just once.

Now consider the largest j such that there exists some set S in the tree of size β^j such that $S \subset D_{(a,b)}$. In other words, S is the largest set that we built an index for and that entirely consists of points within the filtered dataset corresponding to the query. There may be multiple sets of size β^j within $D_{(a,b)}$.

Let all sets of size β^j be ordered such that each set's labels are strictly less than the next set, and let these sets be indexed by $\{S_i\}$. Let S_{first} be the first set in this ordering that is a subset of $D_{(a,b)}$ and S_{last} be the last set in this ordering that is a subset of $D_{(a,b)}$.

By completeness, every point in $S_{first}, \ldots, S_{last}$ is evaluated at some level, so the recursive traversal must go through $S_{first}, \ldots, S_{last}$, and since each of these sets is a subset of $D_{(a,b)}$, we will run the ANN search on Line 6 on each of $S_{first}, \ldots, S_{last}$.

By construction, every β sets in $\{S_i\}$ are partitions of a set from level $j + 1$ (e.g., sets

$\{S_1, \ldots, S_\beta\}$, $\{S_{\beta+1}, \ldots, S_{2\beta}\}, \ldots$). Thus, any subsequence of $\{S_i\}$ that is of length $\geq 2\beta - 1$ must have at least one complete partition of a set from level $j+1$. Since all of $S_{first}, \ldots, S_{last}$ are subsets of $D_{(a,b)}$, by the one time evaluation guarantee, we know that their parents cannot be subsets of $D_{(a,b)}$ (since then in the recursive traversal, we would have run an ANN search on their parent). Thus, $S_{first}, \ldots, S_{last}$ cannot contain a complete partition of a set from level $j+1$, so the list $S_{first}, \ldots, S_{last}$ contains fewer than $2\beta - 1$ sets, or equivalently $last - first + 1 \leq 2\beta - 2$.

We also know that at level j, there are at most two more sets, $S_{first-1}$ and S_{last+1}, that have a non-empty intersection with $D_{(a,b)}$ (these are the sets that potentially contain points with labels just larger and just smaller than a and just larger and just smaller than b).

This leads to our inductive hypothesis, which has three claims:

1. For all levels with $j' \leq j$ (i.e., with $|S| = \beta^{j'}$), there can be at most two sets that have a non-empty intersection with $D_{(a,b)}$ that are not fully evaluated (i.e., that we recurse into).

2. No set that we recurse into or evaluate on level $j' \leq j$ is a superset of $D_{(a,b)}$.

3. We will run the ANN search on Line 6 at most $2\beta - 2$ times on each level.

We have just shown the base case for $j' = j$. Now consider some $0 \leq j' < j$. By part 1 of the inductive hypothesis, we recurse into 2 or fewer sets on level $j'+1$ that have a nonempty intersection with $D_{(a,b)}$. For each of these sets, at most $\beta - 1$ of its children will be a subset of $D_{(a,b)}$ (if all β of its children were a subset of $D_{(a,b)}$, then the set itself would have been a subset of $D_{(a,b)}$ and would have been fully evaluated), and thus at most $2(\beta - 1) = 2\beta - 2$ ANN searches are run on level j'. This proves part 3 of our inductive claim. Furthermore, since each of the at most two sets that we are recursing into are not a superset of $D_{(a,b)}$ by inductive claim 2, there can only be at most one side of each of their label ranges that expand beyond (a, b). Thus,

when we partition each of these sets, only one child of each of the sets can have a label range that overlaps (a, b); the rest will either have labels entirely in (a, b) or entirely outside of (a, b). Thus there will be at most two sets that have a nonempty intersection with $D_{(a,b)}$ that are not fully evaluated, proving inductive claim 1. Finally, because each set that we recurse into on level j' is not a superset of $D_{(a,b)}$, all of the children we recurse into that are subsets of these sets are also not supersets of $D_{(a,b)}$, proving inductive claim 2.

We do work in Algorithm 2 on Line 6, the loop on Line 8, and Line 13 (we do not do work on Line 4 because the $\arg\min$ is just over one point; the $\arg\min$ is necessary for the more general case where N is a not a power of j and the leaf nodes may have more than one point). As a direct result from the first part of our inductive claim, we have that we will only make it to the loop on Line 8 and the $\arg\min$ on Line 13 twice for each of the $\log_\beta(N)$ levels. The time complexity for the loop is $O(\beta)$, and the time complexity for Line 13 is $O(\beta d)$ because the maximum size for the candidate list is β, and for each candidate in the list we spend $O(d)$ doing a distance computation. Also, from the third part of our inductive claim, we have that we call ANN search on an index of size $m = \beta^j$ at most $2\beta - 2$ times for all $j \in 1, \ldots, \log_\beta(N)$. Finally, again from the third part our inductive claim, we evaluate at most $2\beta - 2$ sets of size 1, so we evaluate Line 4 a maximum of $2\beta - 2$ times. This gives the following total runtime for Algorithm 2:

$$O\left(\log_\beta(N) * (\beta + \beta d) + (2\beta - 2) * \sum_{j=0}^{\log_\beta(N)} A_q(D, N * \beta^{-j})\right) =$$

$$O\left(\beta \log_\beta(N) d + \beta * \sum_{j=0}^{\log_\beta(N)} A_q(D, N * \beta^{-j})\right)$$

□

Many theoretical ANN results in the literature have a runtime of $O(N^\rho)$ for constant ρ, e.g., LSH [67] and k-nearest neighbor graphs [68]. Other results have a runtime that is paramterized only with constants describing the data distribution, and have no reliance on N. By applying Theorem 4.4.1, we have the following results for these common function classes:

Lemma 4.4.2. *If A is a c-ANN algorithm with $A_q(D, m) = O(Cdm^\rho)$ for $\rho \in (0, 1)$ for some constant C depending on D, the running time of Algorithm 2 is*

$$O\left(\frac{C\beta dN^\rho}{1-\beta^{-\rho}}\right).$$

If A is a c-ANN algorithm with $O(A_q(D, m)) = O(A_q(D))$, then the running time of Algorithm 2 is

$$O\left(\beta \log_\beta(N)[d + A_q(D)]\right).$$

Proof. For the first result, we have via substitution into Theorem 4.4.1:

$$\begin{aligned}
&O\left(\beta \log_\beta(N)d + \beta \sum_{j=0}^{\log_\beta N} A_q(D, N \cdot \beta^{-j})\right) \\
&= O\left(\beta \log_\beta(N)d + \beta \sum_{j=0}^{\log_\beta N} CdN^\rho \cdot \beta^{-j\rho}\right) \\
&= O\left(\beta \log_\beta(N)d + C\beta dN^\rho \sum_{j=0}^{\log_\beta N} \beta^{-j\rho})\right) \\
&= O\left(\beta \log_\beta(N)d + C\beta dN^\rho \frac{1}{1-\beta^{-\rho}}\right) \\
&= O\left(\frac{C\beta dN^\rho}{1-\beta^{-\rho}}\right)
\end{aligned}$$

and for the second result, we have via substitution into Theorem 4.4.1:

$$
\begin{aligned}
&O\left(\beta \log_\beta(N) d + \beta \sum_{j=0}^{\log_\beta N} A_q(D, N \cdot \beta^{-j})\right) \\
&= O\left(\beta \log_\beta(N) d + \beta \sum_{j=0}^{\log_\beta N} A_q(D)\right) \\
&= O\left(\beta \log_\beta(N) d + \beta \log_\beta(N) A_q(D)\right) \\
&= O\left(\beta \log_\beta(N)(d + A_q(D))\right)
\end{aligned}
$$

□

Finally, we can apply Lemma 4.4.2 to a recent c-ANN result for DiskANN graph-based search [39]. This is particularly relevant because we use DiskANN graphs for our experiments. Letting Δ be the *aspect ratio* of D, i.e., the ratio between the maximum and minimum distances between any two pairs of points, we have the following result:

Lemma 4.4.3. *Algorithm 2 instantiated with a "slow preprocessed" α-DiskANN graph solves the c-approximate window search problem in any metric space on a dataset with doubling dimension δ and aspect ratio Δ in running time*

$$
O\left(\beta \log_\beta(N)\left[d + \log_\alpha\left(\frac{\Delta}{(\alpha-1)(c-\frac{\alpha+1}{\alpha-1})}\right)(4\alpha)^\delta \log \Delta\right]\right).
$$

Proof. [39] considers greedy search on an α-DiskANN graph built on a dataset D with "slow preprocessing." They show that the search procedure is guaranteed to return a $(\frac{\alpha+1}{\alpha-1}+\epsilon)$-ANN in $O(\log_\alpha(\frac{\Delta}{(\alpha-1)\epsilon}))$ steps, each of which take $O((4\alpha)^\delta \log \Delta)$. See Section 2.2.2.3 for an explanation of α and a complete overview of DiskANN.

We can multiply together these two bounds on the running times to get an upper bound on the running time of the entire procedure:

$$O\left(\log_\alpha\left(\frac{\Delta}{(\alpha-1)\,\epsilon}\right)(4\alpha)^\delta \log\Delta\right) = O\left(\log_\alpha\left(\frac{\Delta}{(\alpha-1)\,(c-\frac{\alpha+1}{\alpha-1})}\right)(4\alpha)^\delta \log\Delta\right)$$

Note the equality is due to the fact that $c = \frac{\alpha+1}{\alpha-1} + \epsilon$.

