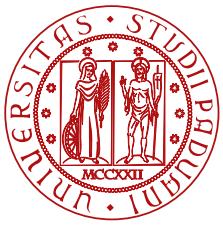


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

BACHELOR DEGREE IN COMPUTER ENGINEERING

Study and testing of the Falconn++ algorithm

Supervisor
Prof. Silvestri Francesco

Candidate
Zuech Riccardo

ACADEMIC YEAR 2022-2023

Dissertation date 21/11/2023

Abstract

The problem of nearest-neighbour search (NNS) is one of the most studied topics of data analysis, seeing application in numerous fields, e.g., recommendation systems, web search, machine learning, and computer vision.

There are several efficient solutions to address the problem of NNS in low dimensions, but to overcome the ”curse of dimensionality” and build efficient solutions for high dimensions, researchers have opted to use approximated approaches.

This thesis analyses Falconn++, a locality-sensitive filtering algorithm for the problem of approximate nearest-neighbour search on angular distance. The strength of Falconn++ lies in a filtering mechanism based on the theory of concomitants of extreme order statistics (CEOs) that achieves higher quality candidates compared to previous hashing-based solutions.

Sommario

La classe di problemi di nearest-neighbour search (NNS) è uno degli argomenti più studiati nell'analisi di dati, con applicazioni in numerosi ambiti quali recommendation systems, web search, machine learning e computer vision.

Esistono diverse soluzioni efficienti al problema della NNS in basse dimensioni, ma per costruire soluzioni che affrontino il problema in grandi dimensioni, superando così la "maledizione delle dimensioni", la ricerca si è diretta verso approcci approssimati.

In questa tesi analizziamo Falconn++, un algoritmo di locality-sensitive filtering per risolvere la classe di problemi di nearest-neighbour approssimato in distanza angolare. La caratteristica principale di Falconn++ è un meccanismo di filtraggio basato sulla teoria dei concomitanti dell'estremo ordine (CEOs) che garantisce candidati di miglior qualità rispetto alle precedenti soluzioni basate sull'hashing.

Contents

List of Figures

Chapter 1

Introduction

The nearest-neighbour problem can be defined as follows: Given a collection of n points, build a data structure that, when given a query point q, answers with the data point closest to q.

Solutions to this problem are a topic of interest, as we can represent features-rich objects (e.g., images, products, audio, etc.) as points in $\mathbb{R}^d$ and use a distance metric to measure the similarity between them. When we want the top-k closest data points to q the problem is also referred to as the k-nearest-neighbour search (k-NNS).

There are several efficient solutions to address the problem of NNS when the dimension d is low; however, they suffer from space complexity and/or query time being exponential in d, providing a small improvement over a brute-force linear scanning approach. This problem is referred to as "the curse of dimensionality".

In order to overcome "the curse of dimensionality", researchers opted to use approximation, i.e. the algorithm is allowed to report points that are not necessarily the nearest but whose distance is at maximum c times the distance from the query to its nearest points. In these cases, we talk about the problem of approximate nearest-neighbour search (ANNS).

Given the large number of application fields, we want an indexing scheme that is ideal, possessing the following properties:

- Accurate: the results of a query operation should be very close to the ones returned by the brute-force linear scan approach.
- Time-efficient: a query operation should take sublinear time in the number of objects, otherwise it cannot perform better than the brute-force solution
- Space-efficient: the data structure used for indexing should not require exponentially more space than the dataset itself; ideally, it should be linear in the size of the data set.

For approximate high-dimensional similarity search, one of the most popular indexing methods is locality-sensitive hashing (LHS).

Since we represent objects as points in vector spaces and use the distances between them as a similarity metric, we want to first spend some words on introducing the two principal distance measures:

- Euclidean distance: the l_2 norm of the difference between two points x,$y \in \mathbb{R}^d$, that is:

$$\|x - y\|_2 = \sqrt{\sum_{i=1}^{d} |x_i - y_i|^2}$$

  Points with a small Euclidean distance from one another are located in the same region of the space.

