ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

“ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ & ΥΠΗΡΕΣΙΕΣ”

BLC-RFANN: B+ tree with Lazy Caching for RFANN

By

Panagiotis Lampropoulos

Submitted

in partial fulfillment of the requirements for the degree of

Master of Information Systems and Services

at the

UNIVERSITY OF PIRAEUS

March 2026

Thesis Supervisor: CHRISTOS DOULKERIDIS

Title: PROFESSOR

Συγγραφέας / Author ΛΑΜΠΡΟΠΟΥΛΟΣ ΠΑΝΑΓΙΩΤΗΣ / LAMPROPOULOS PANAGIOTIS

ΣΕΛΙΔΑ ΕΓΚΥΡΟΤΗΤΑΣ

Ονοματεπώνυμο Φοιτητή/Φοιτήτριας: ……..

Τίτλος Μεταπτυχιακής Διπλωματικής Εργασίας: …….

Η παρούσα Μεταπτυχιακή Διπλωματική Εργασία υποβάλλεται ως μερική εκπλήρωση των απαιτήσεων του Προγράμματος Μεταπτυχιακών Σπουδών "Πληροφοριακά Συστήματα & Υπηρεσίες" του Τμήματος Ψηφιακών Συστημάτων του Πανεπιστημίου Πειραιώς και εγκρίθηκε στις …… [ημερομηνία έγκρισης] από τα μέλη της Εξεταστικής Επιτροπής.

Εξεταστική Επιτροπή

Επιβλέπων/ουσα (Τμήμα Ψηφιακών Συστημάτων, Πανεπιστήμιο

Πειραιώς)……[ονοματεπώνυμο, βαθμίδα]

Μέλος Εξεταστικής Επιτροπής: ……[ονοματεπώνυμο, βαθμίδα]

Μέλος Εξεταστικής Επιτροπής: ……[ονοματεπώνυμο, βαθμίδα]

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΑΥΘΕΝΤΙΚΟΤΗΤΑΣ

Ο/Η……., γνωρίζοντας τις συνέπειες της λογοκλοπής, δηλώνω υπεύθυνα ότι η παρούσα εργασία με τίτλο «……», αποτελεί προϊόν αυστηρά προσωπικής εργασίας και όλες οι πηγές που έχω χρησιμοποιήσει, έχουν δηλωθεί κατάλληλα στις βιβλιογραφικές παραπομπές και αναφορές. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή.
Επιπλέον *δηλώνω υπεύθυνα ότι η συγκεκριμένη Μεταπτυχιακή Διπλωματική Εργασία έχει συγγραφεί από εμένα προσωπικά και δεν έχει υποβληθεί ούτε έχει αξιολογηθεί στο πλαίσιο κάποιου άλλου μεταπτυχιακού ή προπτυχιακού τίτλου σπουδών, στην Ελλάδα ή στο εξωτερικό.*
Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του πτυχίου μου. Σε κάθε περίπτωση, αναληθούς ή ανακριβούς δηλώσεως, υπόκειμαι στις συνέπειες που προβλέπονται τις διατάξεις που προβλέπει η Ελληνική και Κοινοτική Νομοθεσία περί πνευματικής ιδιοκτησίας.

Ο/Η ΔΗΛΩΝ/ΟΥΣΑ

Ονοματεπώνυμο:
Αριθμός Μητρώου:
Υπογραφή:

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Dr. Christos Doulkeridis for his guidance and continuous support throughout the creation of this thesis.

Moreover, I would like to thank the thesis committee for both their time and their valuable input.

Finally, I wish to thank my family and all my friends for both their emotional (and creative) support during this semester. Moreover, I would especially like to thank my friend Dimitrios Tsirmpas for his feedback after reviewing a draft of this thesis.

Table of contents

Table of figures

List of tables

List of algorithms

1. Abstract

The following thesis examines a new algorithm for performing Range Filtered Approximate Nearest Neighbor (or RFANN for short) queries named *BLC-RFANN* (*B+ tree with Lazy Caching for RFANN*). Existing RFANN methods tend to work with static datasets (i.e. datasets whose data remain the same throughout time) and moreover lack any type of caching mechanisms to improve query execution speed. BLC-RFANN's two basic components (the B+ tree and the caching system) try to address these two limitations.

Given a set of data points (each data point consisting of a vector and a numeric value) and an input vector, RFANN searches try to find the (approximate) nearest data points to the input vector and whose numeric value is in a specified, continuous range. It is an issue that has attracted an increasing amount of interest in both academia and industry for its wide range of applications.

The method which we discuss is based on the B+ tree, an (existing) variation of the B-tree where data is only stored in the leaf level (instead of being stored on every node of the tree), and an intricate caching mechanism which we introduce in this study in order to reduce query response time.

Our goal in this study is to compare our method's performance with other, state-of-the-art algorithms used for RFANN queries in various metrics, such as index build time (the amount of time needed in order for an index to be created), the memory footprint of an index (how much memory space does an index use), the amount of queries processed per second and recall (the number of true positives compared to the real positives), which is the most commonly used metric for measuring the validity of RFANN metrics.

Our experiments show that BLC-RFANN requires less time and (generally speaking) less space to build its index compared to its competitors, while also achieving a very high level of recall (up to 100% with specific configuration), at a cost of reduced throughput.

However, features such as our caching mechanism manage to improve the number of queries processed per second, and in specific scenarios (e.g., where selectivity is low or scenarios where the datasets are very large compared to the available resources) BLC-RFANN manages to deliver a comparable throughput as other modern RFANN algorithms.

2. Περίληψη (στα Ελληνικά)

Η παρακάτω Μεταπτυχιακή Διπλωματική Εργασία (Μ.Δ.Ε.) εξετάζει τον αλγόριθμο BLC-RFANN (*B+ tree with Lazy Caching for RFANN* ή B+ δέντρο με "νωθρή" προσωρινή μνήμη για το RFANN), μία νέα μέθοδο η οποία εκτελεί ερωτήματα *Προσεγγιστικής Εύρεσης Κοντινότερων Γειτόνων με Περιορισμό Εύρους* (η οποία είναι γνωστή με το αγγλικό της ακρωνύμιο *RFANN*). Οι υπάρχουσες μέθοδοι είναι σχεδιασμένες να λειτουργούν με στατικά σύνολα δεδομένων (δηλαδή σύνολα δεδομένων όπου τα δεδομένα παραμένουν τα ίδια σε βάθος χρόνου), ενώ δεν έχουν κάποιον ιδιαίτερο μηχανισμό κρυφής μνήμης για την βελτίωση της ταχύτητας εκτέλεσης. Τα δύο βασικά στοιχεία της μεθόδου μας (δηλαδή το B+ δέντρο και ο μηχανισμός κρύφης μνήμης) προσπαθούν να αντιμετωπίσουν αυτούς τους περιορισμούς.

Δοθέντος ενός συνόλου δεδομένων (στο οποίο κάθε δεδομένο αποτελείται από ένα διάνυσμα και μία αριθμητική τιμή) και ενός διανύσματος-εισόδου συνοδευόμενου από ένα εύρος τιμών, ο στόχος μας είναι να βρούμε τους κοντινότερους γείτονες του διανύσματος-εισόδου, των οποίων οι αριθμητικές τιμές ανήκουν στο προαναφερθέν εύρος. Είναι ένα ερώτημα το οποίο παρουσιάζει πλήθος εφαρμογών.

Η παραπάνω μέθοδος βασίζεται στο B+ δέντρο, μία (ήδη υπάρχουσα) παραλλαγή του B-δέντρου όπου τα δεδομένα αποθηκεύονται μόνο στα φύλλα του δέντρου (και όχι σε κάποιον άλλο εσωτερικό κόμβο) και έναν σύνθετο μηχανισμό κρυφής μνήμης (cache) ο οποίος κατασκευάστηκε για να μειώσει τον χρόνο απάντησης των ερωτημάτων.

Ο στόχος μας σε αυτήν την εργασία είναι να συγκρίνουμε την επίδοση της με άλλους μοντέρνους αλγορίθμους που χρησιμοποιούνται για ερωτήματα RFANN σε διάφορες μετρικές, όπως τον χρόνο δημιουργίας των ευρετηρίων τα οποία παράγουν οι μέθοδοι, το μέγεθος χώρου που χρειάζεται για να αποθηκευτούν στην μνήμη τα εν λόγω ευρετήρια, τον αριθμό ερωτημάτων ο οποίος εκτελείται ανά δευτερόλεπτο, καθώς και την ανάκληση (ή *recall*, δηλαδή τον αριθμό των ανακληθέντων σωστών αποτελεσμάτων ως προς τον αριθμό των αληθινών αποτελεσμάτων), η οποία είναι η πιο συνήθης μετρική για να μετρήσουμε την ορθότητα των απαντήσεων ενός αλγορίθμου.

Τα πειράματα τα οποία εκτελέσαμε δείχνουν ότι ο BLC-RFANN χτίζει το ευρετήριό του πιο γρήγορα και χρειάζεται (κατά μέσο όρο) λιγότερο χώρο αποθήκευσης σε σχέση με τους ανταγωνιστές του, καθώς επίσης ότι έχει ένα πολύ υψηλό επίπεδο ανάκλησης (με συγκεκριμένες ρυθμίσεις μπορεί να φτάσει το 100%), αν και προφανώς με ένα μικρότερο ρυθμό εκτέλεσης ερωτημάτων.

Παρόλα αυτά, λειτουργίες όπως ο μηχανισμός μας κρυφής μνήμης κατάφεραν να αυξήσουν τον ρυθμό εκτέλεσης ερωτημάτων, ενώ σε κάποια συγκεκριμένα σενάρια (όπως για ερωτήματα όπου το εύρος τιμών είναι σχετικά μικρό, ή όταν το σύνολο δεδομένων είναι μεγάλο ως προς τους διαθέσιμους υπολογιστικούς πόρους) ο BLC-RFANN μπορεί να εκτελέσει ερωτήματα με την ίδια ταχύτητα που έχουν οι μοντέρνες μέθοδοι RFANN.

3. Introduction – Structure of the study

This study is submitted as a partial fulfillment of the requirements for the M.Sc. program in "Information Systems and Services" of the University of Piraeus. The subject of study is a new (albeit somewhat unconventional) method for performing *Range-Filtered Approximate Nearest Neighbor queries* (or *RFANN* queries for short) named *BLC-RFANN* (*B+ tree with Lazy Caching for RFANN*) and examines its effectiveness compared to various other techniques.

The study begins by providing the necessary background for understanding RFANN and previous research in Section 4 (*Background – RFANN Search*). We start Section 4 by presenting RFANN as a problem, while also describing many of its practical applications. Afterwards, we continue by showcasing the characteristics of various characteristics of already existing algorithms in Section 4.1 (*Existing methods for solving RFANN queries*), while then getting into more detail about some of the most well-known methods from Section 4.2 (*iRangeGraph*) to Section 4.6 (*OptimizedPostfiltering/SuperPostfiltering*), studying on what types of structures they create to perform queries, on how they build these structures, and on how they perform queries. Finally, we present existing academic literature that tries to compare various RFANN methods and briefly showcase their results in Section 4.7 (*Existing experimental studies and their results*).

In Section 5 (*Implementation*) we elaborate further on BLC-RFANN itself, outlining how BLC-RFANN performs various operations such as index construction, querying and element update (such as element addition or removal). We first start by showing thoroughly the two basic parts of our method, which are the B+ tree and the lazy caching mechanism in Sections 5.1 (*B+ Tree*) and 5.2 (*Caching mechanism*) respectively. In these sections, we discuss on how those components work in an isolated manner, showcasing scenarios such as element insertion, element removal and querying. Afterwards, in Section 5.3 (*BLC-RFANN*) we present our algorithm in its entirety, thus displaying how the two components mentioned above (the B+ tree and the lazy caching mechanism) work together. Finally, we briefly go through technical aspects of BLC-RFANN's implementation in Section 5.4 (*Technical implementation*), providing a basic structure of code and the executable files.

In the following section (*Experimental Setup*), we present the structure of our experiments in depth, first by listing the experiments themselves (Section 6.1, *Planned Experiments*), then introducing the datasets we are going to use RFANN (Section 6.2, *Datasets used in the experiments*), afterwards listing the RFANN methods we are going to compare BLC-RFANN against (Section 6.3, *RFANN methods used in the experiments*) as well as describing the metrics we are going to use to measure the methods' efficiency (Section 6.4, *Metrics used in the experiments*), and the environment where the experiments will be conducted (Section 6.5 *Experiment environment / Hardware specs*).

In Section 7 (*Experiment results*) we showcase the results of our results in their entirety, also offering our remarks, where we provide our broader findings about BLC-RFANN in Section 8 (*Conclusions/Discussion*), while also presenting possible avenues for future research (in Section 8.1, *Possible areas of future interest*).

Finally, in the *References* section, we present all papers, books, articles, online documentation and miscellaneous resources cited in this study.

4. Background - RFANN Search

In today's digitalized world, with its ever-larger volume of data, there is an ever-increasing demand for mechanisms to store, retrieve and extract value from data in a manner as efficient as possible. We want to use these data for a very large number of use cases, which generally start from business and research, but they can also lead to the everyday and mundane. For example, let's suppose we are in a car dealership, and we want to find the car closest to our wants, but whose price is within our (very strict) budget. This is a scenario that most of us will go through at least once in our lives.

We can consider each car to essentially be a combination of two parts: its characteristics (its engine, its color, its maximum speed, the number of doors it has, and many others) and its price. In turn, these characteristics can be represented as a vector, of which each dimension corresponds to a single characteristic (or a linear combination thereof).

Our goal then becomes to find the car whose vector is the most similar to the characteristics that we want (which can also be represented as a vector), and whose price is within our constraints. Even if we do not exactly find the exact closest car (because the computational cost of finding it might be too high), we want to find one that is very similar to what we want[1].

The problem we are describing is called *Range Filtered Approximate Nearest Neighbor Search* (or *RFANN search* for short). Our goal is to find the K (where K is a positive integer) approximate nearest neighbours of a specific data object, whose values are inside a specified data range. This is an issue that has attracted a large amount of interest in academia and industry alike [1]. RFANN search has been used for a very large and diverse set of applications which include (but are not limited to) product search (such as our car example shows), vehicle or face identification (for example trying to identify a vehicle or a person passing through a street in a specific point in time) [2], search engines (search results that were made in a specific time-range) [3], and others.

The main challenges in creating RFANN algorithms are to return accurate answers to queries, while also maintaining high throughput. Moreover, if the algorithm has to create any type of index, it should be memory-efficient, while also being fast to build (and ideally being able to handle any changes on the dataset).

4.1. Existing methods for solving RFANN queries

Many methods have been created in recent years in order to perform RFANN searches, using many different approaches. The most common ones are graph-based methods [1, 2, 3, 4], and quantization-based[2] methods (such as the IVF indices used by *Milvus* [5]). Moreover,

[1] We use Euclidean distance as a vector similarity metric in this paper, although RFANN algorithms can (theoretically) also be applied with any other similarity metric

[2] For more detail, see section 4.5

tree-based methods (such as BLC-RFANN, the method we propose) and hashing based methods are also used, albeit much less frequently [6].

Graph-based methods generally tend to construct neighborhood graphs that are quite efficient on query execution time and recall [1] (the exact definition of these metrics is defined in Section 4.7), which is a possible cause for their widespread usage. On the other hand, quantization-based methods tend to be efficient on the amount of time needed to build an index and the amount of space needed to store the index in memory [6].

Another way to group RFANN methods is depending on their filtering strategy (that is on how they filter out-of-range items). Many methods (such as *iRangeGraph* and *Milvus* [1, 5, 6]) use what is called *pre-filtering*. This means that before they start performing a approximate nearest neighbor search, those methods filter all data that are outside the search space, and thus only perform the approximate nearest neighbor search on the data objects inside the range of the query [1, 6].

The opposite approach to pre-filtering is called *post-filtering*. Methods that perform post-filtering first try to find the approximate nearest neighbors in the whole dataset, and then try to filter the results, in order to find the K nearest neighbors. It is used by methods such as *Faiss-HNSW* and *SuperPostfiltering* [3, 6].

A third approach that is employed by various methods (such as *SeRF*, *FilteredVamana* and *StitchedVamana* [2, 4]) is called *joint-filtering* or *in-filtering*. In-filtering tries to integrate filtering during the search itself, by only visiting data objects that are inside the query boundaries. For example, a graph-based in-filtering method might prune paths leading to nodes that are outside the requested data range [6].

All three methods have advantages and disadvantages. Pre-filtering can be very efficient when *selectivity* is low (selectivity being the percentage of data items that are inside the query range), that is if only a small part of the dataset is inside the query range [6]. On the other hand, if selectivity is very high (when many, if not most of the items of the dataset are situated within the query bounds), pre-filtering might degenerate into a sequential search [1]. Newer methods such as *UNIFY* try to provide more flexibility in selecting filtering strategy (by providing the *Sel_low* and *Sel_High* parameters). *UNIFY* uses these parameters to alternate between pre-filtering (if selectivity is below *Sel_low*), post-filtering (if selectivity is over *Sel_High*) and in-filtering (if selectivity is between *Sel_low* and *Sel_High*) [6, 7].

Post-filtering and in-filtering however face the opposite problems. While they are quite effective when selectivity is high, they will face issues in low selectivity queries. For example, post-filtering will be less efficient, since it will need to discard many out-of-range candidate nodes [1]. In-filtering will also face issues, the (actual) nearest neighbors might become unreachable (since they might traverse through nodes that are out of range), thus reducing recall [1, 6].

A third way to group methods that perform RFANN queries is depending on the type of structures they use to create their indices. Graph-based RFANN methods most often tend to

use one (or a variation) of the three following structures: *Relative Neighbor Graphs* (or *RNG*s), *Vamana graphs* and *Hierarchical Navigable Small World Graphs* (or *HNSW graphs*).

The first type of structure used is the *Relative Neighbor Graph* (or *RNG*). RNGs are graphs that store the approximate nearest neighbors of each node. They are created from scratch by finding the (approximate) nearest neighbors of each datum in the dataset. However, RNGs remove redundant edges in order to reduce the total amount of edges in the graph. A redundant edge is any edge between two nodes u and v where there is another node u' where $\delta(u,u') < \delta(u,v)$ and $\delta(v,u') < \delta(v,u)$. A commonly used method that utilizes the RNG is iRangeGraph [1], however many structures used by other RFANN search methods (such as *HNSW*, *NSW* and *Vamana* graphs) are based on RNG or one of its variants [1, 6]

Vamana graphs [4, 8] are a variation of RNGs, which are also used for various RFANN and *LFANN* (*Label-filtered Approximate Neighbor*) algorithms. Vamana graphs are created by first generating a randomly generated graph between the various vectors of the dataset, and then performing RFANN searches for each vector (like RNGs). They also use the same pruning algorithm as RNG to prune redundant edges. Due to the similarities mentioned above, many considered Vamana graphs to be a variation of RNGs [6]. Examples of methods that use Vamana graphs are *FilteredVamana* and *StitchedVamana* [4].