Now consider some $S \subset D$. We claim that the doubling dimension δ and the aspect ratio Δ for S are no greater than for D. [39] describe doubling dimension as the the minimum value δ such that for any ball of radius r centered at some point x in D, 2^δ balls of radius $r/2$ can be arranged to cover all points in $D \cap B(x, r)$ (many previous works use the same or an extremely similar definition of doubling dimension (see, e.g., [69, 70]). Because S is a subset of D, any covering of $D \cap B(x, r)$ is also a covering of $S \cap B(x, r)$ for all x in D (and also trivially therefore all x in $S \subset D$), so we know that the doubling dimension of S is at most the doubling dimension of D. Similarly, [39] use the aspect ratio of D, which is

$$\frac{\max_{x_1,x_2\in D, x_1\neq x_2} \texttt{dist}_V(x_1, x_2)}{\min_{x_1,x_2\in D, x_1\neq x_2} \texttt{dist}_V(x_1, x_2)}.$$

For any subset S of D, let the two points corresponding to the smallest distance be x_{small} and y_{small} and the two points corresponding to the largest distance be x_{large} and y_{large}. Since $S \subset D$, x_{small}, y_{small} are also in D, so the smallest distance in D is less than or equal to $\texttt{dist}_V(x_{small}, y_{small})$, and similarly x_{large}, y_{large} are also in D, so the largest distance in D is greater than or equal to $\texttt{dist}_V(x_{large}, y_{large})$. Thus compared to Δ, the numerator of Δ_S is no larger and the denominator is no smaller, and so $\Delta_S \leq \Delta$. In other words, the aspect ratio of any

subset S is less than the aspect ratio Δ of D.

Because our running time result above is monotonic in Δ and δ, and the other parameters α and c are constant, we have that c-ANN search on any subset $S \subset D$ is upper bounded by the running time on the entire dataset D. Thus, since $O(A_q(D, m) = A_q(D))$, we can now plug in to Lemma 4.4.2, giving us our final result. □

4.4.2 Analysis of Super Postfiltering

SuperPostfiltering operates on an arbitrary collection of window filtered subsets $D_{(a_i,b_i)}$. A natural question is how to quantify the quality of a particular choice of subsets to index, which motivates the following definition:

Definition 4.4.4. [Blowup Factor, Cost] Given a set of ranges R, with $R_i = [a_i, b_i]$ for $a_i, b_i \in \{1, \ldots, N\}$ and $a_i < b_i$, we can define the *blowup factor* of a new query range $[a, b]$ with $a, b \in \{1, \ldots N\}$ as follows:

$$\mathsf{blowup}(R, [a, b]) = \min_{\substack{[a_i,b_i]\in R_i \\ [a_i,b_i]\supset[a,b]}} \left(\frac{b_i - a_i}{b - a}\right).$$

Intuitively, the blowup for $[a, b]$ is the ratio between the size of $[a, b]$ and the smallest range in R that contains it. Note that if no range $[a_i, b_i]$ contains $[a, b]$, we say that the blowup is equal to ∞. We can further define the worst case blowup for a *set of ranges* (like R) by taking the maximum blowup over all possible query ranges $[a, b]$:

$$\mathsf{blowup}(R) = \max_{\substack{a,b\in\{1,\ldots,N\} \\ a<b}} \mathsf{blowup}(R, [a, b]).$$

We additionally define the *cost* of a set of ranges R as $\mathsf{cost}(R) = \sum_i (b_i - a_i)$.

Intuitively, if we build an ANN index for the points corresponding to each range, then the worst-case blowup limits how expensive a query can be, while the cost approximates the memory required (since most practical ANN indices, e.g., LSH [67] and DiskANN [2], have memory that is approximately linear in the number of points they index). Note that here, a range of, e.g., $[17, 35]$ corresponds to an ANN index built on the 17'th point through the 35'th point in D, assuming the points are sorted by label value.

As a quick warmup, we can achieve a worst-case blowup of N and cost of N by choosing $R = \{[1, N]\}$, and we can get a worst-case blowup of 1 and cost of $O(N^3)$ by choosing $R = \{[i, j] \mid i, j \in \{1, \ldots, N\}, i < j\}$. We are interested in choices for R that lead to better tradeoffs.

We can construct an R consisting of the ranges corresponding to each subset indexed in a β-WST, so we can analyze it using Definition 4.4.4.

Lemma 4.4.5. *The ranges corresponding to a β-WST have worst case blowup factor $B = N/2 = O(N)$ and cost $\leq N\lceil \log_\beta(N) \rceil = O(N \log_\beta(N))$.*

Proof. To see that the worst case blowup factor is $O(N)$, consider the first split of D into children S_i for $i = 1, \ldots \beta$. These S_i correspond to label ranges $\{[a_i, b_i]\}$, where $a_0 = 0$, $b_{last} = N$, and each $a_i = b_{i-1} + 1$. Consider the range (b_1, a_2). This range is not a subset of any S_i. Furthermore, because all smaller ranges further down the tree are strict subsets of some S_i, this range is also not a subset of any smaller range. Thus the smallest range that S_i is a subset of is the top level range, so a β-WST has a worst case blowup of $\frac{N}{2} = O(N)$.

For the cost, we note that the label ranges of each level of the tree (except possibly the last, since it might be only partially full) are a partition of $\{1, \ldots, N\}$, so the sum of $b_i - a_i$ is equal to N for all levels but the last. There are $\lfloor \log_\beta(N) \rfloor$ levels, and one possibly non-full level at the

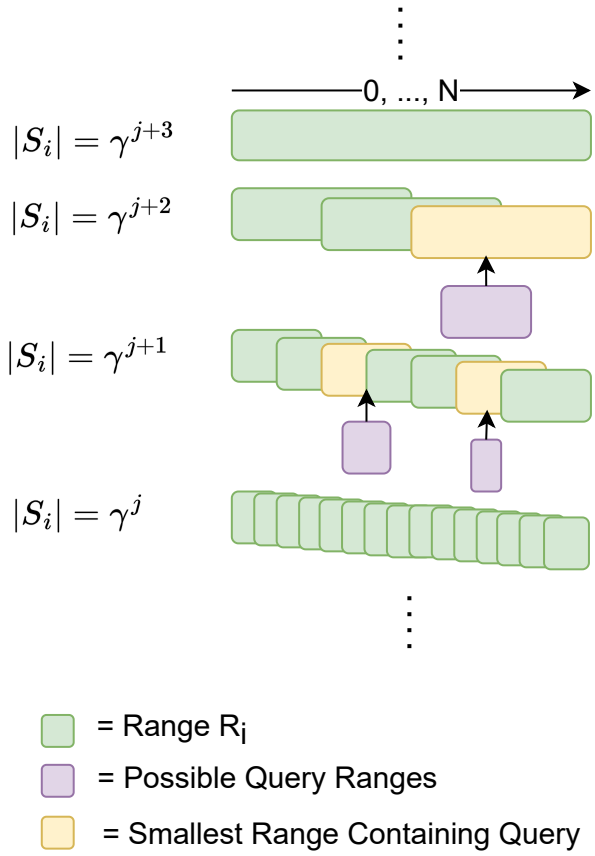


Figure 4.2: Illustration of structure of ranges for Theorem 4.4.6.

bottom of the tree which is smaller than or equal to a full partition of $\{1, \ldots, N\}$ and so has a cost less than or equal to N. Thus the total cost is bounded by $\lceil \log_\beta(N) \rceil \cdot N$. □

Finally, we prove that we can do better than a β-WST.

Theorem 4.4.6. *For any N and for any $\gamma > 1$, there exists an R with worst case blowup factor 2γ that has cost at most $N\left(2\log_\gamma(N) + 1\right)$.*

Proof. At a high level, our approach is to devise a strategy that can ensure all sets of size m have a blowup factor of 2. We will then repeat this strategy for $m = \gamma^j$ for all possible powers of j, which will ensure that all possible ranges have a small worst case blowup factor. For a diagram of this structure see Figure 4.2.

First, consider the problem of choosing ranges R_i such that every range of size m is a subset of some R_i with blowup factor equal to 2. One approach is to choose ranges of

$$cover(m) = \{[jm+1, (j+2)m] \mid j \in \mathbb{Z}_{\geq 0}, (j+2)m \leq N\} \cup [N-2m+1, N]$$

The ends of the ranges start at $2m$ and go until N by multiples of m, for a total of $\lfloor \frac{N}{m} \rfloor - 1$ ranges. These, plus the additional range $[N-2m+1, N]$, lead to a total of $\lfloor \frac{N}{m} \rfloor$ ranges created using this strategy. Each range has width $2m$, so the arrangement has cost

$$\left\lfloor \frac{N}{m} \right\rfloor \cdot 2m \leq 2N.$$

We now show that these ranges R_i do indeed cover all ranges of size m with blowup factor equal to 2. Consider some range of length m starting at a. If a is within the first $m+1$ integers in a range, then it is entirely within the range. Therefore, we are interested in the union of the first $m+1$ integers in all of the ranges, or

$$\bigcup_{(j+2)m \leq N} \{[jm+1, (j+1)m+1]\} \cup [N-2m+1, N-m+1]$$

$$\supset [1, N-2m] \cup [N-2m+1, N-m+1]$$

$$= [1, N-m+1]$$

This is all possible starting points for a range of length m, so R_i does indeed cover all ranges of size m. Furthermore, each range is of size $2m$, so the blowup factor for these ranges of size m is 2.