Other commonly used structures used for RFANN search (or other adjacent problems) are *Hierarchical Navigable Small World graphs* (or *HNSW graphs* for short). HNSW graphs are based on the so-called *Navigable Small World graphs*, which are graphs that are created for allowing efficient communication between across a graph, by creating edges between far-away nodes, which reduce the number of hops required to travel between distant vertices (this is why they are named “small world”). As their name suggests, HNSW graphs try to add a hierarchal aspect to NSW trees, by separating links between nodes into levels, depending on their length (larger hops going to higher levels of the hierarchy) [9]. Methods that use HNSW graphs to perform RFANN queries include *SeRF* [2] and *Milvus* [5] (although it is not the only type of structure used by Milvus).

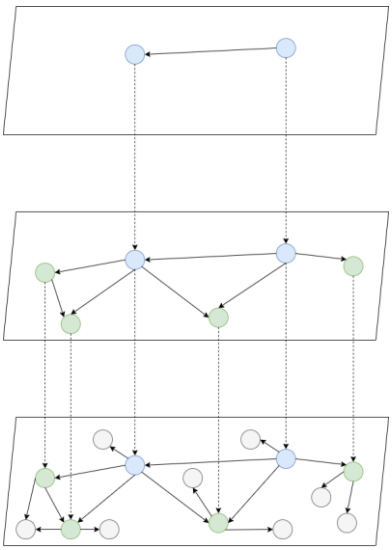


Figure 1: Example of a Hierarchical Navigable Small World graph. Nodes and edges are being stored on each level. The higher the level, fewer nodes are being stored, and the edges tend to connect farther away nodes.

A final graph-based approach used for both RFANN and LFANN is *KGraph*, which similarly to Vamana, starts with a randomly-generated graph (similar to Vamana graphs), and then improves it by using the principle of "neighbors are more likely to be neighbors of each other" [10]. It is used by methods such as *NHQ* [6].

As for quantization-based methods, a structure that is often used for RFANN algorithms (such as Milvus [5]) is the *Inverted File Index* (or *IVF index*). IVF indices divide the dataset into partitions using k-means clustering. When a query is performed, only partitions nearest to the query vector will end up being processed. Moreover, there are many compression techniques such as *SQ8* (*Scalar Quantization*) and *PQ* (*Product Quantization*), which can reduce the amount of space needed to store the vectors [5] (they will be explained in detail in the Milvus section).

A common problem for most of the existing methods is the fact they do not allow dynamic data insertion and deletion. This means that if new data is added to the dataset (or if data is removed from it) we will need to build the index from scratch. This quickly becomes a problem, since index building can be a very time-consuming process for many methods [6]. As a result, newer methods such as *DIGRA* [11] have been developed, which allow for data insertion and deletion. DIGRA uses what it calls a *lazy-weight mechanism* in order to reduce insertion costs, which operates similarly to a B-tree. After inserting a new datum to the index, DIGRA checks if any node of the index has more than B children. It there are such nodes, the children are split into a new sub-tree, otherwise we keep it as is. In the case of deletion, if any node has less than $B/2$ children, we merge sub-trees in order for each (non-leaf) node to have more than $B/2$ children. [6, 11]

In the following sections, we first examine some of the most commonly used algorithms that are used for performing RFANN searches: iRangeGraph, SeRF, FilteredVamana, StitchedVamana, Milvus and OptimizedPostfilering (along with its variant Superpostfiltering). In the following section we provide information on how they work, their implementations and existing studies comparing them, before showcasing our own implementation in Section 5. We present some basic information in *Table 1*.

Table 1: Overview of commonly used RFANN methods

Algorithm	**Type of algorithm**	**Type of index**	**Filtering method**	**Notes**
iRangeGraph [1]	Graph	RNG	Pre-filtering	
SeRF [2]	Graph	HNSW	In-filtering	Also known as 2DSegmentGraph
FilteredVamana [4]	Graph	Vamana	In-filtering	Designed for LFANN
StitchedVamana [4]	Graph	Vamana	In-filtering	Designed for LFANN
Milvus [5]	Graph/Quantization	IVF/HNSW	Pre-filtering	Vector data-management system, not a specific algorithm per se
OptimizedPostfiltering	Graph	β-WST	Post-filtering	

/SuperPostfiltering [3]				

4.2. iRangeGraph

iRangeGraph [1] is a graph-based method for performing RFANN queries. It is influenced by SeRF, which is a method that tries to create compressed graphs for every possible sub-range of the datasets while building its index (see Section 4.3).

However, instead of constructing a graph for every possible range, iRangeGraph uses a structure called a *segment tree* to construct graphs for only a specific amount of ranges when it builds its index (creating what the iRangeGraph's paper calls *elemental graphs*). Afterwards, when a query is sent to the index, the edges of the elemental graphs are used to create what is called a *dedicated graph*, a structure which is specifically made for the query's range and used to find the approximate nearest neighbors of the query vector.

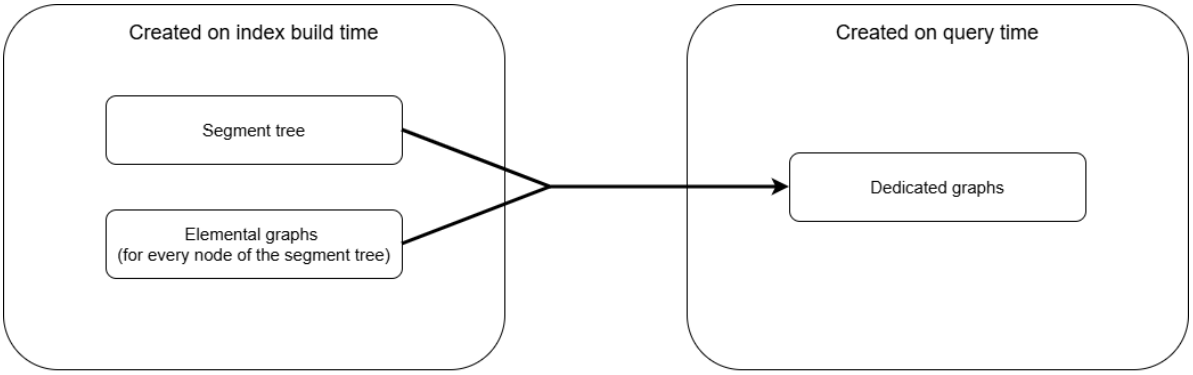


Figure 2: The structures created by iRangeGraph

4.2.1. Building the index – Creating the segment tree

iRangeGraph's index is based on the so-called *segment tree*. A segment tree is a balanced binary tree (that means the height between the left and right sub-trees is the same, or their heights differ by one) with a height of $O(\log(n))$, whose node represents a specific range (also called a *segment*). The root's range is defined to be the range of the entire dataset, while all M children of the root receive an equal (or almost equal) part of the root segment. We continue this process recursively (splitting a node's range equally amongst its children), until the segments cannot be split even further (i.e. when the segment's length is equal to 1).

Based on the nature of the segment tree, every data object ends up appearing only once on each layer of the tree. As a result, the space complexity of building an elemental graph for each segment in the tree is equal to $O(nMlog(n))$, which is far lower than the $O(n^3M)$ complexity needed if a graph was created for each possible range.[3]

[3] In the sentence above, n equals the number of nodes in the data, and M equals the max out-degree of a node in the segment tree (i.e. how many children can a node of the segment tree have).

The next step for building iRangeGraph's index is to create *elemental graphs* for all segments of the segment tree (the process of which we will cover below).

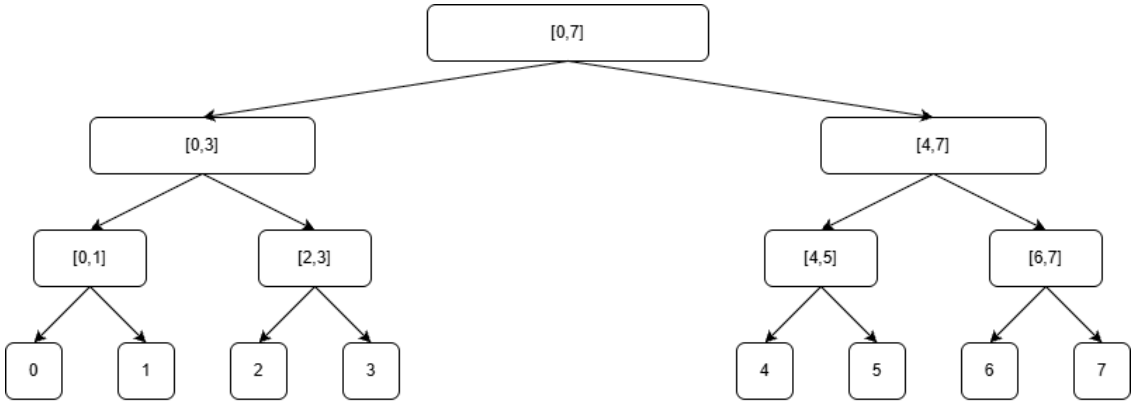


Figure 3: An example of a segment tree for the [0,7] range. iRangeGraph creates an elemental graph for each node of the segment tree.

4.2.2. Building the index – Creating the elemental graphs

The next task for building an index is creating the elemental graphs. Elemental graphs are graphs that store information about which data objects are approximate nearest neighbors of each other in a specific query range. As we have also mentioned above, they are then used to create dedicated graphs, which are used to respond to queries.

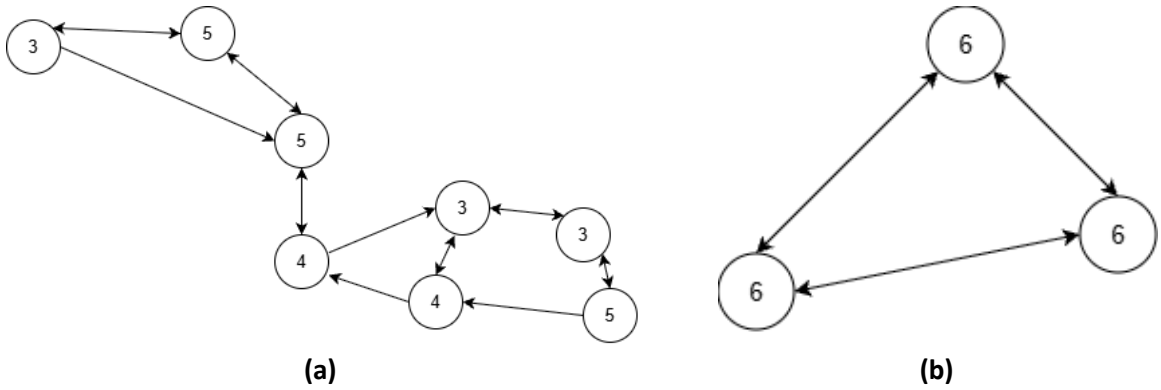


Figure 4: Two examples of elemental graphs. Figure (a) shows an elemental graph for the [3,5] range, while (b) shows the elemental graph for the [6,6]. Each node represents a data object, whose numerical value is represented with the number inside the node. An edge from node x to node y represents that y is an approximate nearest neighbor of x

Elemental graphs are created by a variety of methods such as *DiskANN*, *NSG* (*Navigating Spreading-out Graph*), and *HNSW* (*Hierarchical Navigable Small World*) graphs. They create a structure similar to a RNG (a Relative Neighborhood Graph), a common structure used in RFANN algorithms (as we have also mentioned in Section 4.1). RNG graphs are structures whose vertices represent data objects and whose edges are defined in the following

manner: an edge from u to v is included only if there is no point u' that dominates u (meaning that there is no point u' where $\delta(u,u') < \delta(u,v)$ and $\delta(v,u') < \delta(u,v)$)

Creating RNG graphs for n data objects has a computational complexity equal to $O(n^3)$, which can be prohibitively high for large datasets. As a result, most methods try to create an approximate RNG graph in order to reduce the amount of time needed to create them.

After all elemental graphs have been created, the index is ready to be used for performing queries.

4.2.3. Performing RFANN queries with iRangeGraph

The first step in performing a RFANN query (with a vector v and a numerical range $[l,r]$) with iRangeGraph is to find the segments whose values are in $[l,r]$. iRangeGraph then uses those segments to create a dedicated graph, a graph that contains all the data objects whose values are inside $[l,r]$. In essence the construction of a dedicated graph is to select which edges need to be kept from the elemental graphs.

In order to select the dedicated graph's edges, the following algorithm is performed from the root of the segment tree downwards: all in-range-edges (u,v) in the elemental graph of a layer are selected. In order to decrease the computational cost, the algorithm tries to only select edges from segments that overlap the most with the query range (for example, if a segment and its child both have the same intersection with the query range, the edge selection process in the parent segment can be skipped and the edges can be fetched from the child). The edge selection algorithm terminates after edges from range which are fully covered by the query range have been added.

The algorithm for selecting the dedicated graph's edges is presented below:

	Algorithm 1: Selecting edges of dedicated graphs (Adapted from [1]'s Algorithm 1)
	Input:
	u : data object
	$[l,r]$: query range
	M : maximum out-degree of elemental and dedicated graphs
	$N_{(layer,u)}$: neighbors of u in layer $layer$
	x : upper limit of numerical values in dataset
	Output:
	S : u's neighbors in the elemental graph
1	$start \leftarrow 0$
2	$end \leftarrow x$
3	$S \leftarrow \emptyset$
4	**while** $\lvert S \rvert < m$ **do**
5	$[start_{seg}, end_{seg}] \leftarrow$ child segment of $[start, end]$ containing u
6	**if** $[start_{seg}, end_{seg}] \cap [l,r] = [start, end] \cap [l,r]$ **then**

7	$start \leftarrow start_{seg}$
8	$end \leftarrow end_{seg}$
9	$layer \leftarrow layer + 1$
10	**else**
11	$S \leftarrow S \cup (N_{(layer,u)} \cap [l,r])$
12	Reduce $\|S\|$ to m
13	**if** $[start, end] \subseteq [l,r]$ **then**
14	**break loop**
15	**end if**
16	$start \leftarrow start_{seg}$
17	$end \leftarrow end_{seg}$
18	$layer \leftarrow layer + 1$
19	**end if**
20	**end while**
21	**return** S

Algorithm 1: iRangeGraph - Selecting edges of dedicated graphs

After the dedicated graph is created, iRangeGraph uses greedy beam search on the dedicated graph in order to find the approximate nearest neighbors.

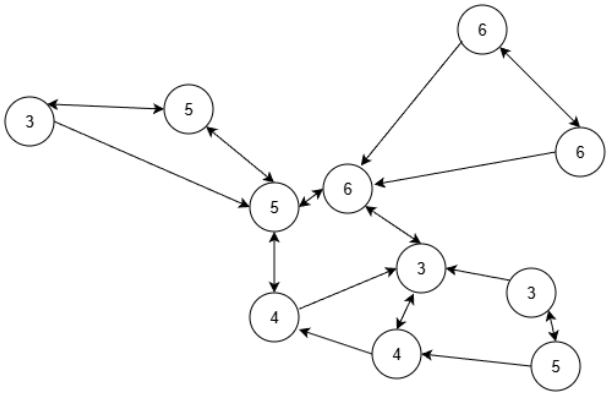


Figure 5: Example of a dedicated graph for the [4,6] range. Edges were also used from the [4,7] segment

Greedy beam search is an extension of the typical greedy search algorithm, where we decide to visit the b (where $b \neq 1$) most promising neighbors instead of evaluating all possible nodes. The algorithm starts with a specific entry point (generally defined by the elemental graph construction method). On every step, we iteratively expand the search to each current candidate's neighbors and we end up storing only the b best candidate nodes from the current candidate's neighbors. [12]

The greedy beam search algorithm terminates when the neighbors of the b nearest nodes that have already been visited are further away from the query vector than these already visited nodes.

4.2.4. Possible variants of iRangeGraph

Finally, a possible variation of iRangeGraph has been proposed by the paper's authors, which is named *iRangeGraph+*. During search, iRangeGraph+ can visit neighbors of a data point which are outside the query range with a probability of p. However, in order not to visit too many points outside the query range, the paper's authors recommended setting p to e^{-t}, where t is the number of consecutive out-of-range data objects visited by algorithm.

4.3. SeRF

Another well-known RFANN method is *SeRF* [2] (which stands for *Segment Graph for Range-Filtering Approximate Nearest Neighbor Search*). It is also a graph-based method for performing RFANN queries (as iRangeGraph is). Contrary to iRangeGraph, which only creates graphs for a subset of ranges, SeRF tries to build an index for each range, using what is called a segment graph (not to be confused by the segment tree used by iRangeGraph), which is a structure that stores HNSW (Hierarchical Navigable Small World) graphs (which themselves are used for Approximate Nearest Neighbor Search queries). A segment graph contains all attribute values lower than a specific threshold, compressing the indices that would be created if we had created an index for each possible range (albeit not losslessly). These segment graphs are then combined into a *2DSegmentGraph*[4], which compresses segment graphs.

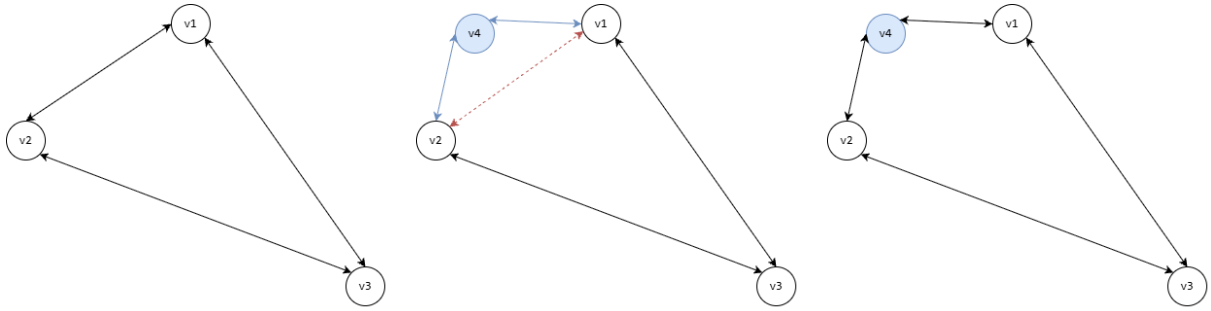
2DSegmentGraph has also proven itself to be a quite efficient and effective method for performing RFANN queries. It is a quite memory-efficient algorithm, as its memory footprint can be much smaller than iRangeGraph. However, 2DSegmentGraph is also slower and less accurate than iRangeGraph in most datasets it was tested on (something which we will get into more detail in Section 4.8) [1, 2]

4.3.1. Building the index

As we have mentioned before, 2DSegmentGraph's index is based on Hierarchical Navigable Small World graphs (also known as HNSW [9]). HNSW graphs can be considered as an extension of probabilistic skip lists, but instead of linked lists, they use proximity graphs. This means that HNSW graphs are hierarchical structures, which contain multiple layers in order to allow efficient movement between nodes with only few steps. Higher layers contain fewer nodes, while the lowest layer of the graph contains all of them. The connections between the nodes in the higher layers allow for faster connections (even between elements in the lower layers). HNSW graphs are probabilistic in the sense that a node can advance levels with a probability p. This allows HNSW graphs not to rearrange their layers upon either insertion or deletion of elements (which can be costly). A HNSW graph takes $O(nM)$ space (where n is the number of nodes, and M is the number of edges). This means that storing HNSW graphs for all possible ranges has a space complexity of $O(n^2M)$, which is prohibitive. As a result, SeRF compresses the segment graphs by pruning unnecessary edges (i.e. removing dominated edges, meaning it uses the same criterion as iRangeGraph uses to remove edges). This compression reduces the space needed to $O(nM)$. For this compression, a structure known as segment graph is used.