Table 4.2: Summary of datasets used in our experiments.

Dataset	Description	Labels	Num. dimensions	Dataset size	Num. queries
SIFT	Image feature vectors	Uniform random	128	$1M$	$10K$
GloVe	Word embeddings	Uniform random	100	$1.18M$	$10K$
Deep	GoogLeNet embeddings	Uniform random	96	$9.9M$	$10K$
Redcaps	CLIP image embeddings	Timestamps	512	$11.6M$	800
Adverse	Mixture of Gaussians	Noisy cluster ID	100	$1M$	$9.9K$

Now consider $R = \{cover(\gamma^j) \mid j \in \mathbb{Z}_{\geq 0}, \gamma^j < N\} \cup (0, N)$. We have that

$$\begin{aligned} cost &= \lfloor \log_\gamma(N) \rfloor \cdot 2N + N \\ &\leq 2N \log_\gamma(N) + N \\ &= N\left(2\log_\gamma(N) + 1\right) \end{aligned}$$

Furthermore, we now show that R has worst case blowup factor 2γ. Consider some range r_q of size m. Consider the minimum j such that γ^j is greater than m. Let us first consider the case when γ^j is less than N. Consider some range r of size γ^j that contains r_q. By the definition of $cover(\gamma^j)$, there is some range of size $2\gamma^j$ that contains r and that therefore contains r_q. Furthermore, since this j is the minimum j such that $\gamma^j > m$, we have that $m > \gamma^{j-1}$. Thus the maximum blowup factor for r_q is less than $2\gamma^j/\gamma^{j-1} = 2\gamma$. In the case where $\gamma^j \geq N$, the smallest containing range is $(0, N)$. We have that $m > \gamma^{j-1} \geq N/\gamma$, so $N/m < \gamma$ and thus the blowup factor for r_q is less than γ (and also less than 2γ).

□

4.4.3 Memory and Construction Time Analysis

Consider some function parameterized by the dataset and subset size $O(A_f(D, m))$. For example, f may be a construction time function or a memory function. This function evaluated

on all nodes of Algorithm 1 takes

$$O\left(\sum_{j=0}^{\log_\beta N} \beta^j A_f(D, N \cdot \beta^{-j})\right)$$

If A_f is of the form Cm^ρ for $\rho \geq 1$ and for some constant C depending on D, then this is equivalent to:

$$\begin{aligned}
&= O\left(CN^\rho \sum_{j=0}^{\log_\beta N} \beta^{j-j\rho}\right) \\
&= O\left(CN^\rho \sum_{j=0}^{\log_\beta N} (\beta^{1-\rho})^j\right) \\
&= \begin{cases} O(CN^\rho \log_\beta N) & \text{if } \rho = 1 \\ O\left(CN^\rho \frac{1-N^{1-\rho}}{1-\beta^{1-\rho}}\right) = O((\frac{1}{1-\beta^{1-\rho}})CN^\rho) & \text{if } \rho > 1 \end{cases}
\end{aligned}$$

The slow preprocessing version of DiskANN takes $O(N^3)$ for construction time and takes up $O(N(4\alpha)^\delta \log \Delta)$ space, so plugging into these results we have that a β-WST tree with a slow preprocessing DiskANN implementation takes $O(\frac{1}{1-\beta^{-2}}N^3) = O(N^3)$ time to build and has memory size $O((4\alpha)^\delta \log(\Delta) N \log_\beta N)$.

4.5 Experiments

Experiment Setup. We run all query methods on all datasets and filter widths on a 2.20GHz Intel Xeon machine with 40 cores and two-way hyper-threading, 100 MiB L3 cache, and 504 GB of RAM. We run index building using all 80 hyper-threads and restrict queries to 16 threads, and parallelize across (and not within) queries. We run index construction experiments with varying

β on a separate 2.10GHz 4 socket Intel Xeon machine with 96 cores and two way hyperthreading, 132 MiB L3 cache, and 1.47 TB of RAM. We use all threads on the machine for these experiments.

We use a DiskANN [2] index with $\alpha = 1$, *degree* $= 64$, and the construction beam search width set to 500 for all ANN indices, except for the Milvus and VBASE baselines. DiskANN allows searching for $k \geq 1$ nearest neighbors; for simplicity of presentation, we assume that the query beam search width is always set to k. We defer a description of DiskANN and its associated hyperparameters to Section 2.2.2.3.

We note that our theory focuses on the c-ANN problem, which only concerns whether a single c-approximate nearest neighbor is returned. However, as is standard in much of the ANN literature, in our experiments we report *recall* of the top 10 filtered neighbors to the query. While our runtime proofs in Theorem 4.4.1, Lemma 4.4.2, and Lemma 4.4.3 assume that the underlying ANNS algorithm returns a single ANN, in practice, we find that DiskANN has high recall for both single ANN and top-10 ANN. Therefore, we believe that our theoretical analyses still provide insights into our empirical results for top-10 ANN.

Finally, we ensure that our smallest filter fractions are still wide enough such that there are at least 10 points that meet the filter constraint, i.e., we ensure $|D_{(a,b)}| \geq 10$.

Implementation Details. We provide a performant, open source C++ library with Python bindings. [1] Our code is built on the Parlay library [71] for parallel programming and the recent ParlayANN suite of parallel ANN algorithms [3]. We implement a number of memory and performance optimizations, including using a larger base case of 1000 in Algorithm 1 and storing the entire dataset just once across all sub-indices.

[1] https://github.com/JoshEngels/RangeFilteredANN

Filter Fraction. Answering a window filter query that matches almost the entire dataset is a substantially different problem than one that matches almost none of it. Thus, we define the ***filter fraction*** as a way of quantifying where in this filter regime we are: let a query with filter fraction 2^i for $i \leq 0$ be a query whose window filter matches a 2^i fraction of the points in D. For example, a query with filter fraction 2^{-3} has a window filter (a, b) that matches $\frac{1}{8}$ of the dataset, or in other words $|D_{(a,b)}| = \frac{1}{8}|D|$. Queries with a small filter fraction (e.g., 2^{-15}) restrict to a small portion of the dataset, queries with a large filter fraction (e.g., 2^{-2}) restrict to a large portion of the dataset, and queries with a medium filter fraction (e.g., 2^{-8}) restrict somewhere in between. A query with a random filter of fraction 2^i is a query where we randomly select the filter (a, b) so that all possibilities for the $2^i * |D|$ filtered points are equally likely.

Datasets. The datasets we use are listed in Table 4.2 and further explained below. All datasets are available in the same repository as the code and have free for research licenses; more licensing information is included in Appendix B.2.

- `SIFT`, `GloVe`, and `Deep` are ANN datasets from the widely used and standardized ANN benchmarks repository [72]. To adapt them to window search, we uniformly at random generate a label for each point between 0 and 1. We create 16 different query sets $Q_1, \ldots, Q_{16}$, each one using the same 10000 query vectors from ANN benchmarks with random filters of fraction 2^{-i}.
- `Redcaps` is a dataset we created that builds on the RedCaps [73] image and caption dataset, which consists of $11.6M$ Reddit, Imgur, and Flickr images and associated captions. To adapt RedCaps to window search, we use CLIP [15] to generate an embedding for each image and use the timestamp of the image as its label. We create a set of 800 query vectors by asking ChatGPT-4 [74] to come up with queries for an image search system, which we then embed