[4] The term 2DSegmentgraph is sometimes used for the algorithm as a whole [1]

Segment graphs contain information about data objects, their M nearest neighbors, and a *timestamp range* (which signifies from which right bounds two data objects are neighbors). They are used for half-bounded range queries (that is queries where either the lower or the upper limit equals the lower or upper limit of the domain).



(a) **(b)** **(c)**

Figure 6: Snapshot from the construction of a segment graph: As we can see, a new data object ($v4$) has been inserted to the segment graph. Since it is closer both to $v2$ and $v1$ than they are to each other (i.e. it dominates them), the edge between $v1$ and $v2$ is pruned.

In order to build the segment graph, SeRF first tries to find the approximate nearest neighbors for every value inside the half-bounded range. It then combines these nodes into a graph whose edges that contain the information mentioned above (that is the $[b, e]$ timestamp range).

In order to support queries with arbitrary limits (i.e. queries both range limits do not correspond with the limits of the dataset), n segment graphs will be necessary (each for every value in the query range). The space complexity for these segment graphs would be equal to $O(n^2M)$, which is also considered prohibitive. As a result, another structure, called *2DSegmentGraph*, is proposed. 2DSegmentGraph compresses the segment graphs by adding information about which nodes are neighbors for a specific left bound range (adding a $[l, r]$ range to the edges, which signifies the range of the other bound where two nodes are neighbors).

The algorithm for creating the 2DSegmentGraph is presented below:

	Algorithm 2: 2DSegmentGraphConstruction (Adapted from [2]'s Algorithm 8)
	Input:
	D : dataset
	K : number of nearest neighbors
	M : maximum degree
	Output:
	G : two-dimensional segment graph
1	**for** $j = 2$ to n **do**
2	**for** $i = 1$ to $j - 1$ **do**

3	$anns \leftarrow$ 2DSegmentANNSearch(G, v_j, $[i, j-1]$, v_i, K)
4	$i' \leftarrow min\{x \mid u_x \in anns\}$
5	**for** each $u \in$ Prune(v_j, $anns$, M) **do**
6	add edge (i, i', v, j, n) to $G[v_j]$
7	add edge (i, i', v_j, j, n) to $G[v]$
8	**end for**
9	**end for**
10	**end for**

Algorithm 2: SeRF – Construction of a 2DSegmentGraph

The *Prune* method mentioned in this algorithm is used to remove neighbor candidates that are dominated by other neighbor nodes (the definition of domination is the same that iRangeGraph and other methods use)

4.3.2. Querying with 2DSegmentGraph

After building the 2DSegmentGraph, the following algorithm is used for querying:

	Algorithm 3: Performing a query with 2DSegmentGraph (Adapted from [2]'s Algorithm 7)
	Input:
	G : 2DSegmentGraph
	q : query vector
	$[x,y]$: maximum degree
	ep : entry point
	Output:
	$annd$: K approximate nearest neighbors of q, whose values are in $[x,y]$
1	mark ep as visites
2	push ep to min-heap $unvisited\_nodes$ (ordered by distance to q)
3	push ep to max-heap $anns$ (ordered by distance to q)
4	**while** $unvisited\_nodes$ is not empty do
5	$v \leftarrow$ GetClosestTo($unvisited\_nodes$, q)
6	$u \leftarrow$ GetFarthestFrom($anns$, q)
7	**if** $\delta(q,v) > \delta(q,u)$ **then**
8	**break;**
9	**end if**
10	**for** each $unvisited\_node$ with $(l,r,node,b,e)$ in $G[v]$ **do**
11	**if** $x \in [l,r]$ **and** $y \in [b,e]$ **then**
12	mark $node$ as $visited$
13	$u \leftarrow$ GetFarthestFrom($anns$, q)
14	**if** $\lvert anns \rvert < K$ **or** $\delta(q,node) < \delta(q,u)$ **then**
15	push $node$ to $unvisited\_nodes$, $anns$
16	**if** $\lvert anns \rvert > K$ **then**
17	pop $anns$
18	**end if**

19	**end if**
20	**end if**
21	**end for**
22	**end while**
23	**return** $anns$

Algorithm 3: SeRF – Performing a query with 2DSegmentGraph

SeRF's querying algorithm starts by traversing the tree from a specified entry point. Then, it traverses the graph in order to find the approximate nearest neighbor nodes (i.e. nodes who can be considered neighbors in the range $[x, y]$ – which is the query range – and are closer to q (our query vector) than the current approximate nearest neighbors). The search stops when either all nodes are visited or if no suitable candidate nodes are left (because all approximate nearest neighbors have already been found).

4.4. FilteredVamana / StitchedVamana

Two other methods often used for RFANN are *FilteredVamana* and *StitchedVamana* [4]. Although they were created for filtering labeled data [4], they can also be used for range-filtered queries. As their name suggests, both methods are based on the Vamana Graph [8]. Vamana graphs work by creating so-called navigable paths, which try to conduct a greedy search towards possible neighbor candidates.

FilteredVamana starts with an empty graph and adds data points to its index in an incremental fashion. For every point x with a specific set of labels, a set of candidate neighbors is found, adding bi-directional edges between x and the candidates. If the vertex degree of a node ends up exceeding a given threshold, a *RobustPrune* procedure is run in order to prune redundant edges.

On the other hand, the StitchedVamana method takes a bulk-indexing approach. For every label filter, a Vamana index is created over each point associated with it. Then all indices are overlaid into a graph whose edges are the union of the edges in the filter-specific graphs. Then, the RobustPrune procedure is run on every node with a degree higher than a specified threshold.

The Vamana methods can prove themselves better than methods based on IVF indices and HNSW graphs (such as Milvus) in metrics such as specificity and recall [4]. Moreover, they can also be quite efficient compared to methods like iRangeGraph and 2DSegmentGraph [1]. However, since they were created for querying label data, using those methods from RFANN might be quite difficult in many cases.

4.4.1. FilteredVamana

An idea that is basic to understanding how FilteredVamana creates its index is *Filtered Greedy Search* (or *FilteredGreedySearch*). Filtered Greedy Search is used in order to find the approximate nearest neighbors of a specific point. It starts on the query labels' start nodes (the nodes from which the search will start). Then the algorithm uses an L-size priority queue to store the most promising neighbor candidates. For each node in the dataset,

FilteredGreedySearch searches for the nearest unvisited neighbor node, and adds that node's neighbors in L if their label is inside in query's label set. If the queue grows larger than L, the queue is trimmed until its size goes back to L. This procedure stops when all nodes of the dataset are visited.

Another concept central to FilteredVamana (but also StitchedVamana) is the so-called *FilteredRobustPrune*. FilteredRobustPrune is an algorithm run in order to remove edges from a node if its out-degree exceeds a specific threshold (which is denoted as M). It uses a similar (but not the exact same) algorithm as both iRangeGraph and SeRF use (pruning dominated nodes), although FilteredRobustPrune also adds a parameter a (which allows the retention of some slightly dominated nodes).

Below we present the steps of FilteredRobustPrune:

	Algorithm 4: FilteredRobustPrune (Adapted from [4]'s Algorithm 3)
	Input:
	G : graph
	p : data point
	V : candidate set
	a : distance threshold
	M : max out-degree
	Output:
	Modification of graph G
1	$V \leftarrow V \cup N_{out}(p)\{p\}$
2	$\mathrm{N_{out}(p)} \leftarrow \emptyset$
3	**while** $V \neq \emptyset$ **do**
4	$closest \leftarrow$ GetClosestTo(V, p)
5	$N_{out}(p) \leftarrow N_{out}(p) \cup closest$
6	**if** $\lvert N_{out}(p) \rvert = M$ **then**
7	**break**
8	**end if**
9	**for** $node \in V$ **do**
10	**if (**Labels$(node) \cap$ Labels$(p)) \not\subset$ Labels$(closest)$ **then**
11	**continue**
12	**end if**
13	**if** $a * d(closest, node) \leq d(p, node)$ **then**
14	remove node from V
15	**end if**
16	**end for**
17	**end while**

Algorithm 4: FilteredVamana – The FilteredRobustPrune process

The first step in creating the FilteredVamana index is to select a *start node* for each data label. A start node must have that specific data label as its value, and that no single point should be the start node from too many labels.

Then, starting from the start node, and for each node of the dataset, FilteredGreedySearch is run, which returns a set of vectors that we have visited. The node set is then pruned in order not to allow the inserted node x to have more than M out-neighbors. If need be, FilteredRobustPrune is also run on the neighbors of x, if they have more than M out-neighbors.

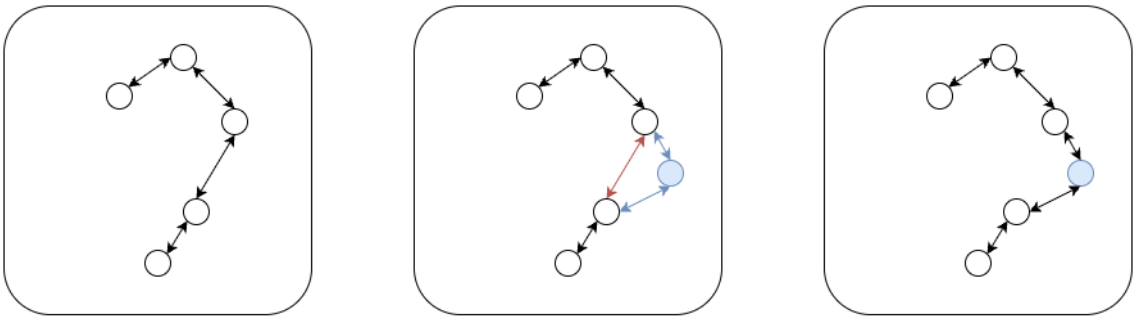


Figure 7: Inserting a node to a FilteredVamana index

In the table below we show the algorithm that Filtered Vamana uses to create its index:

	Algorithm 5: FilteredVamana Indexing Algorithm (Adapted from [4]'s Algorithm 4)
	Input:
	P : dataset
	a : distance threshold
	L : max number of candidates to note
	M : max out-degree
	Output:
	G : directed graph
1	$G \leftarrow empty\_graph$
2	$s \leftarrow$ Medoid(P)
3	shuffle elements of P
4	**for** $i \in P$ **do**
5	$start\_nodes \leftarrow start\_nodes \forall filter\_label$
6	$unvisted\_nodes \leftarrow$ FilteredGreedySearch$(start_nodes, i, 0, L, labels_x)$
7	$V \leftarrow V \cup unvisited\_nodes$
8	run FilteredRobustPrune $(G, node, V, a, M)$ to prune neighbors of $node$
9	**for** $j \in N_{out}(i)$ **do**
10	$N_{out}(j) \leftarrow N_{out}(j) \cup i$
11	**if then**
12	run FilteredRobustPrune$(G, j, N_{out}(j), a, M)$ to prune neighbors of j
13	**end if**
14	**end for**
15	**end for**

Algorithm 5: FilteredVamana – FilteredVamana's index construction algorithm

4.4.2. StitchedVamana

StitchedVamana is another RFANN method which is based on the Vamana method. Contrary to FilteredVamana, it creates separate sub-graphs, each containing all points inside a specific label group, and then unites (or "stitches") all the sub-graphs to create the resulting index. In order for StitchedVamana to create its index, the point data (i.e. all points containing a specific label value) have to be defined beforehand. The results of both the Filtered-DiskANN study [4], but also other studies [1, 6] prove that StitchedVamana is more efficient than FilteredVamana, even though StitchedVamana's index takes more time to build compared to FilteredVamana's.

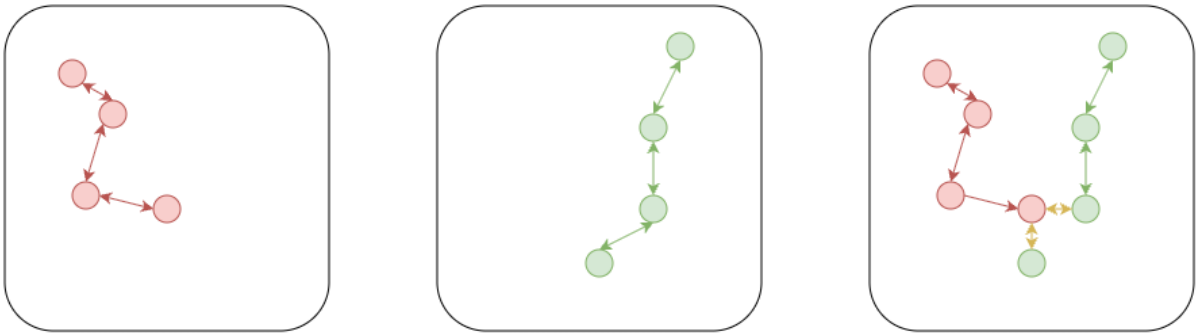


Figure 8: Snapshot from the creation of a StitchedVamana index. The indexing algorithm creates graphs for each label group and then "stitches" them together.

StitchedVamana uses the same algorithm for performing queries as FilteredVamana. Their only major difference is on how each method builds its index.

The algorithm for creating StitchedVamana's index is the following:

	Algorithm 6: StitchedVamana Indexing Algorithm (Adapted from [4]'s Algorithm 5)
	Input:
	P : dataset
	F : set of labels
	a : distance threshold
	L : max number of candidates to note
	$M_{stitched}$: max out-degree for stitching
	M: max out-degree for Vamana
	Output:
	G : directed graph
1	$G \leftarrow empty\_graph$
2	**for** $label \in F$ **do**
3	$P_{label} \leftarrow$ set of points with label equal to $label$
4	$G_{label} \leftarrow$ Vamana(P_{label}, a, L, M)
5	**end for**

6	**for** $vertice \in G$ **do**
7	FilteredRobustPrune($vertice, N_{out}(vertice), a, M_{stitched}$)
8	**end for**

Algorithm 6: StitchedVamana – StitchedVamana's index construction algorithm

In order to create the index faster, L and M need to be smaller than the parameters used in FilteredVamana [4].

4.5. Milvus

Another system that can be used for RFANN, (even though it was not specifically designed for it) is *Milvus* [5]. Milvus is a vector data-management system (not just an algorithm per se) with functionalities such as attribute filtering (filtering by non-vector attributes that are included in the data object) and multi-vector queries (where each data object contains more than one vector).

A concept central to Milvus is the concept of *entity*. An entity consists of one or multiple vectors and optionally some numerical non-vector data. It can be considered as a more generic form of the data objects used in the previous methods. This structure allows Milvus to perform three different kinds of queries: *vector queries* (Getting the k closest vectors, without considering the values of any other attributes), *attribute-filtered queries* (RFANN queries) and multi-vector queries (queries where each entity consists of multiple vectors and the goal is to find the k most similar entities based on the result of an aggregation function - for example the weighted sum- between multiple vectors).

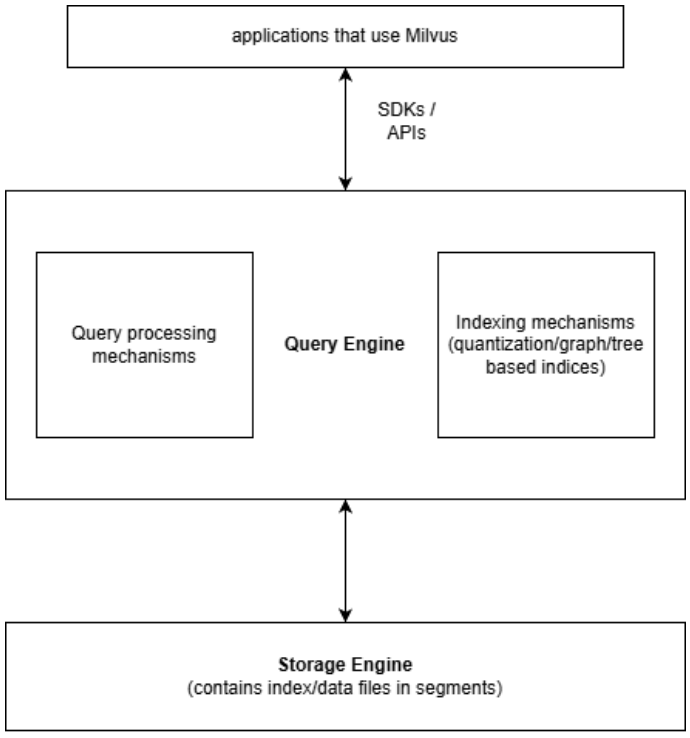


Figure 9: Basic components of Milvus

When building RFANN indices with Milvus, two types of indices are most commonly used: quantization-based indices (such as *IVF_FLAT*, *IVF_SQ8* and *IVF_PQ*) [13] and graph-based indices such as HNSW (which was already mentioned in the 2DSegmentGraph section).

Milvus' quantization-based indices are based on the Inverted File indices (or IVF indexes, a structure we have already mentioned in Section 4.1). IVF indices use two quantizers: the coarse quantizer (which applies the K-means algorithm to divide the vectors into buckets), and the fine quantizer (which encodes the data inside each bucket). Generally speaking, the coarse quantizer tends to be the same across different types of IVG indices, while it is the fine quantizer that is different between indexing methods. Milvus provides a number of quantization methods which can help compress vector data by a large margin. The most commonly used methods are IVF_FLAT, IVF_SQ8 and IVF_PQ.

IVF_FLAT does not encode the vector data, instead storing them in their original form [14]. On the other hand, IVF_SQ8 uses a scalar quantizer to compress vectors in order to reduce memory. The vectors are compressed by first normalizing their values to a range between 0 and 1, afterwards multiplying the normalized values with 255 and rounding the result into the nearest integer. This method allows the vector values to be represented as 1-byte integers (instead of storing them as 4-byte integers or 4-8 byte floats) [15]. Finally, IVF_PQ uses a product quantizer which first decomposes high-dimensional vectors into smaller sub-vectors, thus separating the dataset into subspaces. Afterwards, the product quantizer performs k-means clustering to find the centroids for each subspace, and then links each sub-vector with its corresponding centroid (and only storing that instead of the sub-vector's coordinates). As a result, each vector is represented as a set of PQ codes, which represent the centroids of each sub-vector [16].

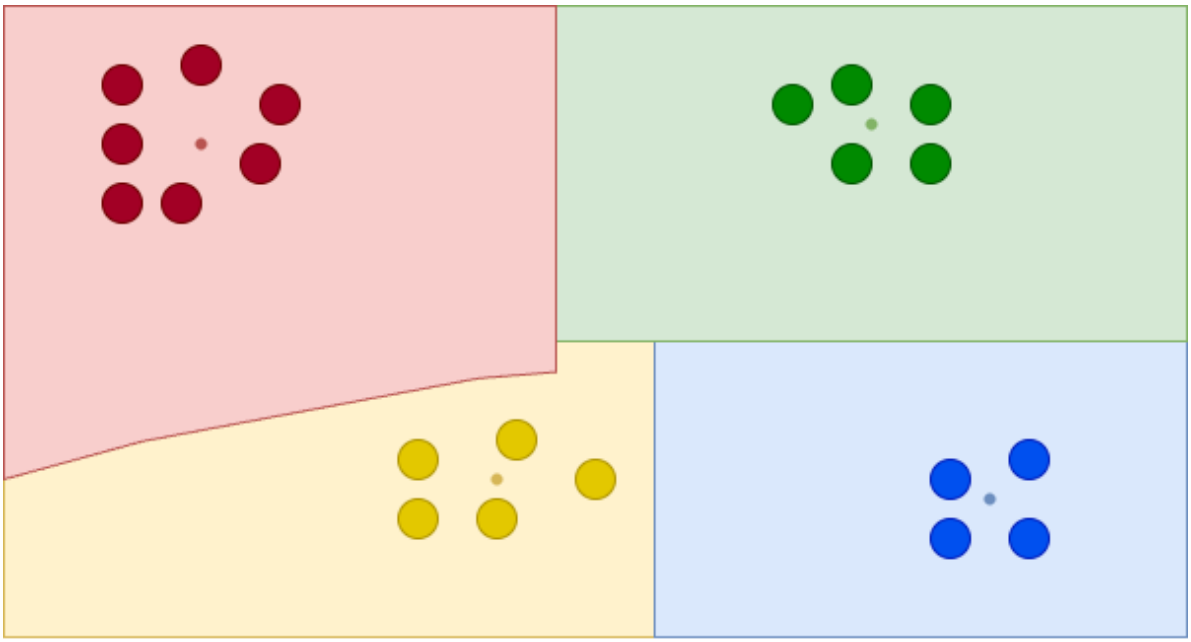


Figure 10: Example of IVF partitions. Milvus uses methods such as k-means clustering in order to divide the dataset into clusters (which Milvus calls "partitions"). Different IVF indexing methods then perform different operations of the partitions to compress the vectors either in memory or storage.

Another advantage of Milvus is its ability to efficiently add or remove data from its structure, since it uses *LSM-trees* (*Log Structured Merge Trees*). LSM-trees are commonly used in many NoSQL database systems, due to their efficiency in writing data, their storage space efficiency and ability to recover data [17].

When adding entries to a LSM-tree, they are first stored in memory, inside what is called a *MemTable*. A MemTable is an in-memory structure that stores newly added objects, typically implemented as a balanced tree or a skip-list [18]. When a MemTable grows large enough (or once per second), it is stored to the LSM-tree as an immutable segment called *SortedString table* (or *SSTable*) [17]. Milvus tries to merge disk segments similar in size to one another, in order to create segments of a specified size (e.g., 1 GB).

Milvus proves itself as a quite capable general-purpose vector data-management system, however more specialized methods such as iRangeGraph and SeRF produce more accurate results (even though they might need more time and memory) [1].

4.6. OptimizedPostfiltering/SuperPostfiltering

The final methods we cover are OptimizedPostfiltering and SuperPostfiltering [3], which are both graph-based query methods. They are designed for RFANN queries (which are referred to as Window Search Queries in this paper).

A structure named *β-Window Search Tree* (or *β-WST* for short) is the basis of both OptimizedPostfiltering and SuperPostfiltering, since it is the index both methods use to perform RFANN queries. It is a structure quite similar to the segment trees used by iRangeGraph, although β-WSTs are not necessarily binary trees. Similarly to segment trees, every node of a β-WST contains an ANN graph for all data objects inside the node's range.

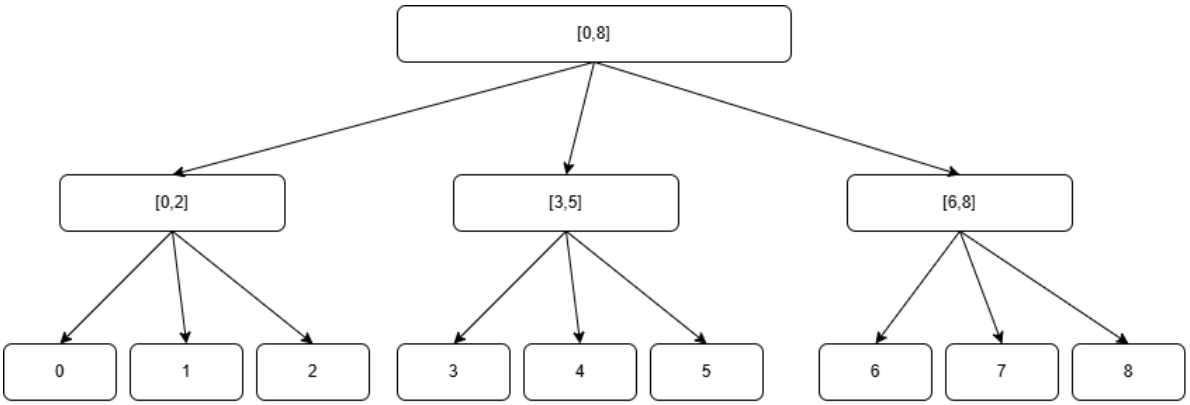


Figure 11: Example of a β-WST tree with β equal to 3 for the [0,8]

In order to create a β-WST, its building algorithms first checks if the dataset's size is smaller than a certain threshold (which is typically set to be the number of partitions). If that is the case, then the algorithm concludes and no tree is created. Otherwise, the process proceeds, arbitrarily splitting the dataset into N partitions (which are generally similar in size, except the last one). Then, for every partition, the algorithm creates trees in a recursive manner,

using the same process until the size of a partition is lesser than the threshold (which then terminates the algorithm).

In the next page we present in detail the algorithm for the creation of a β-WST:

	Algorithm 7: Creating a β-WST tree (Adapted from [3]'s Algorithm 1)
	Input:
	D : dataset sorted by value
	β : number of partitions
	a : distance threshold
	L : max number of candidates to note
	$M_{stitched}$: max out-degree for stitching
	M: max out-degree for Vamana
	Output:
	Window Search Tree of D
1	**if** $\lvert D\rvert < \beta$ **then**
2	**return** (null,null,D)
3	**end if**
4	$partitions \leftarrow$ Partition(D, β)
5	$sizes \leftarrow$ [size($partition$) **for** $partition \in parititons$]
6	$children \leftarrow \beta$-sized empty array
7	**for** i in $[0, N)$ **do** in parallel
8	$children[i] \leftarrow$ BuildB-WstTree($parititons[i], \beta, partitions[i], l$)
9	**end for**
10	**return** ($index$, ($children$, $sizes$), D)

Algorithm 7: OptimizedPostfiltering/SuperPostfiltering – Creation of a β-WST tree

In order to perform queries, SuperPostfiltering tries to find the smallest subset of the dataset whose points are in the desired range. This is done by finding the smallest possible sub-tree of the index whose sub-nodes have values that are inside the range. It is a variation of another method mentioned in this paper named *OptimizedPostfiltering* [3], but contrary to it, SuperPostfiltering runs on an arbitrary-defined subset of D (our dataset).

Finally, we present the OptimizedPostfiltering and SuperPostfiltering algorithms:

	Algorithm 8: OptimizedPostfiltering (Adapted from [3]'s Algorithm 3)
	Input:
	T : β-WST = ($index$, ($children$, $sizers$), D)
	q: query vector
	(a, b): range filter
	Output:
	$anns$: approximate nearest neighbours (if they exist)

1	$index \leftarrow$ smallest index of tree containing all data points with values in (a, b)
2	$k \leftarrow 1$
3	**while** $k < N$ **do**
4	$unfiltered \leftarrow \text{Index}(q, k)$
5	$candidates \leftarrow \{point \in unfiltered \mid l(point) \in (a, b)\}$
6	**if** $candidates \neq \emptyset$ **then**
7	**return** $argmin_{y \in candidates \cap subset} \ \delta(v, q)$
8	**end if**
9	$k \leftarrow 2 * k$
10	**end while**

Algorithm 8: OptimizedPostfiltering/SuperPostfiltering - Performing queries with OptimizedPostfiltering

	Algorithm 9: SuperPostfiltering (Adapted from [3]'s Algorithm 3)
	Input:
	T : β-WST = $(index, (children, sizers), subset \subset D)$
	q: query vector
	(a, b): range filter
	Output:
	$anns$: approximate nearest neighbours (if they exist)
1	$index \leftarrow$ smallest index of tree containing all data points with values in (a, b)
2	$k \leftarrow 1$
3	**while** $k < N$ **do**
4	$unfiltered \leftarrow \text{Index}(q, k)$
5	$candidates \leftarrow \{point \in unfiltered \mid l(point) \in (a, b)\}$
5	**if** $candidates \neq \emptyset$ **then**
6	**return** $argmin_{y \in candidates \cap subset} \ \delta(v, q)$
7	**end if**
8	$k \leftarrow 2 * k$
9	**end while**
10	**return** $(index, (children, sizes), D)$

Algorithm 9: OptimizedPostfiltering/SuperPostfiltering – Performing queries with SuperPostfiltering

4.7. Existing experimental studies and their results

There exists recent literature that tries to compare various methods used for RFANN search. Most papers that present RFANN search algorithms [1, 2, 3, 4] compare their methods with existing ones in order to showcase the capabilities of their proposed methods. Meanwhile, benchmarking studies such as [6] have been conducted, which compare various RFANN and LFANN search methods in various scenarios (such as different datasets or selectivity levels).

The two most complete experimental studies are the experiments conducted to compare iRangeGraph with other methods in the iRangeGraph paper [1] and the recent paper from Li, Yen, Lu, Zhang, Cheng, and Ma called "Attribute Filtering in Approximate Nearest Neighbor Search: An In-depth Experimental Study"[5], which manages to compare various RFANN algorithms in different scenarios.

Most studies [1, 2, 3, 4, 6] tend to use the following metrics in order to evaluate the performance of a RFANN method (or a slight variation of them):

- *QPS* (or queries per second): the number of queries responded per second
- *Recall*: It is defined as $\frac{|TP|}{|TP|+|FP|}$, where TP signifies true positives and FN signifies false negatives. For our context, it is defined by $\frac{|G \cap S|}{K}$ where G signifies the correct nearest neighbors (the *ground truth*), S signifies the points returned by a specific method and K equals the request number of approximate nearest neighbors.
- *Memory usage*: How much space does a method need in memory. Although efficiency might not be critical for smaller datasets, it ends up being necessary for very large datasets (or spaces where space is limited)[6]
- *Index building time*: How much time does an index take to be created (in seconds).

iRangeGraph's study compares its namesake method with the rest of the methods mentioned in the previous sections (minus *OptimizedPostfiltering*), while SuperPostfiltering is missing in the experimental study paper, where its variant *OptimizedPostfiltering* was tested instead.

Both papers used a variety of datasets such as *RedCaps* (a dataset of image-text data pairs collected from Reddit [19]), *SIFT* (image embeddings) [20], *SpaceV* (a dataset created by Microsoft, commonly used for ANN queries and benchmarks [6]), and *Tripclick* (logs from the Trip Database health search engine [21]), among others.

In both papers [1, 6], iRangeGraph proves to be a very effective method, executing a quite high amount of QPS, while also preserving a particularly good level of recall. However, iRangeGraph takes a larger amount of time to create its index and needs more memory space to do so compared to its competitors (the only method that uses more memory space is SuperPostfiltering/OptimizedPostfiltering).

SeRF struggles with scenarios when selectivity is very low, returning less accurate results than other methods [6]. Due to its heavy compression, it manages to use a smaller amount of memory than iRangeGraph and SuperPostfiltering/OptimizedPostfiltering (its memory footprint being comparable to the compact FilteredVamana and StitchedVamana) [1, 2]. As for SeRF's index build time, results are inconclusive and contradictory between the two papers (in iRangeGraph's paper, SeRF's index build time is the highest out of all the aforementioned methods [1], while the opposite stands true in the experimental study [6]).

[5] I would personally want to thank the team behind [6] for releasing their source code on GitHub. It helped the author when he performed his own benchmarks.

[6] Some studies tend to subtract the size of the vectors, while others (such as this one) do not.

In both studies, FilteredVamana performs rather poorly in scenarios where selectivity was very low, while the opposite stands true for StitchedVamana which performs quite well on such scenarios [6]. Even though those methods were not originally created for RFANN queries, they might be useful in certain cases, especially due to their compact nature.

Milvus also performs rather consistently, showing a moderate amount of QPS and an acceptable level of recall on both possible index types (HNSW and IVF) [1, 6]. Its index build time is somewhat higher than the Vamana methods, but it does not take nearly as much time as iRangeGraph or SuperPostfiltering/OptimizedPostfiltering. Unfortunately, we cannot have a clear picture of its memory needs since it was tested on a containerized environment in the experimental study [6].

Finally, SuperPostfiltering (or its variant OptimizedPostfiltering) exhibits a consistently good performance (generally speaking even better than Milvus) [1, 6], and in some scenarios even surpassing iRangeGraph. However, the very high memory needs and comparatively large amount of time of time needed to build their indices might cause problems in some specific scenarios (for example when the size of the dataset is quite high compared to available memory).

5. Design and Implementation of BLC-RFANN

In this section we will get into more detail about the BLC-RFANN (*B+ tree with Lazy Caching for RFANN*), the B+ Tree based method which we propose.

As stated and shown in Section 4, most modern RFANN methods are graph-based. However, we chose a radically different approach for several reasons. First, we want to propose a method that can work for dynamic datasets (i.e. for datasets whose elements change over time). Second, we want to showcase our caching mechanism and test its effectiveness. Finally, we wish to benchmark how a simple and explainable baseline would compare with other methods.

Our method consists of two basic parts: a B+ tree (Section 5.1) and a caching system (Section 5.2). Both components perform their own roles in this solution: the B+ tree is the workhorse, while the cache, with its many features, increases overall performance by increasing the number of queries responded per second (at a rather small cost in memory).

In the following sections, we will get into more detail about the main components of this proposed algorithm and aspects of how it performs RFANN queries.

5.1. B+ Tree

This first component of our implementation is the B+ tree. It is a structure based on the B-tree [22, 23], which is considered to be a search tree (i.e. a tree that can be used for retrieving items from a dataset) [22].

More specifically, a B+ tree is a N-*ary* tree (meaning that every node has no more than N children) where all data in stored in leaf nodes (as opposed to also storing them on internal nodes, as B-trees do), and a tree where leaves are connected through links (even though they might not have the same parents) [22, 23].

It is a data structure that has been used for more than five decades [24]. Its primary advantage is its ability to efficiently store and retrieving data in storage [23], so it has been frequently used in file system indexing [25] and database indices (for example SQLite [26], Microsoft SQL Server [23] and even NoSQL systems such as CouchDB [27]).

In regards to BLC-RFANN, we use the B+ tree to store the numerical values of all objects in the dataset and to store all data objects with a specific numerical value.

5.1.1. Structure of a B+ tree

As most trees do, a B+ tree consists of a root, internal nodes and leaves. In a B+ tree, every non-leaf node except the root needs to have between $N/2$ and N children (but even the root should have no more than N children under it) [23]

Similarly to binary and B-trees, it is a balanced tree, meaning that all its sub-trees have a specific height (or almost the same height) [22, 23]. This attribute of the B+ tree allows for easier search through the tree, and a consistent height of about $\log_N M$ (where N is the order of a tree, and M is equal to the number to the elements inside the B+ tree). The order of a B+ tree signifies the maximum amount of children a node can have.

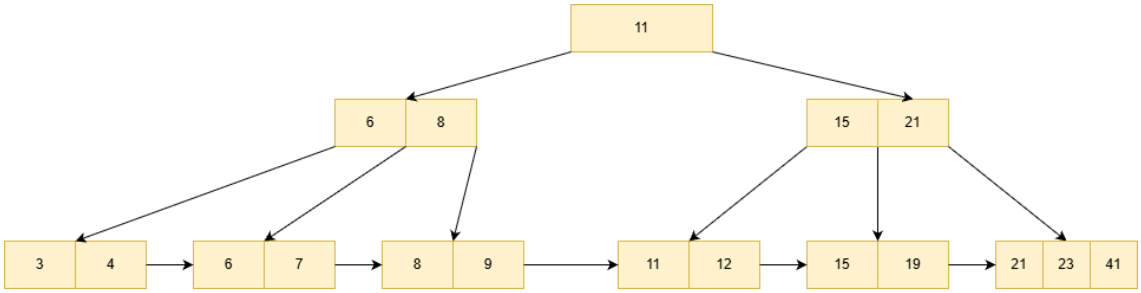


Figure 12: Example of a B+ tree whose order is equal to 4

As we can see from the figure above, under each leaf level of a B+ tree, data is stored. Each node is linked with each other, in order to allow ranged queries by traversing through the leaves whose values are inside the range.

When trying to perform a query for a specific value, we start at the root and we compare our query value with the keys which are contained inside the root node. We then select the appropriate sub-tree to follow, and then we iteratively continue this process until we reach a leaf node. Finally, we then perform a local search within the target leaf to find the datum with value equal to our query value.

This process is almost the same for performing range queries. However, after reaching a child node, we then scan all the data from the leaves whose values are inside our range using the linked list between the leaves, thus collecting the results of the query [22].

5.1.2. Adding elements to a B+ tree

An interesting feature of a B+ tree is its ability to balance itself (i.e. that all its sub-trees manage to have the same or almost the same height) when insertions or deletions occur.

When inserting a new element in B+ tree, we first try to find the leaf node where it should reside. This process is the same as the lookup operation when we try to perform to perform a query. After reaching the leaf, we need to check if the leaf has capacity (i.e. if the leaf contains less than N data keys). If it has, the insertion is complete. [23, 22]

However, if the leaf does not have enough capacity (i.e. it contains N data keys), we split the leaf into two different nodes (right and left), each taking about half of the data keys. We also update the pointers on the keys, in order to maintain the linked-list functionality between the keys. Afterwards, we take the smallest key of the newly created right node and we promote it to the parent node (however we also still keep the key on the leaf level, contrary to B-trees). If the parent node also has N children, we also perform the procedure mentioned above recursively across the tree.

If the aforementioned procedure ends up reaching the root, and the root also has N children, we separate it as well, and we create a new root which has a single separation key (the smaller key of the right node which is created as a result of the split).