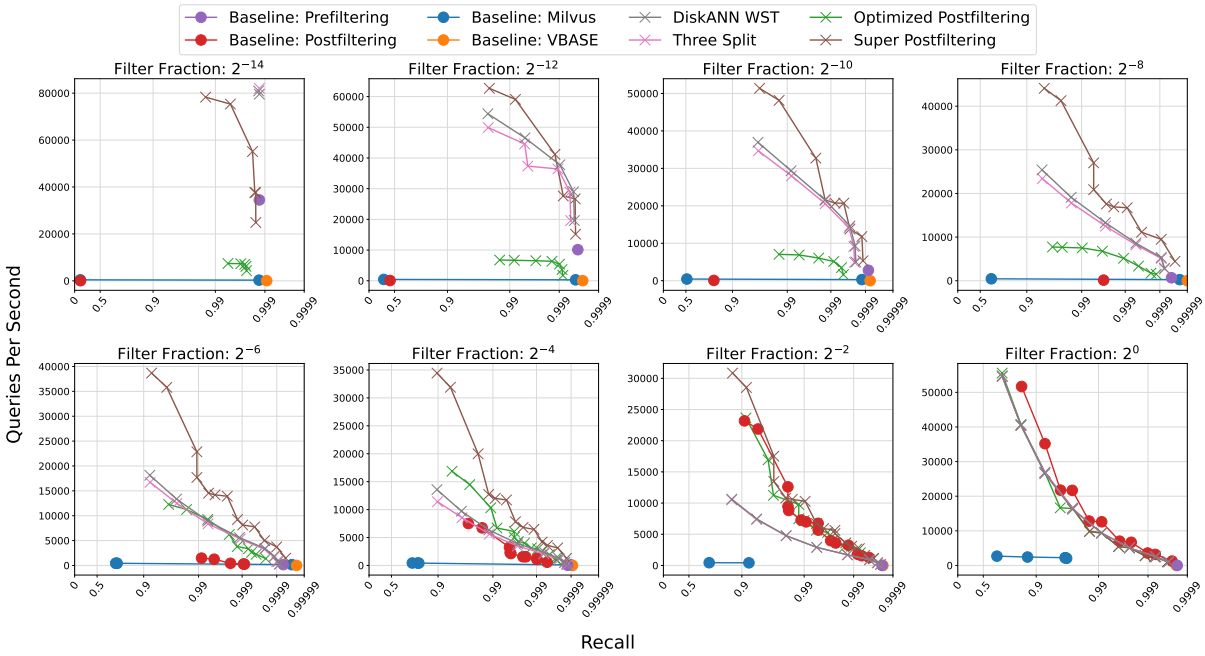


Figure 4.3: Comparison of Pareto frontiers of all methods on window search with different filter fractions on `deep` using 16 threads. Up and to the right is better. On medium filter fraction settings, our methods achieve orders of magnitude more queries per second than the baselines at the same recall levels. Points along the Pareto frontier are denoted by circles for baseline methods and X's for our methods.

using CLIP. See Appendix B.1 for full prompt details. We again create 16 different query sets $Q_1, \ldots, Q_{16}$ using the same 800 query vectors with random filters of fraction 2^{-i}.

- `Adverse` is a synthetic dataset tailored to disadvantage methods that rely on the label and point distributions being independent. The overall idea is to craft a dataset and queries where the points that meet the filter constraint are much farther away from the query than the rest of the dataset. To do this, we let D be a mixture of 100 Gaussians and draw 10000 points from each Gaussian, where the means μ_i are drawn from $N(0, I)$ and each Gaussian is distributed as $N(\mu_i, 0.01 \cdot I)$ (I is the 100 dimensional identity matrix). A point generated from the i'th Gaussian has a label equal to $i + \text{Uniform}(-0.5, 0.5)$, and we generate a query for every pair $i, j \in \{1, \ldots, 100\}$ with $i \neq j$ that consists of a random point drawn from Gaussian i with window filter $(j - 0.5, j + 0.5)$. In other words, each query filters to only points from a different

cluster than it itself is drawn from.

Query Methods and Hyperparameters. We run all of the query methods described in Section 4.3.1, Section 4.3.2, and Section 4.3.3. For all methods that use an arbitrary index A, we use a DiskANN index as described earlier. We run Algorithm 2 with $\beta = 2$ unless specified otherwise; we call this method DiskANNWST in our experiments. We use Prefiltering unmodified as described in Section 4.3.1. We expect Prefiltering to always achieve near 100% recall; it may not be 100% exactly due to numerical precision issues. For Postfiltering, OptimizedPostfiltering, ThreeSplit, and SuperPostfiltering we search over initial k in $[10, 20, 40, 80, 160, 320, 640, 1280]$ and final_multiply in $[1, 2, 3, 4, 8, 16, 32]$. We use the setting $\gamma = 2$ from Theorem 4.4.6 for SuperPostfiltering, which guarantees a worst case blowup factor of 4 using in practice about 1.5 times as much memory as DiskANNWST.

We also compare against Milvus [75] and VBASE [76], two existing systems that support window search.

Milvus treats window search as an instantiation of categorial filter search. Before querying the underlying ANN index, Milvus creates a bitset that marks all of the points that meet the window filter. Then, while traversing the underlying ANN index, Milvus ignores points that are not set in the bitset. We try all supported underlying Milvus indices: HNSW, IVF_PQ, IVF_SQ8, SCANN, and IVF_FLAT. We search over the same beam sizes as for Postfiltering. Milvus does not natively support a batch query with different filters for each point in the batch, so we wrote a Python multiprocessing program that spins up many processes that query the constructed index in parallel.

In addition to the tricks described in Section 4.1, VBASE uses a query planning step to

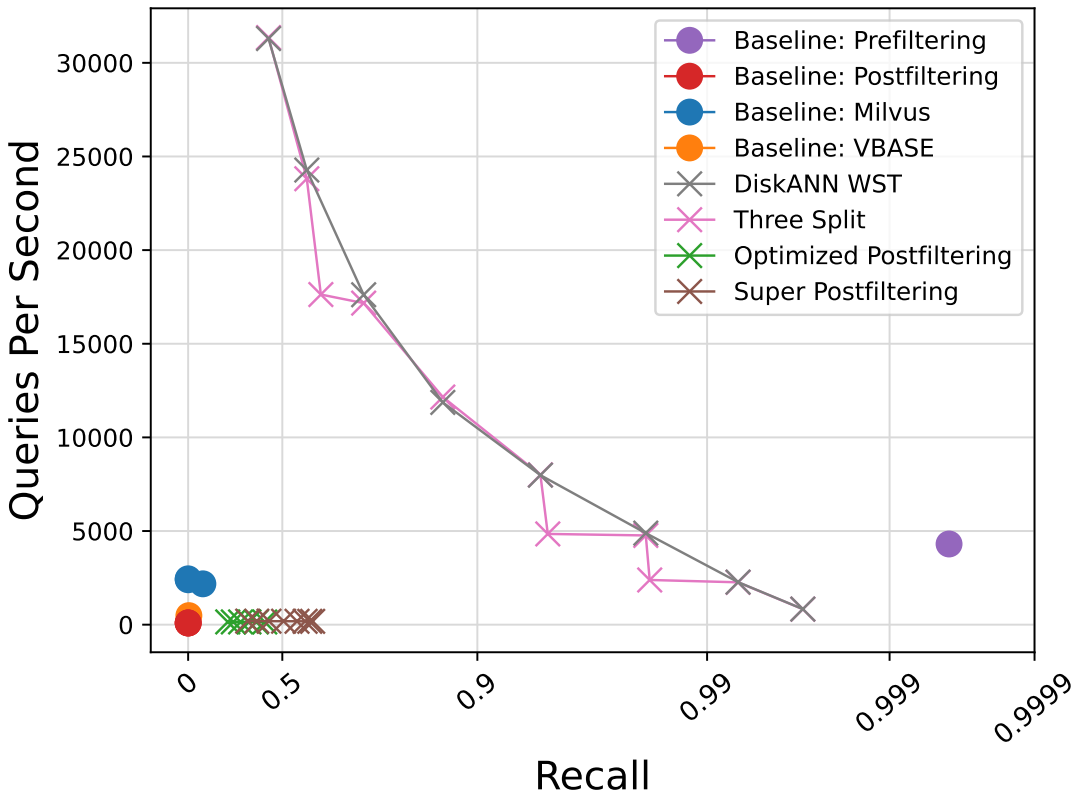


Figure 4.4: Comparison of window search methods on `Adverse`. Up and to the right is better. DiskANNWST and ThreeSplit achieve a good recall vs. latency tradeoff, but besides the Prefiltering baseline all of the other methods are unable to achieve a reasonable recall. All methods are run with 16 threads.

Table 4.3: Speedup of our best method over the best baseline, restricted to hyperparameter settings that yield at least 0.95 recall. All methods are run with 16 threads. We show a speedup across all filter fractions smaller than 2^{-1} on all datasets. We show up to a 75× speedup on medium filter fractions.

Dataset	2^{-11}	2^{-10}	2^{-9}	2^{-8}	2^{-7}	2^{-6}	2^{-5}	2^{-4}	2^{-3}	2^{-2}	2^{-1}	2^{0}
`Deep`	10.49	18.46	35.65	61.21	77.55	24.28	9.35	2.67	1.46	1.39	0.75	0.77
`SIFT`	1.35	1.88	3.05	4.87	8.68	16.51	11.26	4.46	2.26	1.28	0.90	0.92
`GloVe`	1.90	2.27	2.70	3.77	5.02	6.19	9.60	7.62	2.34	1.52	0.92	0.92
`Redcaps`	2.31	3.33	5.47	7.78	10.07	17.22	3.94	3.64	1.75	1.73	0.90	0.90

attempt to predict how many initial results to retrieve, then filters these retrieved results, before finally applying a final top-k ranking to the retrieved results that meet the filter constraint. We were not able to get multiple queries to run in parallel with VBASE (and in the original paper they also only operate in the regime of a single query at a time).

Tradeoff of Queries per Second vs. Recall. We plot the Pareto frontier of recall vs. queries per second of all methods on a selection of filter fractions on `deep` in Figure 4.3. We include

plots of other datasets in B.3.1; the main observations are substantially the same across datasets. Overall, our methods achieve multiple orders of magnitude query speedup at fixed recall levels. SuperPostfiltering does particularly well at lower recall levels of around 0.9 to 0.99, while at high recall levels DiskANNWST and ThreeSplit are the best methods, attaining about the same performance. Additionally, for small filter fractions the Prefiltering baseline is competitive with our methods, whereas for large filter fractions the Postfiltering baseline is competitive. Our methods achieve the largest speedups over baselines in the medium filter fractions. We note that among our methods, OptimizedPostfiltering performs the worst, which we explain with the high worst case blowup of a 2-WST (see Lemma 4.4.5). Finally, we note that the two vector databases that we test against, VBASE and Milvus, are completely dominated by our naive baselines Prefiltering and Postfiltering.