An example of an insertion can be shown on the figure below:

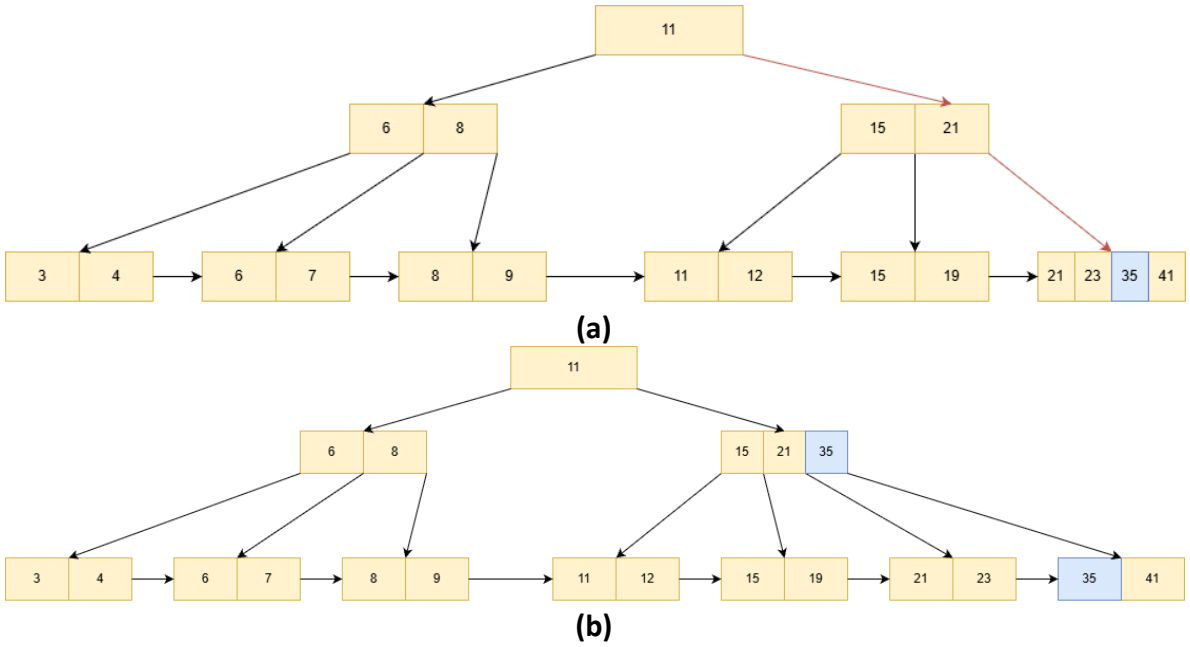


Figure 13: Insertion of a new element to a B+ tree (whose value is 35).The tree is rebalanced because the leaf where the element was originally stored had more values than allowed

5.1.3. Removing elements from a B+ tree

Removing elements from a B+ tree follows a process quite similar to insertions. We start by traversing the tree in order to find the leaf where the value to be deleted is stored. We then remove the key from the leaf. If the leaf ends up having more than $N/2$ nodes, the process ends here. [23, 22]

In the case that the leaf node has less than $N/2$ keys, we first try to borrow surplus nodes from the adjacent sibling leaves. If any adjacent siblings have surplus keys, we give them to the leaf which has less, and we redefine the keys in the parent to reflect the new boundary between the leaves.

However, if redistribution is not possible (i.e. no sibling node has surplus keys) we then need to merge leaves, combining it with an adjacent sibling and removing the separator key from the parent node. If the parent node also ends up having less than $N/2$ children, we also recursively perform the process mentioned upwards (until there is no need for either redistribution or merging).

In the case that the root ends up having exactly one child, the child becomes the new root becomes the new root of the B+ tree.

We show an example of a deletion from a B+ tree on the figure below:

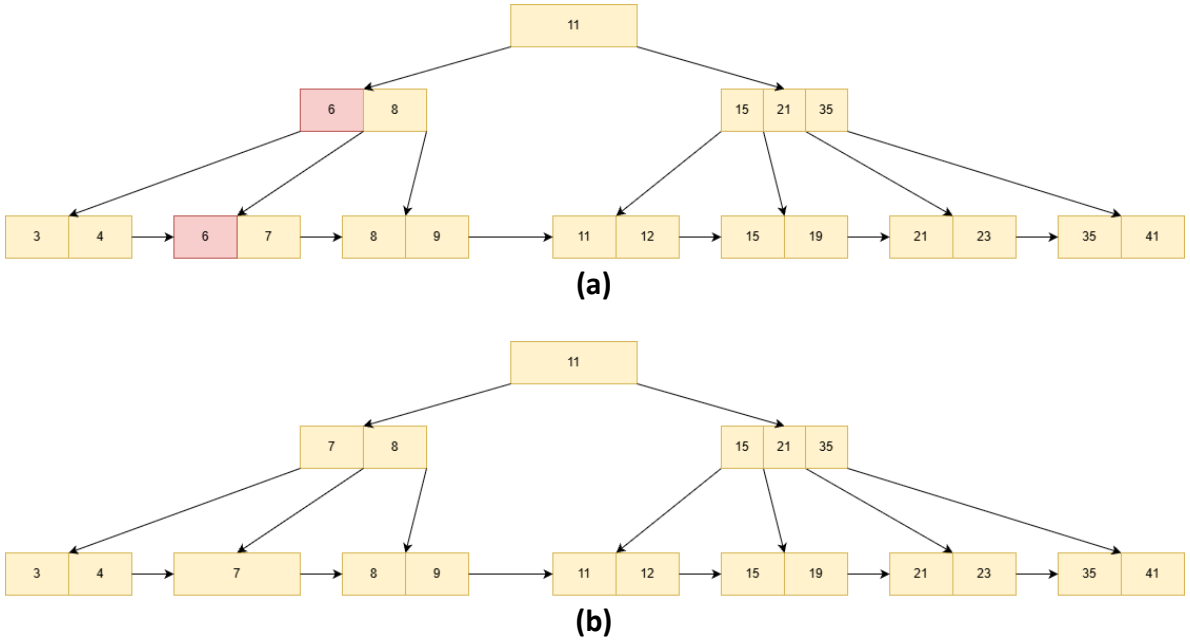


Figure 14: Removal of an object from the B+ tree. Since there was also an internal node with value equal to 6, we also remove 6 from the non-leaf nodes of the tree.

5.2. Lazy Caching mechanism

The second element of BLC-RFANN is its caching mechanism, which allows it to store already-performed queries, and then retrieve them when the need arises. This allows our solution to perform queries faster, instead of using the B+ tree to perform them again. BLC-RFANN's caching mechanism consists of various intricacies in order to both reduce query response time, but also to maintain a very high level of accuracy and recall.

It is called *lazy* due to the fact that we store query results after the query has been executed, instead of doing it beforehand. [28]

5.2.1. Structure Overview

The basic components of the caching mechanism are: (a) an interval tree that maps the various query ranges that have been answered in the past with query IDs (i.e. a query ID is a hash of its query vector and its query range) and (b) a directory which stores binary cache files, each file containing data from a specific previously answered query (such as the results, the query input and other metadata).

An interval tree is a commonly used data structure and a variation of the *red-black tree* that supports operations on numerical ranges [29]. Each node of the tree contains a specific

interval and a max value (which is the maximum value of either the range the node stores inside itself or the maximum value in the ranges of any of its descendants). It is a structure that

Due to being a variation of the red-black tree (which itself is a self-balanced binary tree) [29], an interval tree manages to remain balanced even when we insert or delete elements from it, since it performs rotations or recoloring in order to maintain a balanced height (fulfilling the red-black tree's coloring requirements).

Interval trees allow for both overlap searches (where we try to find intervals which overlap with a specific input range) and exact searches (where we try to find a specific interval inside the tree).

In order to perform an overlap search with an interval tree, we recursively traverse it in a depth-first manner, using the stored max values to prune sub-trees to that do not overlap with the input range [29]. By traversing the tree, we can then find all intervals that overlap with the input.

On the other hand, in order to perform an exact search, we traverse the tree using the start of a node's range (instead of using its max value as we do in the overlap search) in order to find the exact input range (thus performing a process similar to binary search) [29].

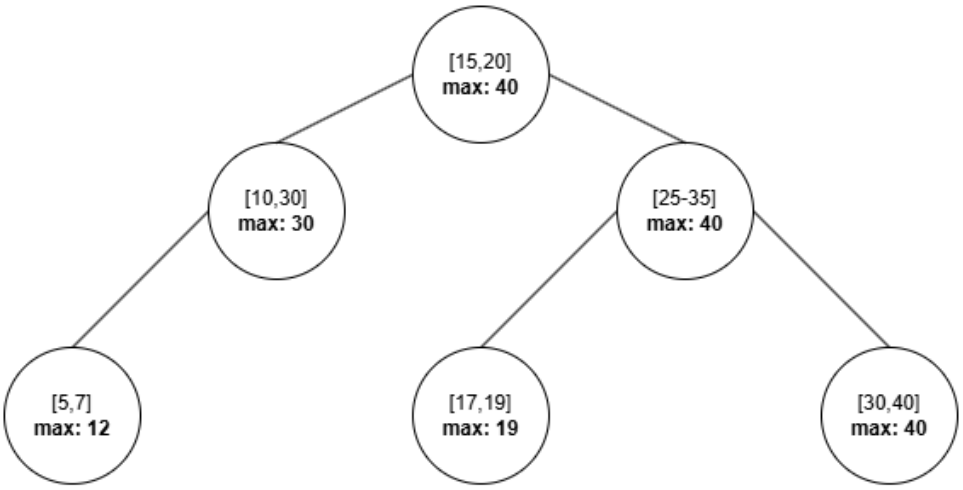


Figure 15: An example of an interval tree

The second component of our caching mechanism is the binary cache files. Each binary cache file contains information about the query such as its created date, the input vector and query range, the results of said query (the approximate nearest neighbors of the vector, along with each neighbor's distance from the query vector) and the moment last used (for reasons regarding its upper capacity limit). Due to the fact that every file is named after a query ID (which is created from hashing the query vector and range), we can fetch all query data just by knowing its query ID. Thus, it can be described to be a structure not too different from a hash-map (outwardly).

As its name suggests, it is a lazy caching mechanism. This means that the cache starts by being empty, and it slowly populates itself when queries are being performed (but not beforehand).

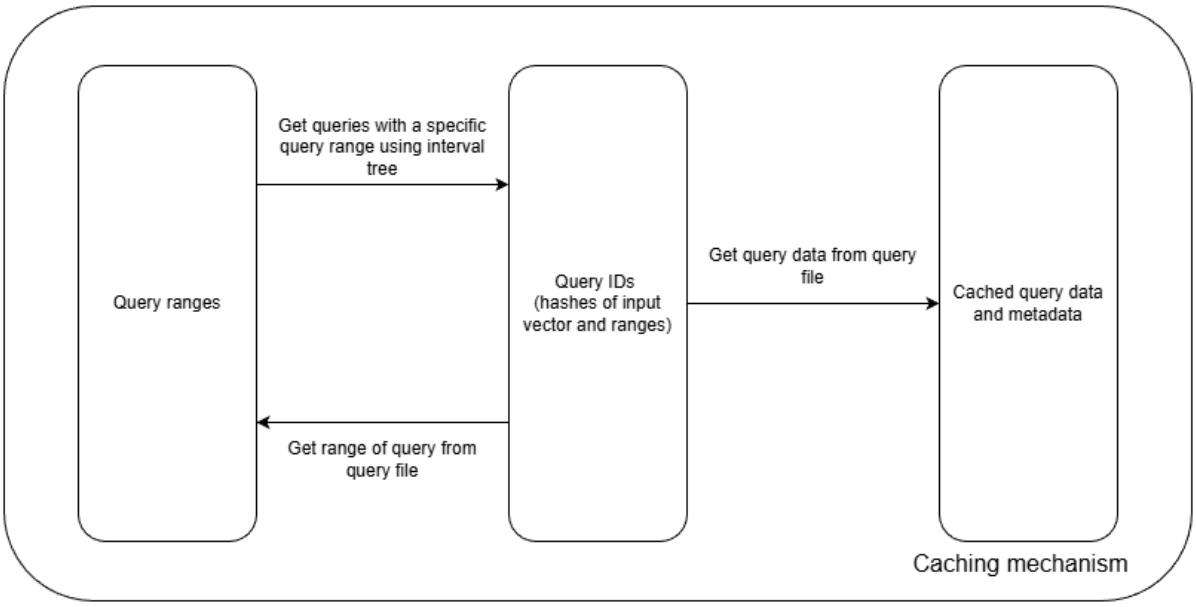


Figure 16: Types of data stored inside BLC-RFANN cache and how they interact with each other

5.2.2. Adding a query to the cache

After a query has been performed, the caching mechanism needs to store it for future reference. First, we hash the input vector along with the query range to produce a query ID using FNV-1a, which is a widely used non-cryptographic hashing algorithm [30]. Afterwards, we create the necessary entries on the interval tree and we also create a binary file named after the query ID, which as we have already mentioned above includes its creation date, the query (both the vector and the range), the number of neighbors asked in the query, the neighbors themselves, each neighbor's distance from the query vector, its creation date, and its last used date (which is initialized to be the same as the creation date).

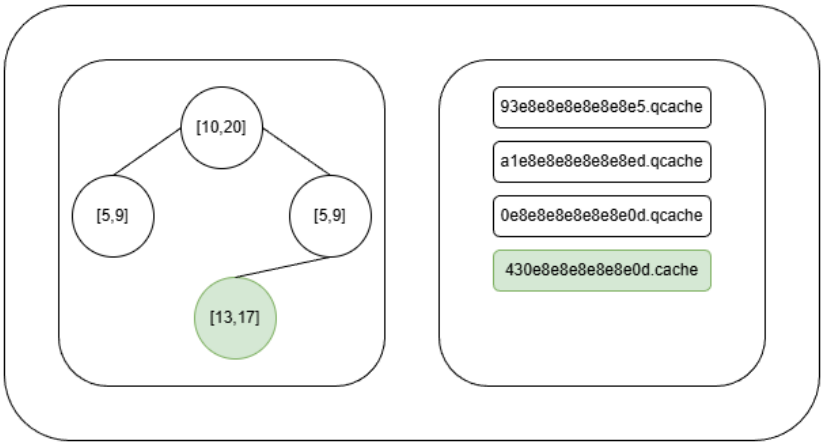


Figure 17: Updating the BLC-RFANN cache after a query has been performed

If the memory size of the cache exceeds a limit (which is configurable on index build time), we delete the queries that have the oldest "last used" dates, until the cache size falls below the upper limit.

5.2.3. Using the cache to perform queries

The lazy caching mechanism of BLC-RFANN begins by hashing the query data (the input vector and the input range) to create a query ID. After we create the query ID, we then use it to perform a lookup on the cache files directory in order to check if the exact query has been answered. If the exact query has been answered, and its number of stored neighbors (or K_{stored}) is greater or equal than the query K_{query}, we fetch the K_{query} nearest neighbors and return them as the response (and we also update the last used date in the cache metadata).

However, if there is no exact match, or if K_{query} is greater than K_{stored}, we go through the ranges stored in the interval tree, and we calculate the *IoU* (or *Intersection over Union*) between our query range and every range in the tree. If the IoU of the query range and a cached range exceeds a specific threshold (which we define by the *–range-sim* flag) we get the query IDs of this cached range. For every query ID, we fetch its cache entry and we then get the cached query's input vector. If the cosine similarity of the two vectors (the query vector and the cached query vector) is greater than another threshold (which is defined by the *–vector-sim* flag) and its $K_{similar\_query}$ is also greater than K_{query}, we can respond using the similar query.

If there are no cached queries where neither its range is similar enough nor its vector is similar enough to our own query, or if the $K_{similar\_query}$ is lesser than K_{query}, then the cache does not return any response. Our query will need to be executed by other methods (such as the B+ tree).

If we return data based on a specific cached query, we will need to update the cached query's last used time to the current moment.

5.2.4. Update cache after element insertion/deletion

An interesting caveat of BLC-RFANN's caching is its ability to adapt its records when elements are inserted into the index (or removed from it).

In either case, we start by taking the numeric value of the data object to be inserted (or removed) and then we fetch the cached queries whose ranges include the numeric value by using the interval tree.

Afterwards, in the case of an insertion we compare the vector to be inserted with lowest-ranked approximate nearest neighbor vector stored in each cache file. If the soon-to-be-inserted vector is closer to the query vector than the lowest-ranked neighbor, we continue

comparing in order to save the newly-inserted vector to its proper position using binary search (in order to continue having valid cache data). Finally, we increase the value of K stored inside the cache file by 1.

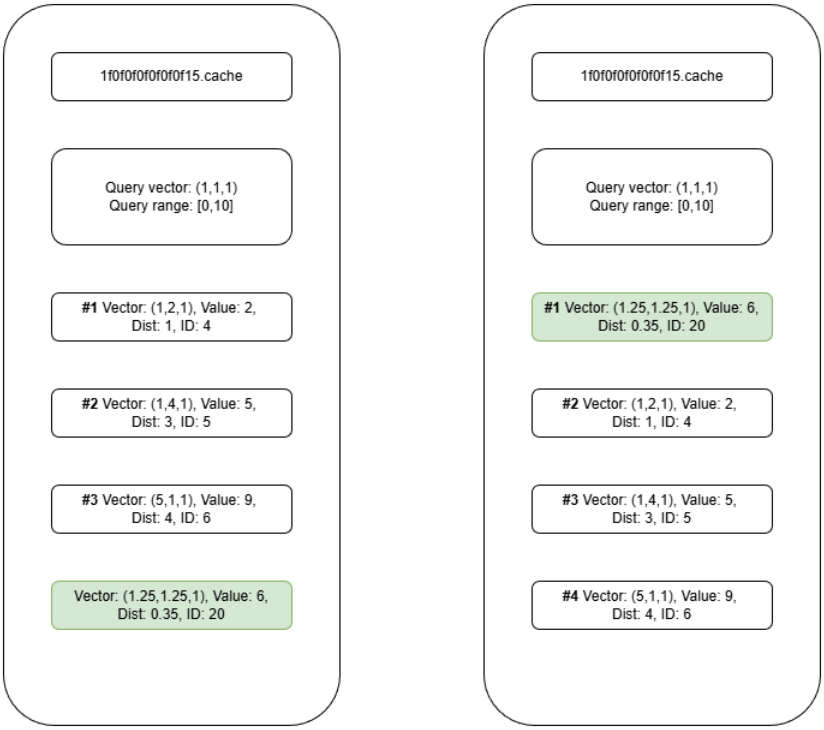


Figure 18: Updating the BLC-RFANN cache after element insertion

On the other hand, when an element is to be deleted, we check if the vector to be deleted is included inside of any cache file. If it is found inside a cache file, it is deleted. As we do during insertions, we also update the K field inside the cache files (in that case reducing it by 1).

This approach, even if it seems counter-intuitive at first, allows the caching mechanism to remain both accurate and intact, even in the face of multiple additions or deletions. As a result, the caching mechanism still manages to perform queries quickly and accurately even on very dynamic datasets.

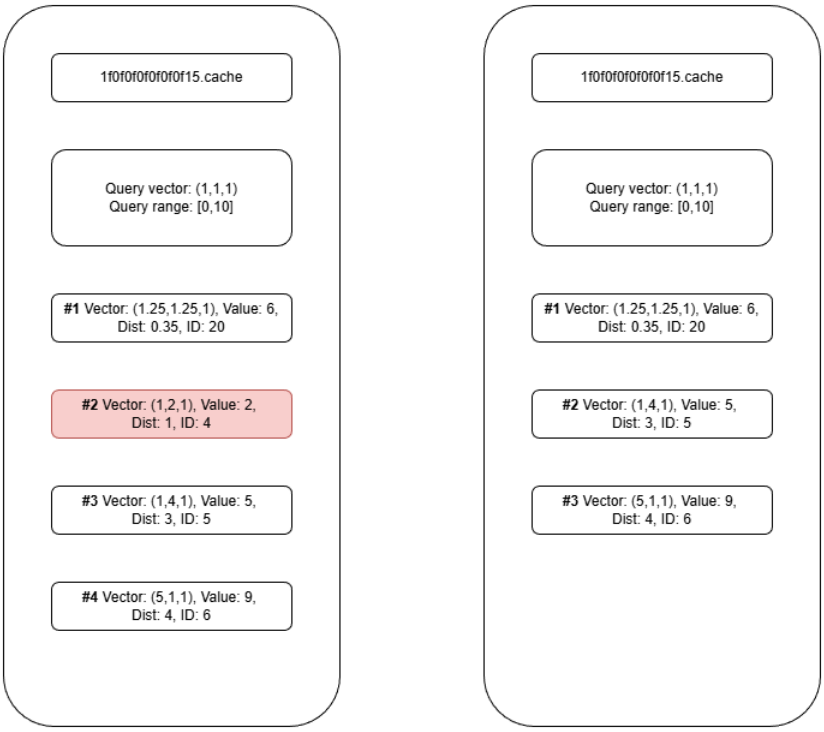


Figure 19: Updating the BLC-RFANN cache after element deletion

5.3. BLC-RFANN

After we have described the two basic components of BLC-RFANN, it is time to demonstrate how these components integrate with each other. In this section, we will get into detail on how BLC-RFANN as a whole builds its index, how it performs queries, and how it updates both its cache and B+ tree when insertions and deletions occur. We also will discuss about BLC-RFANN and its higher-level architecture (instead of discussing its implementation into code).

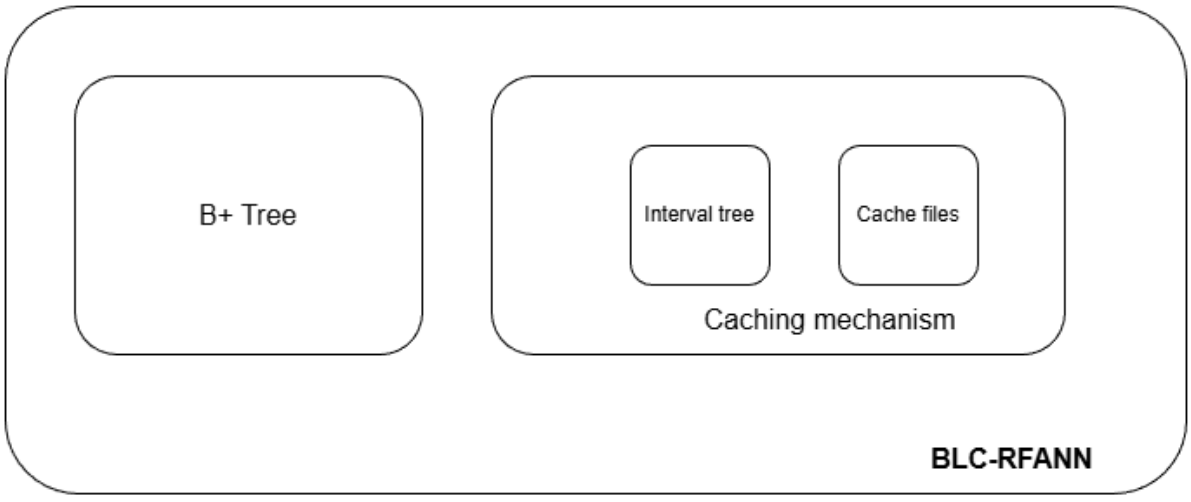


Figure 20: Structure of BLC-RFANN

5.3.1. Building a BLC-RFANN index

BLC-RFANN starts building its index by reading data from its data source, a FVECS file[7]. FVECS files (or *Float Vector files*) are binary files that contain floating point vectors as well as their dimensionality [31]. We load the file in chunks (the size of which is configurable in run time, setting the *–batch-size* flag), and then we insert them into the B+ tree in chunks. This process is repeated until all chucks are loaded and all data objects are inserted to the B+ tree.

Afterwards, we store the B+ tree into the disk, along with the vectors. There is the ability to either store vectors in the index itself or store them in a separate vector file by using the *--separate-vector-storage* flag. Storing vectors in separate files reduces the amount of space needed in disk, but integrating the vectors into the index can reduce query response time.

The cache is not populated during index creation owing to its lazy nature. Instead, it starts by being empty, ready to be populated by later queries.

5.3.2. Searching through a BLC-RFANN index

Contrary to the building process, the search mechanism of BLC-RFANN is somewhat more intricate. First, we want to check if the query (or a very similar query to it) has already been answered. We first use the cache in order to find the exact query (if it has at least K stored neighbors). If we do not find the exact query (or if it had less than K neighbors stored), the cache tries to find a query whose vector and range are similar to the current query's that also contains at least K nearest neighbors (using the mechanism already described in Subsection 5.2.3). If we find a cached query from either of these scenarios, we use it to respond to the current query.

However, if we cannot locate such a query from our cache, we then traverse the B+ tree from the start of the query range to this end in order to find the approximate nearest neighbors. We do that by going through the various leaves of the tree and comparing the vectors with the closest K vectors that we have already found (in a manner similar to finding the largest or smaller value in array by traversing through it). After we find the K approximate nearest neighbors, we then insert the query data into the cache in order to able to use them in the future (except in case we answer by using a cached query that is similar to the original query).

In the next page we present a flow chart, which explains how BLC-RFANN responds to RFANN queries:

[7] However, building an index out of randomly generated data is also possible. This feature was implemented for testing and debugging purposes

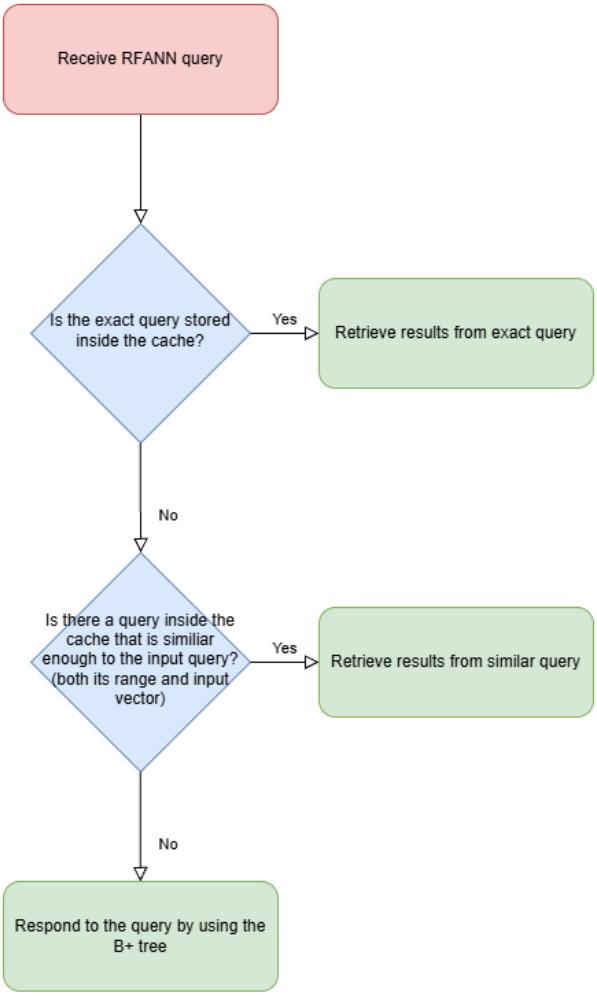


Figure 21: A flow chart explaining BLC-RFANN's query response process

5.3.3. Adding/Removing elements from a BLC-RFANN index

Quickly and efficiently adding and removing elements from one of BLC-RFANN's biggest strengths. As we have mentioned above, most RFANN algorithms do not have such capabilities (instead they have to recreate the index from scratch in order to include new data or removed deleted data).

When adding a data object to a BLC-RFANN, we start by inserting it to the B+ tree, and if need be, we rebalance the tree. Afterwards we check if any query in the cache needs to be updated due to the insertion (as we have mentioned in Subsection 5.2.4). If any query has become invalid due to the newly inserted datum (i.e. if the new data object if closer to a query vector than at least one neighbor already stored in the cache), we update the cache to store the new element as well, maintaining the correct order of nearest neighbors.

In the case of deleting items from a BLC-RFANN we perform an equivalent process. We start by removing the node from the B+ tree, and if it is necessary we rebalance the tree (by merging nodes or borrowing leaves between siblings). Finally, we update the cache in order to remove the deleted data object from any query results.

5.4. Technical implementation

In this section, we will briefly discuss about the BLC-RFANN implementation prototype in a more technical level.

5.4.1. Basic information about the implementation

The source code was written using C++, which is also used by other RFANN methods such as iRangeGraph [1], SeRF [2] and formerly FilteredVamana and StitchedVamana [4] (a new version of Filtered-DiskANN, the software that contains both methods, was recently released, which was written in Rust).

In order to build the executable files we use *CMake* [32], which is a commonly-used cross-platform tool for building applications from C++ code.

Generally speaking, no external libraries have been except *nlohmann*'s JSON parser, a commonly used library for parsing JSON file (the data objects' numerical values and query ranges are derived from JSON files) [33].

5.4.2. Utility files of our implementation

In the table below we present the utility (non-executable) files that our executables use and their purpose:

Table 2: Utility files of BLC-RFANN

File	Description
b_plus_tree.cpp	B+ tree manager. Heavily uses page_manager.cpp
DataObject.cpp	Class definition for DataObject, a structure that contains a vector and a numerical value
index_directory.cpp	Index folder layout and configuration manager
logger.cpp	Internal logger service
page_manager.cpp	Low level page manager for storing our index to disk
query_cache.cpp	Query mechanism subsystem
vector_store.cpp	Utility for storing vectors inside in the vector storage

5.4.3. Executables of our implementation

BLC-RFANN's implementation contains a number of executables which provide various functionalities to the end user. In the tables below, we exhibit a basic description of their use, as well as some of their runtime parameters.

Table 3: Executables created from BLC-RFANN's code

Executable name	**Description**
add_element_to_index	Add element to existing BLC-RFANN index
build_index_from_fvecs	Create index from FVECS file
build_index_from_synthetic	Create index from randomly generated data
clear_cache	Clear cache of a specific index
read_cache	Read data from a index's cache
remove_element_from_index	Remove a specific element from existing BLC-RFANN index
search_from_index_test	Perform multiple queries (used for benchmarks)
search_from_index	Perform a single query

Table 4: Runtime parameters of add_element_to_index

Parameter Name	**Obligatory?**	**Description**
index	Yes	Index to which the element will be added
key	Yes	Numerical value of the data object to be added
vector	Yes	Vector to be added into the index (written in CSV format)

Table 5: Runtime parameters of build_index_from_fvecs

Parameter Name	**Obligatory?**	**Description**
input	Yes	FVECS file containing the dataset's vectors
output	Yes	Location where the index will be stored
order	No	Order of the B+ tree
batch-size	No	Size of each vector batch to be inserted to the index
max-cache-size	No	Maximum size of the cache (in MB
label-path	No	Path to a JSON file that contains the numerical value of each vector (if it does not exist, we use a vector's id as its numerical value)

Table 6: Runtime parameters of build_index_from_synthetic

Parameter Name	**Obligatory?**	**Description**
output	Yes	Location where the index will be stored
dataset-size	Yes	The number of vectors to be added into the index
dimension	No	Dimension of each generated vector. Default value is 128
order	No	Order of the B+ tree

batch-size	No	Size of each vector batch to be inserted to the index
max-cache-size	No	Maximum size of the cache (in MB

Table 7: Runtime parameters of clear_cache

Parameter Name	Obligatory?	Description
index	Yes	Path to the index whose cache will be cleared
yes	No	Skip confirmation prompt

Table 8: Runtime parameters of read_cache

Parameter Name	Obligatory?	Description
index	Yes	Path to the index whose cache be read
query-id	No	Read a specified cached query
summary	No	Read a summary version of the cached data

Table 9: Runtime parameters of remove_element_from_index

Parameter Name	Obligatory?	Description
index	Yes	Path to a specific index
key	Yes	Numerical value whose elements will be removed
vector	No	Delete only a specific vector (and not all objects whose numerical value equals key)

Table 10: Runtime parameters of search_from_index_test

Parameter Name	Obligatory?	Description
index	Yes	Path to a specific index
queries	No	Path to a FVECS query file
groundtruth	No	Path to an IVECS groundtruth file. Used to calculated recall
qrange-path	No	Path to a JSON query range file
num-queries	No	Perform only a specific amount of queries from the queryset
no-cache	No	Do not use cache for this session
parallel	No	Perform queries in parallel
threads	No	How many concurrent queries will be run
memory-index	No	Load index into memory before executing the queries
vec-sim	No	Vector similarity threshold (used to find similar queries from the cache)

range-sim	No	Range similarity threshold (used to find similar queries from the cache)

Table 11: Runtime parameters of search_from_index

Parameter Name	Obligatory?	Description
index	Yes	Path to a specific index
value	No[8]	A specific numeric value
min	No	Minimum value of the query range
max	No	Maximum value of the query range
vector	No	Input vector (in CSV format). If not specified, we perform a window search throughout the dataset
K	No	The number of nearest neighbors requested. Requires an input vector
limit	No	Print only the first results (if no input vector)
parallel	No	Perform queries in parallel
threads	No	How many concurrent queries will be run
memory-index	No	Load index into memory before executing the queries
vec-sim	No	Vector similarity threshold (used to find similar queries from the cache)
range-sim	No	Range similarity threshold (used to find similar queries from the cache)

Finally, each executable has a *–help* flag which provides further information and all possible runtime parameters.

5.4.4. More information – Source code

The source code for BLC-RFANN (along with a user manual) is stored in the following GitHub repository: https://github.com/arstotzkan/b_plus_tree_for_rfann.

[8] It is not obligatory by itself, however either it or min and max are needed

6. Experimental Setup

After we have discussed about various existing RFANN algorithms and provided our implementation (BLC-RFANN) in the previous sections, our next task is to conduct experiments with BLC-RFANN and various other methods, in order to compare BLC-RFANN's performance compared to other state-of-the-art methods.

In this section, we will elaborate on the experiments and their nature, in order to present their results in the *Experiment results* section.

6.1. Planned experiments

We are going to perform experiments that are commonly performed for RFANN (or similar methods) [1, 2, 3, 4, 6] .

Given a specific dataset, we first build an index for each method in order to find the index build time, and the index memory space.

Afterwards, we then perform a set of queries on each index. We try to perform multiple sets of queries, each set of each representing different levels of selectivity (50%, 10% and 1%). For each set, we record recall and the amount of queries responded per second.

We perform queries at different selectivity levels: high (50%), medium (10%) and low (1%) because as we have seen from existing academic literature (see Section 4.7 *Existing experimental studies and their results*), the performance of various methods tends to differ on various selectivity levels [1, 2, 3, 4, 6].

Moreover, for BLC-RFANN we will make tests for different repeatability levels (i.e. the percentage of queries in the query set that can has already been answered in the past) for each dataset-selectivity combination in order to measure how much does the cache reduce query response time. This will not be done on the rest of the methods, since they have no caching mechanism for retrieving past queries, so it is extremely unlikely to experience a noticeable difference in performance if repeatability changes.

6.2. Datasets used in the experiments

We are going to examine the methods using three different datasets, *SIFT* [20], *GloVE* [34] and *GIST* [20].

SIFT is a dataset consisting of image embeddings. Similarly to GIST (which we will cover later), it was introduced in the "*Product Quantization for Nearest Neighbor Search*" paper, written by Hervé Jégou, Matthijs Douze and Cordelia Schmid [20]. It is a very commonly used dataset for ANN studies (for example it is used in [3, 4, 6]). The original dataset consists of 1,000,000,000 (*ANN_SIFT_1B*) vectors each having 128 dimensions. However, versions that contain 1,000,000 vectors (*ANN_SIFT_1M*) or 10,000 vectors (*ANN_SIFT10K* or

ANN_SIFTSMALL) are also available. Due to hardware limitations, we are going to use ANN_SIFT_1M for our study.

GloVe (or *Global Vectors*) is a dataset consisting of word embeddings. It was introduced in "*GloVe: Global Vectors for Word Representation*", a paper by Jeffrey Pennington, Richard Socher and Christopher D. Manning from Stanford University [34]. It is generated by the GloVe model, which is also presented in the above paper. In our study we are using a subset of GloVe called GloVe1.2M, consisting of 1.2 million vectors, each having a dimensionality of 200.

The final dataset we are going to use is GIST [20]. It was presented on the same paper as SIFT, and it is a dataset of image color descriptions. It contains 1,000,000 vectors, with each vector having 960 dimensions.

Table 12: Basic information about the datasets used

Dataset name	Number of vectors	Number of queries	Number of dimensions
SIFT	1,000,000	10,000	128
GloVe	~1,200,000	1,000	200
GIST	1,000,000	1,000	960

6.3. RFANN Methods used in the experiments

The methods we are going to experiment on are the following:

- BLC-RFANN
- iRangeGraph [1]
- SeRF [2]
- FilteredVamana [4]
- StitchedVamana [4]
- Milvus [5]
- OptimizedPostfiltering [3]

More information about these methods is provided in Sections 4.2 (*iRangeGraph*), 4.3 (*SeRF*), 4.4 (*FilteredVamana/StitchedVamana*), 4.5 (*Milvus*), 4.6 (*OptimizedPostfiltering /SuperPostfiltering*) and 5.3 (*BLC-RFANN*).

BLC-RFANN index creation was run with batch-size equal to 333334, while range-sim and vec-sim were set to 1 (select only exact queries from the cache). This was done so we exhibit the performance of the B+ tree (when repeatability is 0) and then showcase the performance improvement caused by the cache.

The parameters used for the other methods were the same ones used in the experimental study [6]. The only exception is Milvus, where we used Milvus Lite, a version of Milvus that does not need Docker to run (in order to accurately measure the index side).

For all methods and datasets, we use 8 threads for index creation, while we only use 1 for querying, since parallelism will improve the performance uniformly. [6]

6.4. Metrics used in the experiments

The metrics which we are used in this study are the ones defined in Section 4.7 (*Existing experimental studies and their results*), that is (1) index build time, (2) index memory size, (3) QPS (i.e. number of queries answered by second) and (4) recall.

For our experiments we further define the metrics as follows:

- Index build time: the time that the build process of a method runs
- Index memory size: how much space do the various index files need
- QPS: the execution time of the search process divided by the number of queries in the queryset

We measure index build time and index memory space during the index build, while recall and QPS (queries answered per second) are recorded when the queries will be performed).

In this study, we do not subtract the sizes of the vectors used for calculating index memory size, contrary to other papers such as the iRangeGraph paper [1].

These metrics allow us to get a fuller picture of both the amount of resources needed to create the indices (or to re-create them, since almost all methods mentioned above do not allow the insertion or deletion of elements), but also their effectiveness (the amount of time needed to respond to a query as well as the reliability of their answers) in a manner that is most useful to end-users (since we calculate index build time and QPS based on execution time).

Finally, we wish to note that Euclidean distance is used for distance calculation in all methods and datasets.