We also plot results on `Adverse` in Figure 4.4. Prefiltering does well, as expected. Furthermore, DiskANNWST offers a good tradeoff between recall and query latency, which makes sense because the query time guarantee from Lemma 4.4.3 (assuming DiskANN also does well for top-10 ANN) holds for any query distribution, even an adversarial one. However, all of the methods that rely solely on postfiltering, as well as the other baselines, fail to achieve meaningful recall. For the postfiltering methods, this result is unsurprising: the index that gets selected for postfiltering has some points that do not meet the query's filter constraints, and by construction these points are frequently closer to the query than the entire target cluster. Surprisingly, although ThreeSplit does use postfiltering as a subroutine, it is able to achieve similar performance to DiskANNWST; this may be because after querying for the indices that make up the center of the label range (which is *not* postfiltering), the label ranges that are left on each side are typically much smaller.

Dataset	DiskANN	2-WST	Super
SIFT	1 m	7.5 m	13.5 m
GloVe	2.5 m	16.5 m	28 m
Deep	16.5 m	2 h	4 h
Redcaps	1.5 h	7 h	18.5 h
Adverse	2.5 m	23 m	41 m

Table 4.4: Build times for different indexing methods across all datasets, rounded to the nearest half unit. Index construction was done using 80 threads.

Dataset	Raw	DiskANN	2-WST	Super
SIFT	0.5 GB	1.0 GB	4.7 GB	7.6 GB
GloVe	0.5 GB	1.1 GB	5.6 GB	9.5 GB
Deep	3.6 GB	6.8 GB	53.2 GB	94.6 GB
Redcaps	23 GB	27.1 GB	79.2 GB	127 GB
Adverse	0.8 GB	0.9 GB	4.6 GB	7.4 GB

Table 4.5: Index sizes for different indexing methods on all datasets, rounded to the nearest 10th of a GB. Note that prefiltering takes just the memory of the original dataset. The "Raw" column is the size of just the dataset.

We also include speedups of our best method (the best of DiskANNWST, OptimizedPostfiltering, SuperPostfiltering, and ThreeSplit) over the best baseline method on filter fractions $i = 2^0, \ldots, 2^{-11}$ on all datasets in Table 4.3 at a recall level of 0.95, and we include the same table at additional recalls in the appendix. These reinforce our findings in Figure 4.3 across other datasets; at a 0.95 recall level, we achieve up to a 75X speedup on Deep, up to a 16X speedup on SIFT, up to a 9X speedup on GloVe, and up to a 17X speedup on Redcaps.

Index Memory and Construction Time. Table 4.4 and Table 4.5 show the construction time and memory sizes for a single DiskANN index (for Postfiltering), a 2-WST with DiskANN indices at each node (for DiskANNWST, OptimizedPostfiltering, and ThreeSplit), and the index for SuperPostfiltering. We also show the memory for just the dataset, which is exactly how much memory Prefiltering needs (the build for Prefiltering is just a sort and takes less than a few seconds across all datasets). We see that the 2-WST takes about 3–8 times as much memory as a single DiskANN index, depending on the dataset, whereas the index for SuperPostfiltering takes about 5–14 times as much memory as a single DiskANN index. The construction times show a similar increase across methods, with a 5–10X increase in build time from DiskANN to 2-WST

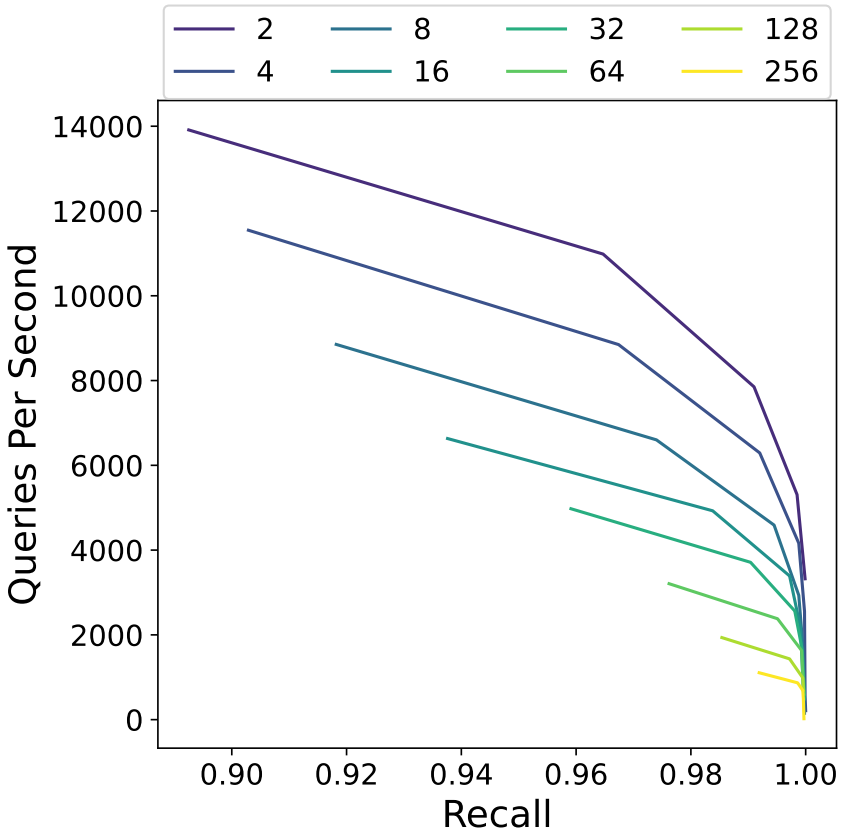


Figure 4.5: Pareto curves of recall vs. throughput on the SIFT dataset for a filter fraction of 2^{-1} and varied branching factors. Up and to the right is better. All trials are run with 16 threads.

and a 10–20X increase from DiskANN to SuperPostfiltering. Finally, we note that while the larger datasets do take signficantly longer to build, we are using a large beam search construction buffer of 500 in order to vary as few hyperparameters as possible, and a smaller buffer size may give faster build times with minimal loss in recall.

Varying β. A larger branching factor β decreases the build time and memory footprint of a DiskANNWST index by reducing the number of levels in the tree and thus reducing the number of graphs that are built (see Figure 4.6 in the appendix for a plot showing the exact reduction in memory and build time as we scale β). However, as shown in Figure 4.5, this comes at the cost of a reduction in recall and latency. These results are substantially the same across different filter fractions; see Figure 4.7 for experiments with more filter fractions.

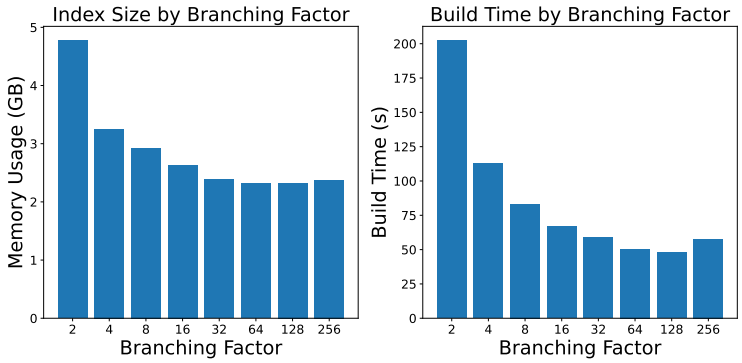


Figure 4.6: Plots of index size and build time for varying branching factors on SIFT. The indices were built using 96 threads. Smaller values are better.