6.5. Experiment environment / Hardware specs

These experiments are run on a *WSL* (*Windows Subsystem for Linux*) environment. WSL is a component of Microsoft Windows that allows the use of a Linux environment inside Windows, allowing for the shared use of the file system.

All experiments are run on a system with an Intel i5-4460 processor, with a frequency of 3.2 GHz, 4 physical cores and 4 threads, 16 gigabytes of DDR3 RAM (of which 12 could be used by WSL) and a hard disk of 1 terabyte.

All indices are built and used from memory (i.e. we load indices in memory before performing any queries). However, in order to allow their reuse, we stored them in storage (so they can be loaded again by the various executables).

7. Experiment results

In this section we demonstrate the results from the experiments described in Section 6. Each section will display results from the different types of experiments we have conducted (Index Build, Performing queries with high selectivity, and also performing queries with medium and low selectivity)

7.1. Index build experiment results

The first experiment that we conduct is to build an index for all methods tested. We perform this experiment in order both to measure BLC-RFANN's index build time and index size and compare it to the performance of the other RFANN methods.

Table 13: Index build time (in seconds)

	SIFT	GloVe	GIST
iRangeGraph	5934	12508	20985
SeRF	460	905	2070
FilteredVamana	298	202.68	1411.19
StitchedVamana	407.61	171.42	8464.67
Milvus	23172	29063	27355
OptimizedPostfiltering	9448	17043	36167
BLC-RFANN	18.26	40.38	1295.9

Table 14: Index size (in MB)

	SIFT	GloVe	GIST
iRangeGraph	1492	1024	1448
SeRF	207	122	141
FilteredVamana	650.87	1113.17	3832.69
StitchedVamana	650.87	1103.17	3824.69
Milvus	799.78	1322.7	4045
OptimizedPostfiltering	2088	480	1588
BLC-RFANN	546.19	977.36	3720.03

As we can see from the results, BLC-RFANN takes significantly less time to build its index compared to other methods, sometimes taking much less than 1% of the time that methods such as iRangeGraph or OptimizedPostfiltering need to build their own indices. However, this disparity becomes much less noticeable on datasets which have much higher dimensionality such as GIST.

The amount of time that BLC-RFANN needs to create an GIST dataset can be considered disproportionate compared the amount needed for other two datasets. Building a BLC-RFANN index for the GIST dataset takes about 30 times more time compared to building an index for GloVe, even though GloVe has about the same amount of vectors and each GloVe vector has about a fourth to a fifth of vector from the GIST dataset. A possible reason for this imbalance can be that CPU can become a bottleneck when trying to load a large volume of

data. This can also be a possible cause for Milvus' extremely slow build time as well, always being among the two methods that take the most amount of time to build their indices.

Regarding index memory size, BLC-RFANN stands competitive compared to other methods, although methods such as SeRF and OptimizedPostfiltering use less memory space. However, due to the fact that BLC-RFANN stores vectors without any compression for its index, it can need much more space compared to other methods, especially for larger sized datasets or datasets whose vectors have high dimensionality (such as GIST).

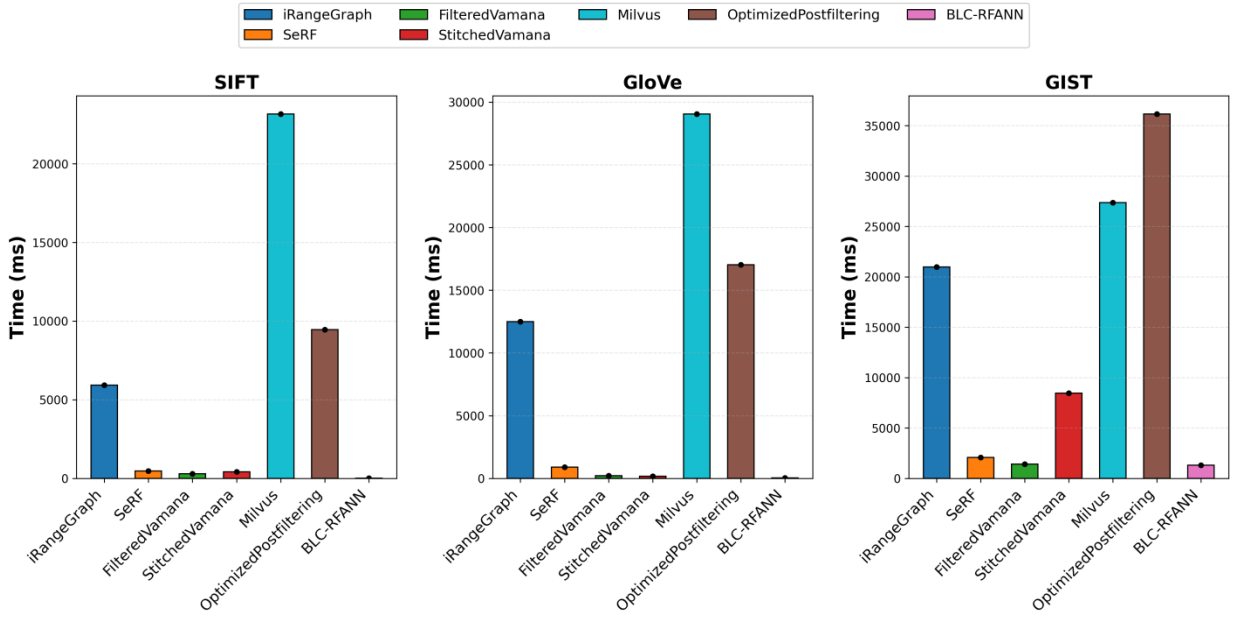


Figure 22: Index build time (in seconds)

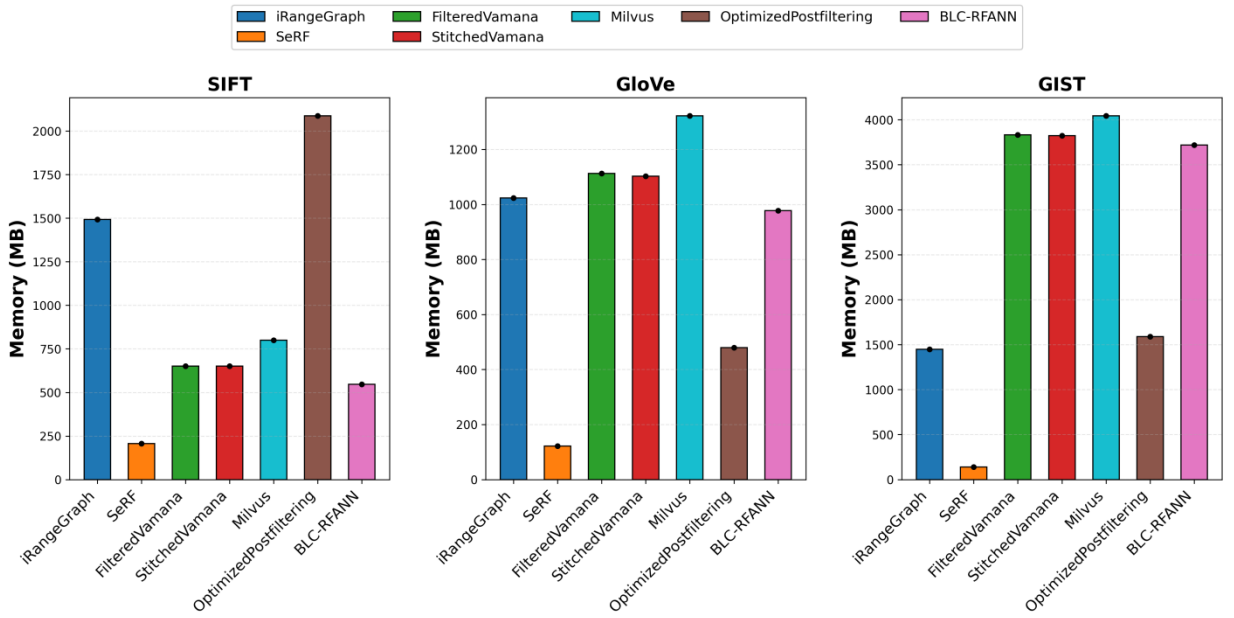


Figure 23: Index memory space (in MB)

7.2. Performing queries with high selectivity

The second experiment we conduct is to perform queries at a very high level of selectivity (at about 50% percent). This means we execute queries where about half of the data objects of the dataset are inside our query range. This scenario is optimal for methods that use post-filtering methods, but causes problems for methods that use pre-filtering (such as BLC-RFANN).

This is be the first experiment where we will test BLC-RFANN in various *repeatability* scenarios, meaning we will test BLC-RFANN with querysets whose parts are repeated again. For example, when we are testing BLC-RFANN in a scenario with 25% repeatability, 25% of the queries will be derived from rest of the queryset (thus performing those queries twice).

Table 15: Queries per second at 50% selectivity

	SIFT	**GloVe**	**GIST**
iRangeGraph	864	19	0.12
SeRF	874	588	8
FilteredVamana	874.59	1936.9	5.74
StitchedVamana	410.78	414.2	0.65
Milvus	164	48.09	0.23
OptimizedPostfiltering	231	0.78	0.1
BLC-RFANN (0% repeatability)	5.0	3.6	0.6
BLC-RFANN (25% repeatability)	6.9	4.8	0.7
BLC-RFANN (50% repeatability)	10.7	7.2	0.9
BLC-RFANN (75% repeatability)	21.8	14.2	2.3

Table 16: Recall at 50% selectivity

	SIFT	**GloVe**	**GIST**
iRangeGraph	0.99	0.83	0.97
SeRF	0.97	0.5	0.79
FilteredVamana	0.96	0.55	0.71
StitchedVamana	>0.99	0.78	0.98
Milvus	0.81	0.65	0.73
OptimizedPostfiltering	0.99	0.88	0.99
BLC-RFANN (0% repeatability)	1.0	1.0	1.0
BLC-RFANN (25% repeatability)	1.0	1.0	1.0
BLC-RFANN (50% repeatability)	1.0	1.0	1.0
BLC-RFANN (75% repeatability)	1.0	1.0	1.0

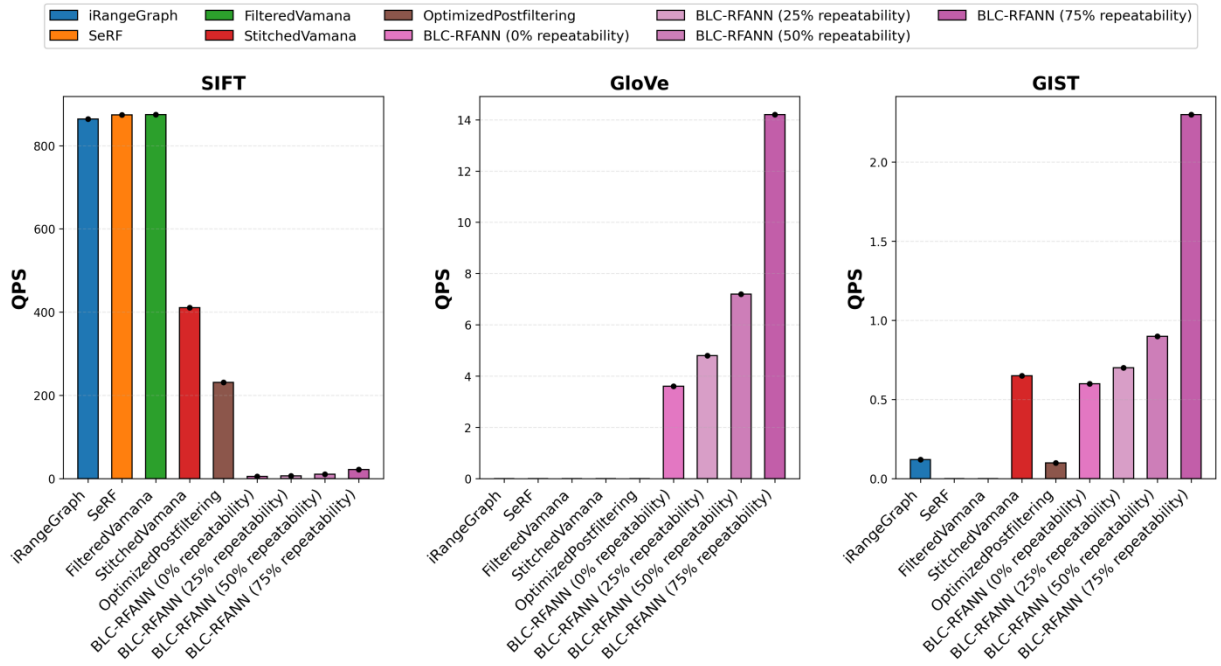


Figure 24: QPS executed for queries with 50% selectivity (only methods with >=90% recall are shown)

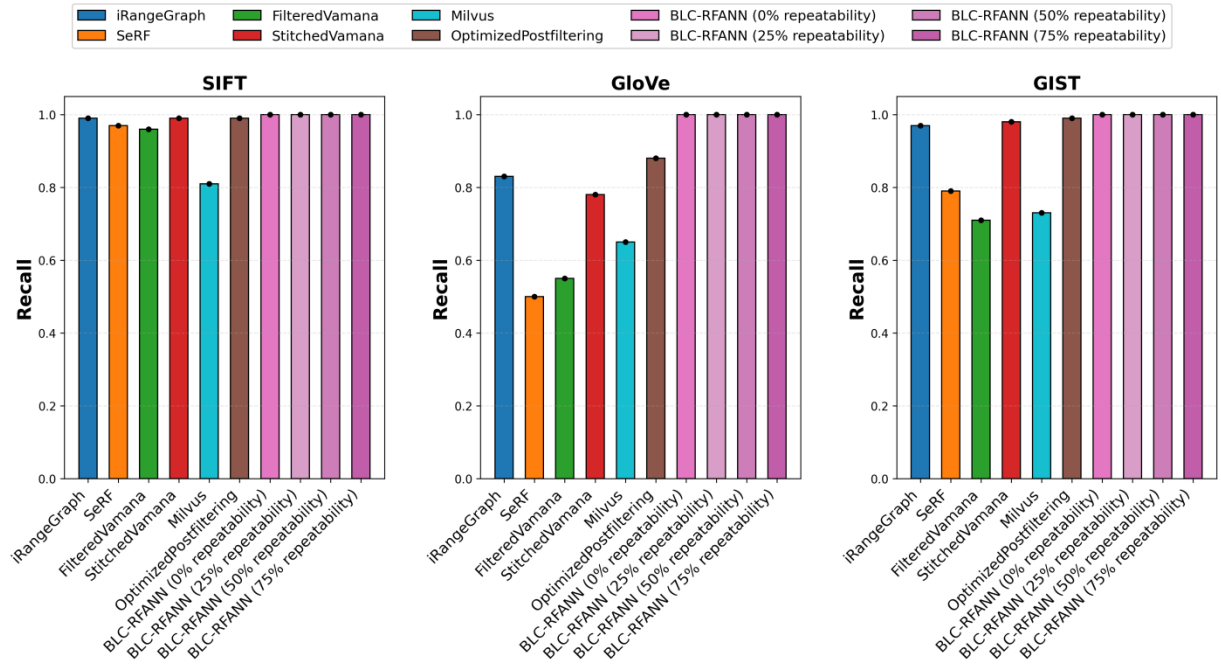


Figure 25: Recall for queries with 50% selectivity

In regards to the tests conducted for the SIFT dataset, BLC-RFANN's performance could not be considered satisfactory. All other RFANN methods managed to perform a higher level of QPS compared to BLC-RFANN even on high-repeatability scenarios. However, BLC-RFANN managed to achieve a perfect level of recall even on the GloVe dataset, where other methods fared rather poorly (with no method reaching recall of 90%, which is considered a baseline for many RFANN papers [1, 6]).

BLC-RFANN also performed rather well on the GIST dataset. Even though methods such as SeRF or FilteredVamana performed more QPS, BLC-RFANN was the only method that both achieved a high level of QPS, while also remained extremely accurate. This can be explained due to BLC-RFANN's rather simple structure, which allows it to consistently perform on data-heavy workloads even on lower-resource systems.

7.3. Performing queries with medium selectivity

The next experiment we conduct is to perform queries at a medium level of selectivity (about 10%). This means for every query in the queryset, about 10% of the dataset elements are inside the query's query range. This is an intermediary experiment, which is unlikely to cause many problems for neither pre-filtering nor post-filtering methods.