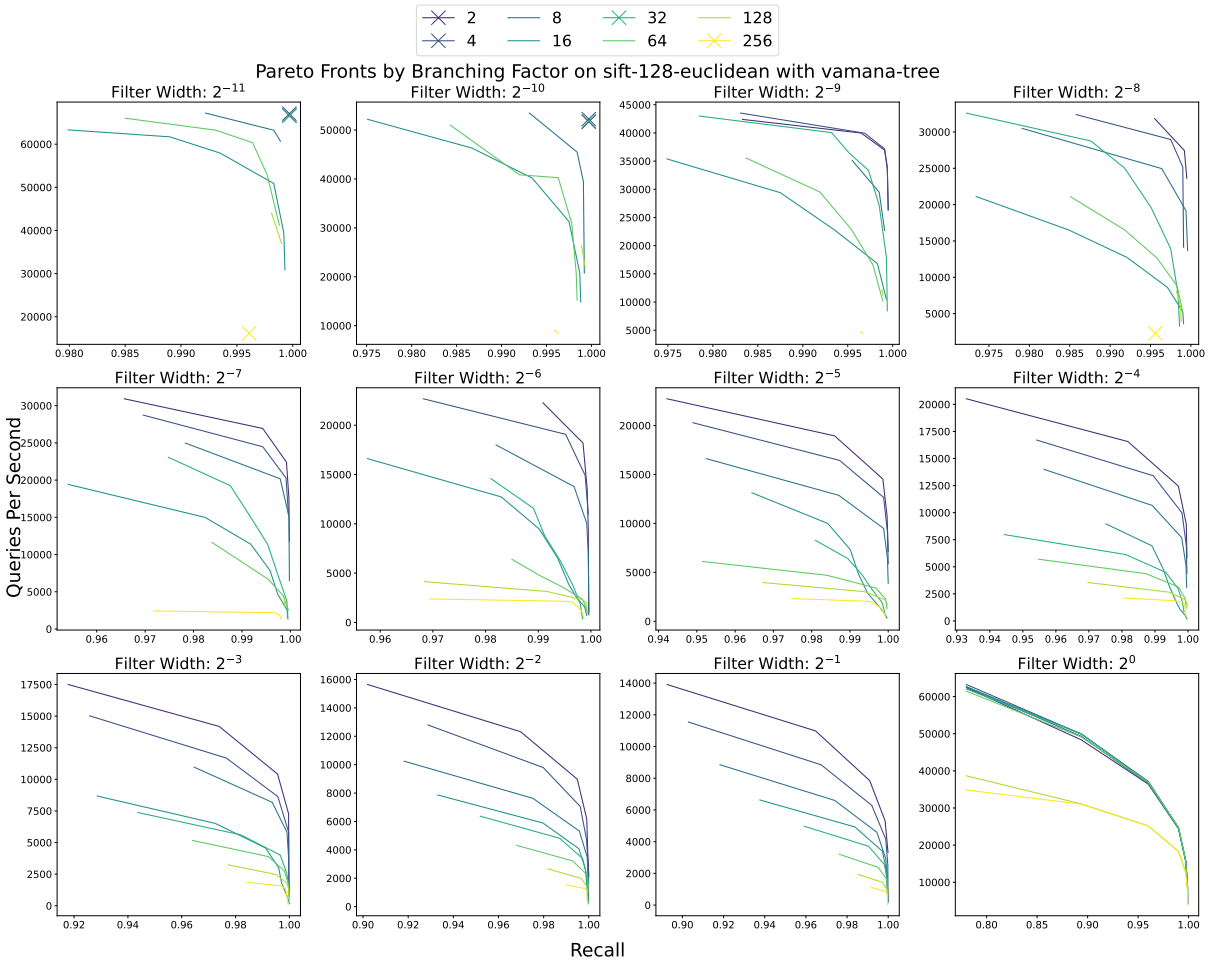


Figure 4.7: Pareto curves of recall vs. throughput on SIFT for varying window sizes and branching factors. The experiment was run using 16 threads. Up and to the right is better.

4.6 Conclusion

We identify window search as an important and overlooked search problem. The methods we present for solving window search give a significant speedup over the state of the art, have strong theoretical guarantees, and are an important step towards ensuring vector databases efficiently support a full range of necessary embedding search operations.

Chapter 5: Future Work

There is much work left to be done making ANNS and filtered ANNS versatile, reliable, and more efficient.

5.1 A Holistic Filtered ANNS

Existing discussion of the filtered ANNS problem either applies no restrictions to the form of predicates, treating them as black box discriminators, or focuses on a constrained type of filter only relevant to a small subset of filtered search cases. Constructing metadata-aware indices which are able to support a broad range of realistic filter scenarios is highly desirable as a step forward in the practicality of filtered ANNS.

5.1.1 Predicates in CNF

Existing work on AND and OR filters suggest the next step to be more general notions of boolean predicates which would bring vector databases closer to supporting arbitrary WHERE clauses in a performant and principled way. To this end, future work should attempt to support arbitrary boolean predicates in conjunctive normal form(CNF). Arbitrarily long OR clauses are already effectively served by existing methods such as FilteredDiskANN [47], and the basic approach used by IVF^2 for AND queries could in principle be extended to arbitrarily many labels.

Combining these methods to serve arbitrary predicates in CNF would improve the practicality of vector databases and is a promising direction for future research.

5.1.2 Combining Boolean and Window Filters

In a typical product search UI, all filters provided for a user to restrict searches by can be represented as boolean ANDs boolean ORs or window filters. As a result, an index effectively serving all three would be sufficient for this real-world application and is therefore highly desirable. As described in this work, both the IVF^2 index and β-WST use Vamana search graphs as components. Either could be composed with the other by replacing these Vamana search graphs and addition of relatively straightforward query processing logic to build an index capable in principle of serving queries dependent on both labels and windows. The feasibility and performance of such an index is not known, and investigations of this and similar ideas for this combined filter case have the potential to be meaningfully impactful.

Appendix B: Window ANN Appendix

B.1 ChatGPT Queries for RedCaps Query Generation

Queries were generated in two sessions with ChatGPT-4. The first consisted of the first query and the second query repeated 4 times. The second consisted of 100 examples copied from the first session and the third and four query. The first query:

> I wish to run an experiment where I generate many possible text queries for my image search system. Can you help me generate queries? I want you to make the queries casual, and be as varied and creative as possible. To the best of your ability, don't repeat yourself! Here are a few example queries: "Funny cat memes" "C++ coding joke" "Vegan recipe with blueberries".

Repeated second query:

> Can you generate 100 more? And still make sure to be as creative and casual as possible, and don't repeat things you've already said! Additionally, try to use unique semantic and syntactic structure where possible.

Third and fourth queries:

> Try not to generate queries with locations in them, because I already have lots of those. Thank you!

and

> Great, now I just need 100 more, again as little repeats as possible, be creative! These can be more ”internet” language, so things like, e.g., ”funny cat memes.”

B.2 Dataset License Information

`SIFT`, `GloVE`, and `DEEP` are released under an MIT license by the ANN benchmarks repository [72]. The original RedCaps license has a restriction to non-commercial use, so our modified `Redcaps` dataset is released under the same restriction. Since we generated the `Adverse` dataset ourselves, we release it under an MIT license.

B.3 Experiments

B.3.1 Pareto Frontiers

See Figure B.1, Figure B.2, and Figure B.3 for full Pareto frontier plots for a representative sample of filter widths on all datasets not included in the main text.

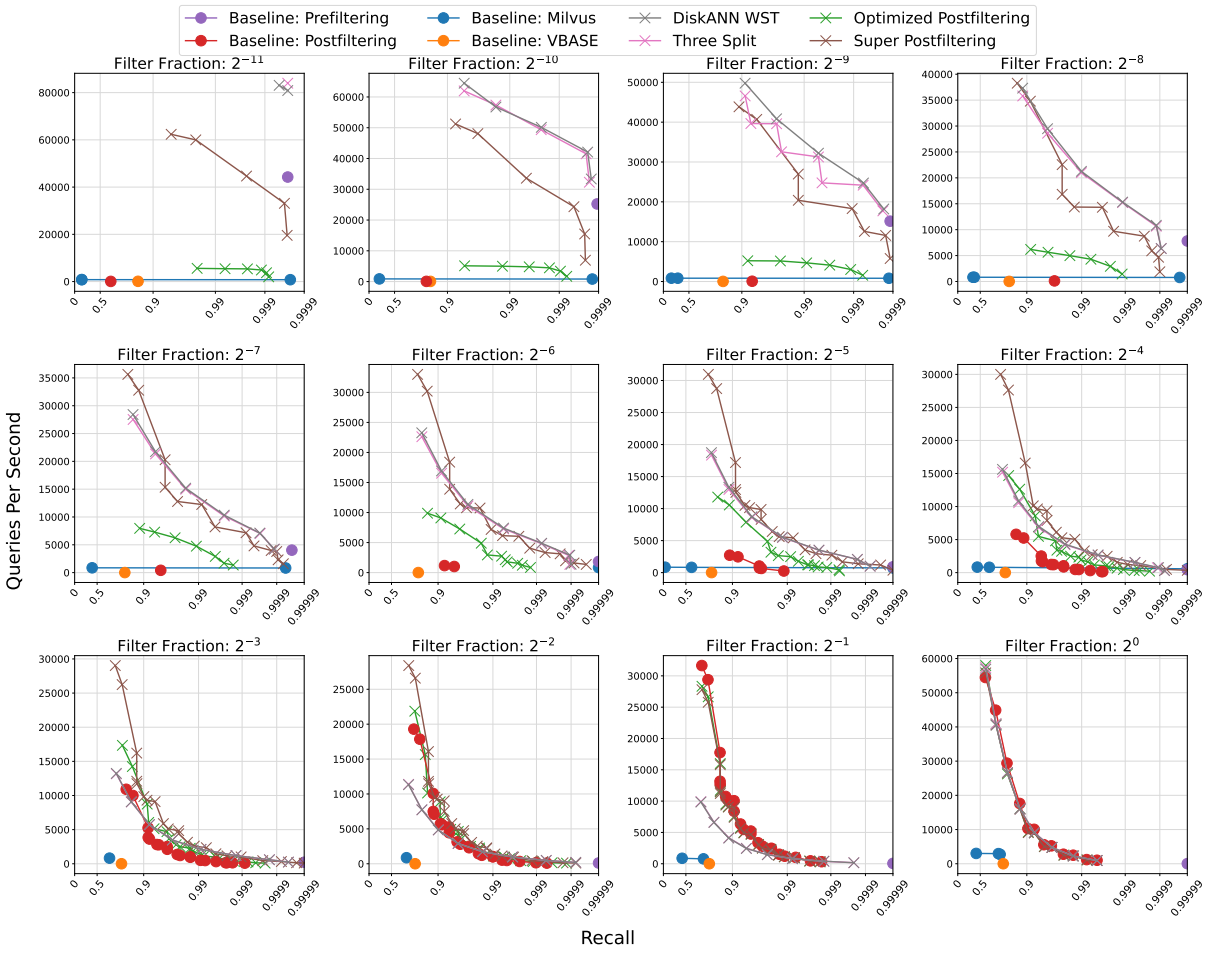


Figure B.1: Comparison of Pareto frontiers of all methods on window search with different filter fractions on `GloVe`. Up and to the right is better. On the medium filter fraction settings, our methods achieve multiple orders of magnitude more queries per second than the baselines at the same recall levels. All methods are run with 16 threads.