Table 17: Queries per second at 10% selectivity

	SIFT	**GloVe**	**GIST**
iRangeGraph	1029.31	43	0.12
SeRF	1321	768	<0.01
FilteredVamana	1185.83	2594.71	9.28
StitchedVamana	509.08	551.08	3.4
Milvus	211.20	97.94	0.82
OptimizedPostfiltering	19.26	1.25	0.09
BLC-RFANN (0% repeatability)	14.6	15.4	0.6
BLC-RFANN (25% repeatability)	20.7	21.9	1.9
BLC-RFANN (50% repeatability)	31.1	28.0	2.7
BLC-RFANN (75% repeatability)	65.7	64.3	3.1

Table 18: Recall at 10% selectivity

	SIFT	**GloVe**	**GIST**
iRangeGraph	0.99	0.94	0.99
SeRF	0.98	0.64	0.83
FilteredVamana	0.77	0.6	0.41
StitchedVamana	>0.99	0.92	>0.99
Milvus	0.79	0.57	0.68
OptimizedPostfiltering	0.99	0.95	0.99
BLC-RFANN (0% repeatability)	1.0	1.0	1.0
BLC-RFANN (25% repeatability)	1.0	1.0	1.0
BLC-RFANN (50% repeatability)	1.0	1.0	1.0
BLC-RFANN (75% repeatability)	1.0	1.0	1.0

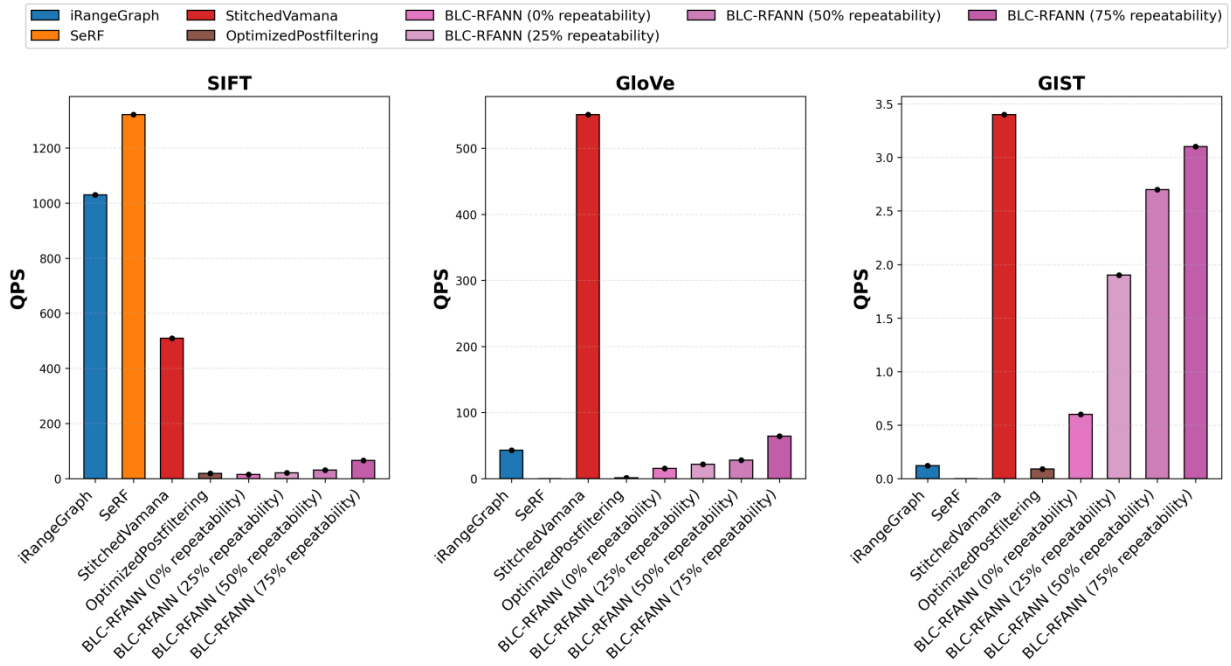


Figure 26: QPS executed for queries with 10% selectivity (only methods with >=90% recall are shown)

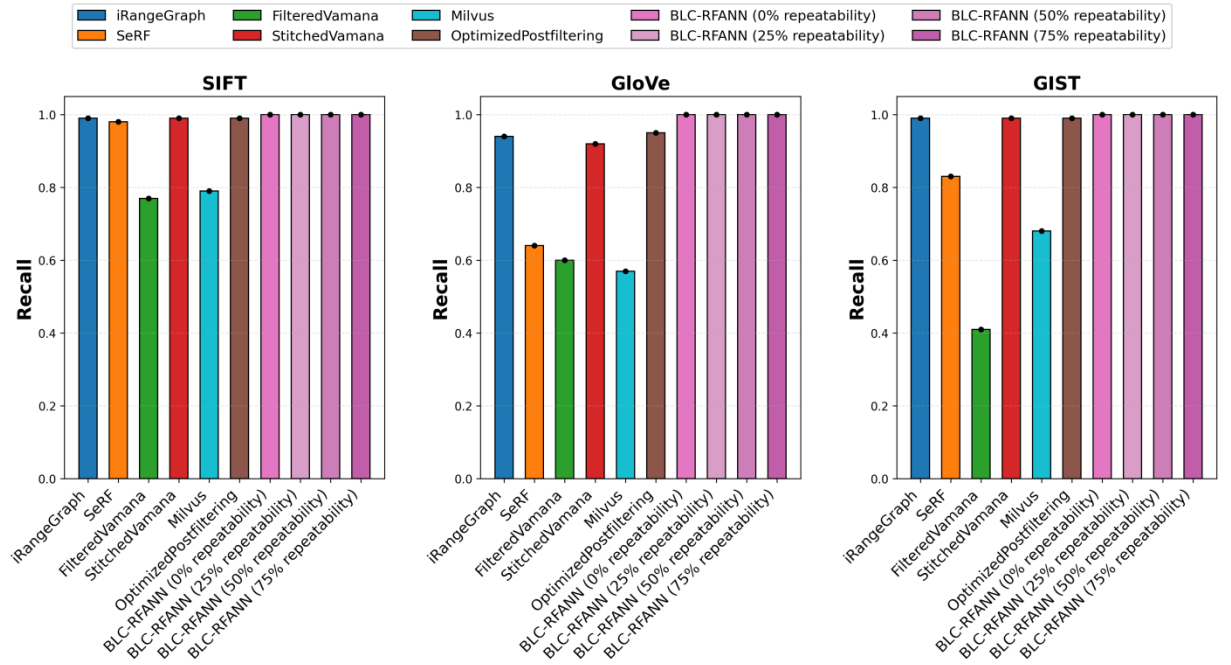


Figure 27: Recall for queries with 10% selectivity

As also shown in the previous experiment, BLC-RFANN's performance for the tests conducted on the SIFT dataset cannot be considered satisfactory, however it performed much better on this dataset compared to the previous experiment.

On the other two datasets, we start noticing a pattern: methods that perform a high level of QPS achieved a low level of recall and vice versa (except StitchedVamana, which performed rather well in both metrics). In these datasets, BLC-RFANN fares quite well, performing more QPS than most of its "accurate" (i.e. methods with a high level of recall) methods such as

iRangeGraph and OptimizedPostfiltering. This effect is even more noticeable on the GIST dataset, where BLC-RFANN performs quite well (narrowly being defeated by StitchedVamana on 75% repeatability), even when repeatability is very low (or non-existent).

7.4. Performing queries with low selectivity

The final experiment we carry out is to perform queries at a low selectivity level (at about 1% percent). This will be an environment where we expect pre-filtering methods such as BLC-RFANN to perform well, while post-filtering methods' performance will decrease.

Table 19: Queries per second at 1% selectivity

	SIFT	**GloVe**	**GIST**
iRangeGraph	169.73	9.62	0.21
SeRF	2163	4792.0	1573.0
FilteredVamana	1185.83	50540.45	86.34
StitchedVamana	784.65	993.42	39.89
Milvus	245.59	162.69	0.49
OptimizedPostfiltering	43.32	5.42	0.22
BLC-RFANN (0% repeatability)	7.4	40.3	1.5
BLC-RFANN (25% repeatability)	37.4	89.2	2.4
BLC-RFANN (50% repeatability)	64.7	225.4	4.2
BLC-RFANN (75% repeatability)	209.2	196.1	2.8

Table 20: Recall at 1% selectivity

	SIFT	**GloVe**	**GIST**
iRangeGraph	>0.99	0.98	>0.99
SeRF	0.93	0.38	0.67
FilteredVamana	<0.01	0.14	<0.01
StitchedVamana	>0.99	0.98	>0.99
Milvus	0.74	0.47	0.60
OptimizedPostfiltering	>0.99	0.97	>0.99
BLC-RFANN (0% repeatability)	1.0	1.0	1.0
BLC-RFANN (25% repeatability)	1.0	1.0	1.0
BLC-RFANN (50% repeatability)	1.0	1.0	1.0
BLC-RFANN (75% repeatability)	1.0	1.0	1.0

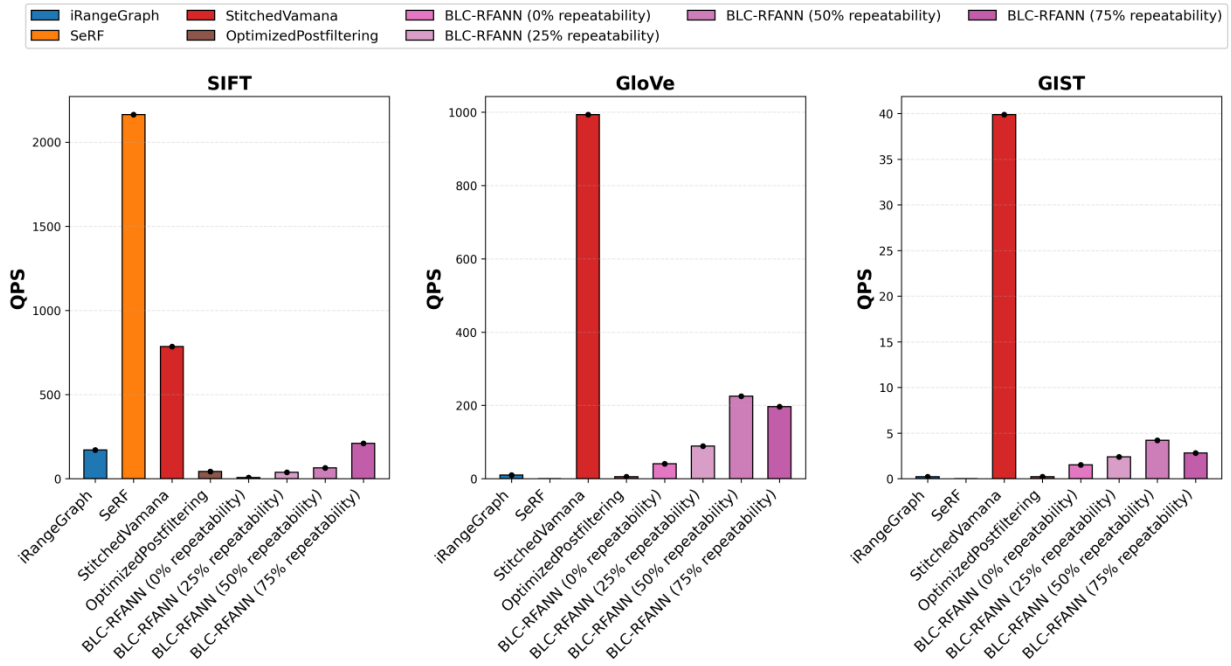


Figure 28: QPS executed for queries with 1% selectivity (only methods with >=90% recall are shown)

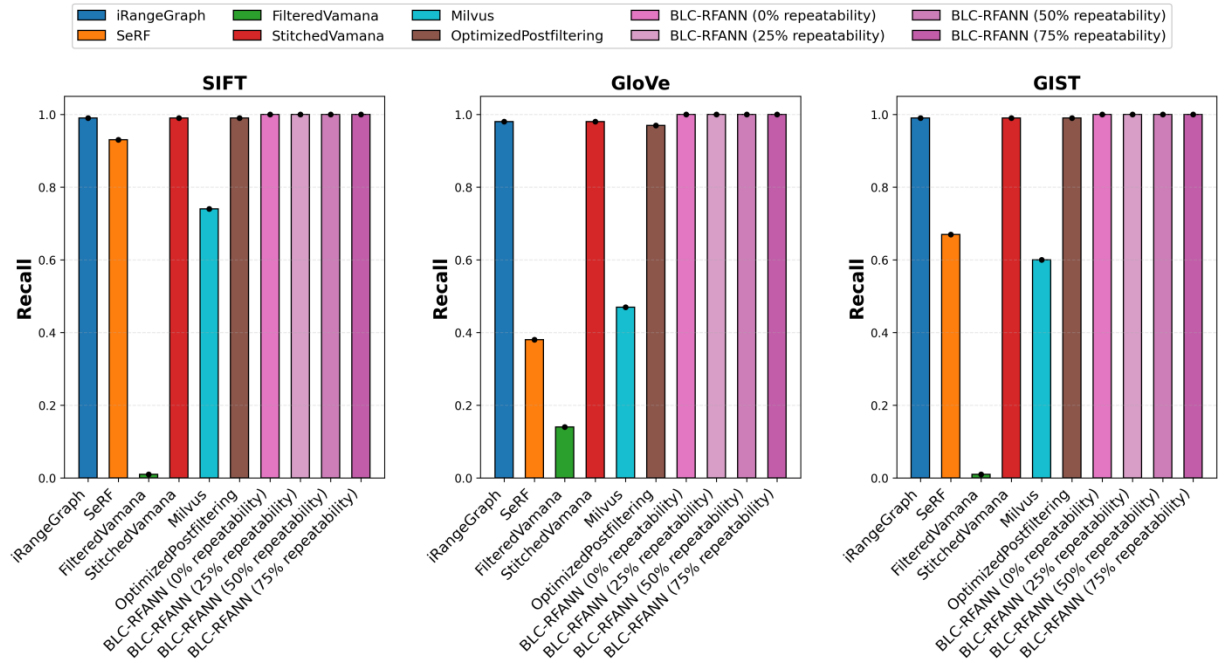


Figure 29: Recall for queries with 1% selectivity

As the results show, BLC-RFANN has an acceptable performance on the SIFT dataset, especially when repeatability is high.

On the GloVe and GIST datasets, we notice the same pattern mentioned in the previous experiment once again: methods that perform the most amount of QPS tend to achieve low levels of recall, while methods achieving high recall do not perform nearly as many QPS

(excluding StitchedVamana). In this regard, BLC-RFANN once again performs very well, performing more QPS than all other high-recall methods except StitchedVamana, while also having a perfect recall level. This happens even when repeatability levels are very low (thus BLC-RFANN good performance is not only caused due to the caching mechanism).

An interesting finding (although not strictly related to the goal of the experiments) is FilteredVamana's poor recall level. In most queries, FilteredVanama failed to find enough neighbors (maybe due to aggressive pruning during index creation), thus reducing its recall score.

8. Conclusions/Discussion

After conducting the experiments, we can reach the conclusion that BLC-RFANN is a method that performs excellently in two of our four metrics we used to compare it to other RFANN methods (those being index build time and recall), while also having a lower than median index size to most other RFANN methods.

However, even on the final metric (that is QPS), BLC-RFANN also manages to remain competitive at many scenarios. Such scenarios include low selectivity, high repeatability, or datasets with either a large amount of elements or datasets whose vectors have very high dimensionality (and scenarios where two or more of these factors are present). In these scenarios, BLC-RFANN can perform at par (or sometimes better) than many state-of-the-art methods such as iRangeGraph.

As we already expected, BLC-RFANN performs quite well on low selectivity scenarios. This can be explained by the nature of a B+ tree: it is easier to perform lower selectivity queries, since we only need to traverse fewer leave nodes compared to higher selectivity queries.

Moreover, the caching mechanism manages to rapidly increase the number of queries executed per second. For example, at 1% selectivity, the amount of QPS performed by BLC-RFANN increases more than 500% (!) when repeatability is increased by only 25%. If we had performed our test on an SSD drive (where I/O is much faster), it would be possible that the caching mechanism's performance would improve even more.

Thus, BLC-RFANN is a RFANN method that can be considered quite effective in certain applications, such as use cases where very high levels of recall are absolutely necessary, scenarios where the dataset's memory size is too large compared to available resources (thus creating bottlenecks for other RFANN methods), or scenarios where the dataset is very dynamic (i.e. new elements are constantly being added to our dataset, or if elements are being removed from it).

8.1. Possible areas of future interest

There are many possible areas for future research derived from BLC-RFANN, however two of them would certainly deserve further study in future research.

The first possible area of research is a test study on how the *–vec-sim* and *–range-sim* parameters influence BLC-RFANN's performance (i.e. recall and QPS). These parameters allow BLC-RFANN to fetch similar enough queries from the cache if the exact queries are not stored inside it. This benchmarking might seem simple at first glance, however other factors will also influence the results, such as cache size (a cache with more entries will be more likely to contain similar queries than an empty cache), and the queryset itself (if the queries of a queryset are similar to one another, then it is more likely for the cache to retrieve similar queries). Any future research must also take these parameters into note instead of only trying to test for various *–vec-sim* and *–range-sim* combinations.

The second possible area would be combining BLC-RFANN's caching mechanism with another RFANN method, such as one of other methods mentioned in this study or with another more "primitive" method, and then test how much would the method's QPS performance change compared to the version without caching. It is expected it would be beneficial for many methods, although it would be less so for ones that use relative neighborhood graphs (since such methods already store such information in their indices).

9. References

[1] Yuexuan Xu, J. G. (2024). IRangeGraph: Improvising range-dedicated graphs for range-filtering nearest neighbor search. Proc. ACM Manag. Data, 2, 1-26.

[2] Chaoji Zuo, M. Q. (2024). SeRF: Segment graph for range-filtering approximate nearest. Proc. ACM Manag. Data, 2, 1-26.

[3] Engels, J. a. (2024). Approximate nearest neighbor search with window filters.

[4] Gollapudi, S. a. (2023). Filtered-DiskANN: Graph algorithms for approximate neighbor search with filters. Proceedings of the ACM Web Conference 2023. Austin TX USA.

[5] Zhifeng, J. W. (2021). Milvus: A Purpose-Built Vector Data Management System. SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.

[6] Ma, M. L. (2025). Attribute Filtering in Approximate Nearest Neighbor Search: An In-depth Experimental Study.

[7] Cheng, A. L. (2024). UNIFY: Unified Index for Range Filtered Approximate Nearest Neighbors Search. Proc. VLDB Endow., 18(4), 1118-1130.

[8] Subramanya, S. J. (n.d.). DiskANN: Fast accurate billion-point nearest neighbor search on a single node.

[9] Malkov, Y. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 42, 824-836.

[10] Cai, C. F. (2016). EFANNA: An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph.

[11] Wang, M. J. (2025). DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. Proc. ACM Manag. Data, 3(3).

[12] Russell, S., & Norvig, P. (2020). Artificial intelligence: A modern approach (4th American ed.). Pearson.

[13] Johnson, J. a. (2019). Billion-Scale Similarity Search with GPUs. IEEE Transactions on Big Data, 7(3), 535-547.

[14] Milvus. (n.d.). Milvus Documentation. Retrieved from https://milvus.io/docs/ivf-flat.md

[15] Milvus. (n.d.). Milvus Documentation. Retrieved from https://milvus.io/docs/ivf-sq8.md

[16] Milvus. (n.d.). Milvus Documentation. Retrieved from https://milvus.io/docs/ivf-pq.md

[17] Luo, C. a. (2019). LSM-based storage techniques: a survey. The VLDB Journal, 29(1), 393-418.

[18] Mishra, S. (2024). A survey of LSM-Tree based Indexes, Data Systems and KV-stores. 2024 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS). Bhopal, India.

[19] Johnson, K. D. (2021). RedCaps: web-curated image-text data created by the people, for the people.

[20] Hervé Jégou, M. D. (2011). Product Quantization for Nearest Neighbor Search. IEEE Transactions on Pattern Analysis and Machine Intelligence, 33(1), 117-128.

[21] Rekabsaz, N. a. (2021). TripClick: The Log Files of a Large Health Web Search Engine. Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval.

[22] Elmasri, R., & Navathe, S. B. (2016). Fundamentals of database systems (7th ed.). Upper Saddle River, N.J.: Pearson Education.

[23] Ramakrishnan, R., & Gehrke, J. (2003). Database management systems (3rd ed.). Boston: McGraw-Hill.

[24] Comer, D. (1979). Ubiquitous B-Tree. ACM Computing Surveys (CSUR), 11(2), 121-137.

[25] Giampaolo, D. (1999). Practical File System Design with the Be File System. Morgan Kaufmann.

[26] SQLite. (2004). SQLite Version 3 Overview. (SQLite) Retrieved from https://sqlite.org/version3.html

[27] CouchDB. (n.d.). The Power of B-trees. (CouchDB) Retrieved from https://guide.couchdb.org/draft/btree.html

[28] Amazon Web Services. (n.d.). Caching Best Practices. (Amazon Web Services) Retrieved from https://aws.amazon.com/caching/best-practices/

[29] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill.

[30] Estébanez, C. Y. (2014). Performance of the most common non-cryptographic hash functions. Software: Practice and Experience(44).

[31] Oracle. (n.d.). Oracle AI Vector Search User's Guide. (Oracle) Retrieved from https://docs.oracle.com/en/database/oracle/oracle-database/26/vecse/load-binary-vector-data-using-sqlloader-example.html

[32] CMake. (n.d.). CMake. Retrieved from https://cmake.org/features/

[33] Lohmann, N. (2013). JSON for Modern C++. Retrieved from https://github.com/nlohmann/json

[34] Jeffrey Pennington, R. S. (2014). GloVe: Global Vectors for Word Representation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Doha, Qatar.