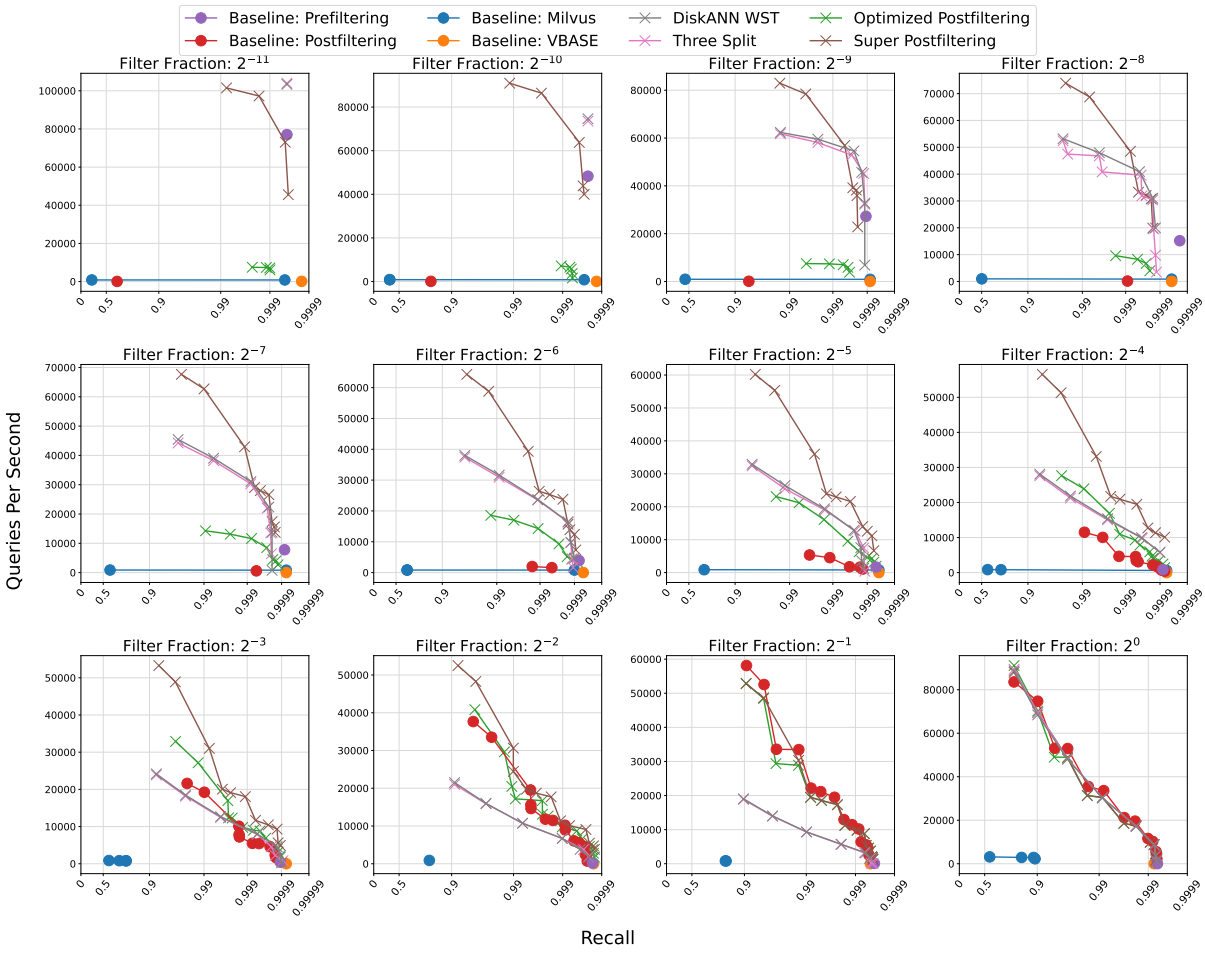


Figure B.2: Comparison of Pareto frontiers of all methods on window search with different filter fractions on `SIFT`. Up and to the right is better. On the medium filter fraction settings, our methods achieve multiple orders of magnitude more queries per second than the baselines at the same recall levels. All methods are run with 16 threads.

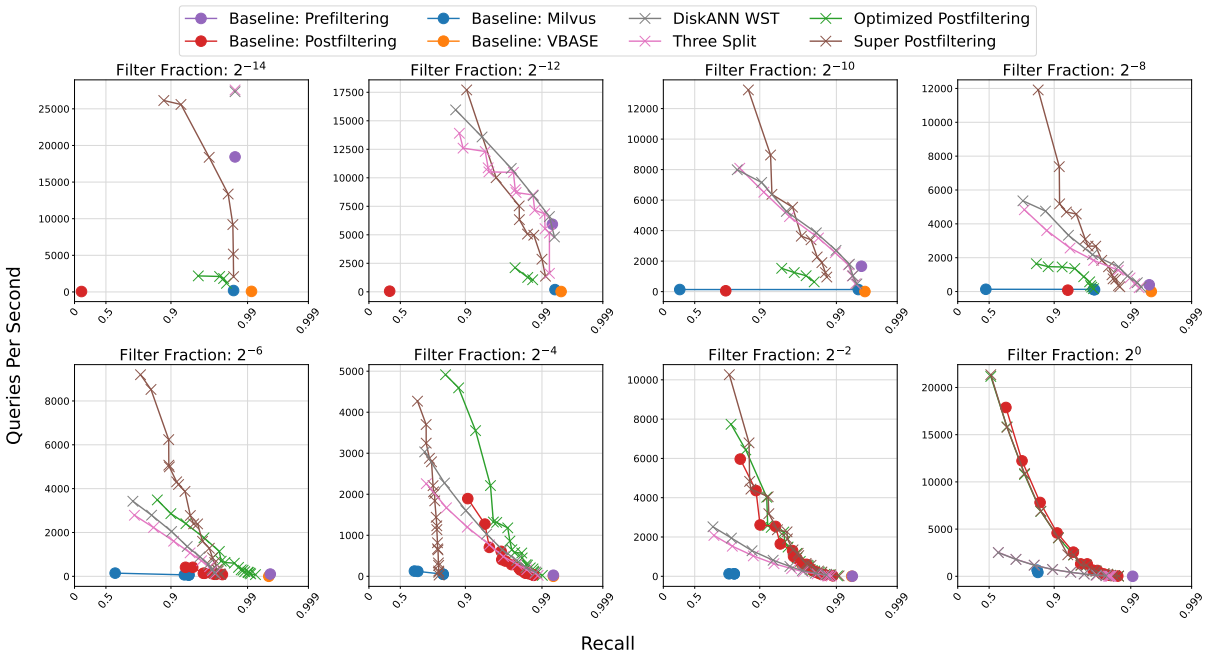


Figure B.3: Comparison of Pareto frontiers of all methods on window search with different filter fractions on `Redcaps`. Up and to the right is better. On the medium filter fraction settings, our methods achieve multiple orders of magnitude more queries per second than the baselines at the same recall levels. All methods are run with 16 threads.

Dataset	2^{-11}	2^{-10}	2^{-9}	2^{-8}	2^{-7}	2^{-6}	2^{-5}	2^{-4}	2^{-3}	2^{-2}	2^{-1}	2^{0}
`Deep` (0.8)	10.49	18.46	35.65	65.40	84.88	26.23	10.23	4.59	2.37	1.33	0.78	0.79
`SIFT` (0.8)	1.35	1.88	3.05	4.87	8.68	16.51	11.26	4.92	2.47	1.39	0.91	0.94
`GloVe` (0.8)	1.90	2.56	3.29	4.90	8.82	16.37	10.57	4.77	1.49	0.90	0.90	0.92
`Redcaps` (0.8)	5.10	7.95	11.86	29.94	36.46	20.69	5.89	2.59	3.27	1.14	0.88	0.88
`Deep` (0.9)	10.49	18.46	35.65	65.40	84.88	26.23	10.23	4.26	2.18	1.23	0.75	0.76
`SIFT` (0.9)	1.35	1.88	3.05	4.87	8.68	16.51	11.26	4.92	2.47	1.39	0.91	0.94
`GloVe` (0.9)	1.90	2.56	3.29	4.46	5.38	9.97	6.96	4.04	1.87	1.57	0.90	0.90
`Redcaps` (0.9)	3.18	5.38	8.92	18.55	19.94	10.46	4.33	1.87	1.70	1.55	0.89	0.89
`Deep` (0.99)	6.80	11.77	22.05	40.06	50.55	9.86	4.33	3.70	1.58	1.47	0.75	0.75
`SIFT` (0.99)	1.35	1.79	2.88	4.53	8.04	10.10	6.73	2.88	1.61	1.01	0.88	0.91
`GloVe` (0.99)	1.90	1.99	2.13	1.96	3.03	4.02	6.00	5.71	4.66	2.67	0.95	0.92
`Redcaps` (0.99)	1.30	1.08	1.50	1.32	1.36	1.87	3.11	0.86	1.82	3.75	N/A	N/A
`Deep` (0.995)	6.80	11.77	22.05	26.07	32.98	11.31	9.91	2.41	1.98	1.49	0.75	0.78
`SIFT` (0.995)	1.35	1.79	2.88	3.20	5.51	10.10	6.73	2.16	1.98	0.96	0.89	0.88
`GloVe` (0.995)	1.90	1.99	1.63	1.96	2.56	3.28	3.93	4.64	4.58	2.97	0.94	0.93
`Redcaps` (0.995)	N/A	0.30	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table B.1: Speedup of our best method over the best baseline, restricted to hyperparameter settings that yield the recall in parenthesis in the dataset column. N/A means that none of our methods achieved that recall (on `Redcaps` this is due to poor DiskANN graph quality).

Bibliography

[1] ANN Benchmarks Authors. Ann-benchmarks. 2023.

[2] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In *Advances in Neural Information Processing Systems*, 2019.

[3] Magdalen Dobson, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. Scaling graph-based anns algorithms to billion-size datasets: A comparative analysis, 2023.

[4] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The Faiss library. *arXiv e-prints*, 2024.

[5] 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. *Proc. VLDB Endow.*, 13(12):3152–3165, 2020.

[6] Apache lucene, 2023.

[7] Vespa use cases: Semi-structured navigation, 2022.

[8] Eric Xing, Michael Jordan, Stuart J Russell, and Andrew Ng. Distance metric learning with application to clustering with side-information. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. MIT Press, 2002.

[9] Suzanna Becker and Geoffrey E. Hinton. Self-organizing neural network that discovers surfaces in random-dot stereograms. *Nature*, 355(6356):161–163, 1992.

[10] Ryan Greene, Ted Sanders, Lilian Weng, and Arvind Neelakantan. New and improved embedding model. Webpage, 2022.

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[12] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*, 2017.

[13] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742, June 2006.

[14] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[15] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763, 2021.

[16] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, page 7–10, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, et al. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*, 2023.

[18] 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 H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[19] Neurips'23 competition track: Big-ann, 2023.

[20] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Approximate nearest neighbor search with window filters, 2024.

[21] Roshan Sumbaly, Mahalia Miller, Hardik Shah, Yang Xie, Sean Chang Culatana, Tim Khatkevich, Enming Luo, Emanuel Strauss, Gergely Szilvasy, Manika Puri, Pratyusa Manadhata, Benjamin Graham, Matthijs Douze, Zeki Yalniz, and Hervé Jegou. Using ai to detect covid-19 misinformation and exploitative content, 2020.

[22] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 604–613. ACM, 1998.

[23] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[24] Ninh Pham and Tao Liu. Falconn++: A locality-sensitive filtering approach for approximate nearest neighbor search. *CoRR*, abs/2206.01382, 2022.

[25] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, 2013.

[26] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 1225–1233, 2015.

[27] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. PUFFINN: parameterless and universally fast finding of nearest neighbors. In *Annual European Symposium on Algorithms (ESA)*, volume 144, pages 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[28] Alex Klibisz. Tour de elastiknn. Webpage, 2021.

[29] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33:117–28, 01 2011.

[30] Gaurav Gupta, Tharun Medini, Anshumali Shrivastava, and Alexander J. Smola. BLISS: A billion scale index using iterative re-partitioning. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 486–495. ACM, 2022.

[31] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: highly-efficient billion-scale approximate nearest neighborhood search. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 5199–5212, 2021.

[32] Matthijs Douze, Hervé Jégou, and Florent Perronnin. Polysemous codes. In *Computer Vision - ECCV 2016*, volume 9906 of *Lecture Notes in Computer Science*, pages 785–801. Springer, 2016.

[33] Yan Xia, Kaiming He, Fang Wen, and Jian Sun. Joint inverted indexing. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3416–3423, 2013.

[34] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, jun 1990.

[35] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

[36] Javier Alvaro Vargas Muñoz, Marcos André Gonçalves, Zanoni Dias, and Ricardo da Silva Torres. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognit.*, 96, 2019.

[37] Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA)*, pages 271–280. ACM/SIAM, 1993.

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

[39] Piotr Indyk and Haike Xu. Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations. In *Advances in Neural Information Processing Systems*, 2023.

[40] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4):744–755, 2014.

[41] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML)*, volume 119, pages 3887–3896. PMLR, 2020.

[42] Pinecone: Vector database for vector search, 2023.

[43] Weaviate: The ai native vector database, 2023.

[44] Qdrant. Qdrant: High-performance, massive-scale vector database for ai. `https://github.com/qdrant/qdrant`, 2024. Accessed: 2024-04-29.

[45] The Milvus Project. Milvus: open source vector database built for scalable similarity search, 2023.

[46] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. Navigable proximity graph-driven native hybrid queries with structured and unstructured constraints. *CoRR*, abs/2203.13601, 2022.

[47] 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*, page 3406–3416, 2023.

[48] Weijie Zhao, Shulong Tan, and Ping Li. Constrained approximate similarity search on proximity graph. *CoRR*, abs/2210.14958, 2022.

[49] Gaurav Gupta, Jonah Yi, Benjamin Coleman, Chen Luo, Vihan Lakshman, and Anshumali Shrivastava. CAPS: A practical partition index for filtered similarity search, 2023.

[50] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011.

[51] George K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.

[52] Clémentine Cottineau. Metazipf. a dynamic meta-analysis of city size distributions. *PLOS ONE*, 12(8):1–22, 08 2017.

[53] Z. K. Silagadze. Citations and the zipf-mandelbrot's law, 1999.

[54] Sebastian Bruch, Franco Maria Nardini, Amir Ingber, and Edo Liberty. Bridging dense and sparse maximum inner product search. *arXiv preprint arXiv:2309.09013*, 2023.

[55] J. MacQueen. Some methods for classification and analysis of multivariate observations. In Lucien M. Le Cam and Jerzy Neyman, editors, *Berkeley Symposium on Mathematical Statistics and Probability, 1967*, volume 5.1, pages 281–297, 1967.

[56] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[57] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[58] Bart Thomee, David A. Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. Yfcc100m: the new data in multimedia research. *Communications of the ACM*, 59(2):64–73, January 2016.

[59] cohere.ai. Cohere wikipedia-22-12-en embeddings. `https://huggingface.co/datasets/Cohere/wikipedia-22-12-en-embeddings`, 2022. Accessed: 2024-03-30.

[60] Herve Jegou, Matthijs Douze, Jeff Johnson, Lucas Hosseini, Chengqi Deng, and Alexandr Guzhva. Faiss wiki. Webpage, 2023.

[61] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.

[62] Jon Louis Bentley. Algorithms for Klee's rectangle problems. Technical report, Technical Report, 1977.

[63] Giulio Ermanno Pibiri and Rossano Venturini. Practical trade-offs for the prefix-sum problem. *Software: Practice and Experience*, 51(5):921–949, 2021.

[64] Yihao Huang, Shangdi Yu, and Julian Shun. Faster parallel exact density peaks clustering. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*, pages 49–62. SIAM, 2023.

[65] Shangdi Yu, Joshua Engels, Yihao Huang, and Julian Shun. Pecann: Parallel efficient clustering with graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2312.03940*, 2023.

[66] Yewang Chen, Xiaoliang Hu, Wentao Fan, Lianlian Shen, Zheng Zhang, Xin Liu, Jixiang Du, Haibo Li, Yi Chen, and Hailin Li. Fast density peak clustering for large scale data based on knn. *Knowledge-Based Systems*, 187:104824, 2020.

[67] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122, 2008.

[68] Liudmila Prokhorenkova and Aleksandr Shekhovtsov. Graph-based nearest neighbor search: From practice to theory. In *International Conference on Machine Learning*, pages 7803–7813, 2020.

[69] Kenneth L Clarkson. Nearest neighbor queries in metric spaces. In *ACM Symposium on Theory of Computing*, pages 609–617, 1997.

[70] Robert Krauthgamer and James R Lee. Navigating nets: simple algorithms for proximity search. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 798–807, 2004.

[71] Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib–a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 507–509, 2020.

[72] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.

[73] Karan Desai, Gaurav Kaul, Zubin Aysola, and Justin Johnson. RedCaps: Web-curated image-text data created by the people, for the people. In *Advances in Neural Information Processing Systems*, 2021.

[74] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[75] Milvus-docs: Conduct a hybrid search, 2022.

[76] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 377–395, 2023.