- Angular distance: the angle between two objects as viewed from an observer. Points with a small angular distance are located in the same general direction from the origin.

  The angular distance is related to the cosine similarity: given two points $x, y \in \mathbb{R}^d$, the cosine similarity corresponds to their dot product divided by the product of their magnitudes.

$$\text{Cosine similarity} := \cos\theta = \frac{x \cdot y}{\|x\|\|y\|}$$

  The relation between the angular distance and cosine similarity is as follows:

$$\text{Angular distance } D_\theta := \frac{\arccos(\cos\theta)}{\pi} = \frac{\theta}{\pi}$$

  When using the cosine similarity to calculate the angular distance, we perform the normalised dot product between x and y.

In this work, we focus on the problem of nearest-neighbour search on the unit sphere: given a set of points $X \subset S^{d-1}$ of size n and a query point $q \in S^{d-1}$, report the point $p \in X$ such that $p = argmin_{x \in X}\|x - q\|$.

The spherical case is important because, by placing ourselves on a unit sphere, the NNS on Euclidean distance becomes equivalent to the NNS on angular distance [1] and therefore we can formulate an equivalent definition of our problem where $p = argmax_{x \in X} x^\top q$.

It is important to keep in mind that placing ourselves on the unit sphere is just a useful simplification to overlap the Euclidean distance and the angular distance, but in practice we are not limited to points on a unit sphere, since we can always normalise them when strictly working on angular distance.

We now explore the fundamentals of locality-sensitive hashing and then analyse Falconn [2], an LSH-based approach for approximate nearest-neighbour search on angular distance. Lastly,

we introduce what Falconn++ offers to improve over Falconn [3].

1.1 Locality-sensitive hashing

Locality-sensitive hashing (LSH) is one of the most popular algorithms to perform an approximate nearest neighbour search in high dimensions, as it boasts a provable sublinear query time [4].

Given a distance function $dist(\cdot,\cdot)$, the definition of LSH is as follows:

Definition (LSH): For positive reals r, c, p_1, p_2 with $0 < p_2 < p_1 \leq 1, c > 1$, a family of functions H is called (r, cr, p_1, p_2)-sensitive if for uniformly chosen $h \in H$ and all $x, y, q \in \mathbb{R}^d$:

- if $dist(x, q) \leq r$ then $Pr_H[h(q) = h(x)] \geq p_1$
- if $dist(y, q) \geq cr$ then $Pr_H[h(q) = h(y)] \leq p_2$

In order to simplify the notation, from this point on we refer to x as a close point if $dist(x, q) \leq r$ and to y as a far away point if $dist(y, q) \geq cr$.

The basic LSH method processes a similarity search for a given query q by scanning all buckets to which q has been hashed and adding all elements from them to a candidate set. Then the candidate set is scanned to find the top-k closest points to q.

In order to minimise the probability of collision between far away points, a common approach is to concatenate $l = O(\log n)$ different LSH functions to form a new (r, cr, p_1^l, p_2^l)-sensitive LSH family. Unfortunately, this also reduces the probability of collision between close points. As a result, we need $L = (1/p_1^l)$ hash tables to guarantee a high probability of collision between close points.

Assuming $p_2^l = 1/n$ and $\rho = \ln(1/p_1)/\ln(1/p_2)$, it is necessary to use $L = O(n^\rho)$ tables for LSH to answer (c, r)-NN queries with constant probability by computing $O(n^\rho)$ distances in $O(dn^\rho)$ time and using $O(nd + n^{1+\rho})$ space [1].

It can be shown that the exponent ρ, governing the complexity of an LSH algorithm, must be $\rho > 1/c^2 - o(1)$ on the Euclidean distance given $p_2 \geq 1/n$ [5].

The main issue with LSH approaches is the resulting indexing space, as different queries require different values of r, hundreds of hash tables are needed to ensure that the query time is not affected.

1.2 Falconn

Falconn [2] is an LSH-based solution using a cross-polytope LSH family on angular distance, i.e. the cosine similarity. Its strength lies in achieving an optimal exponent $\rho \sim 1/c^2$ [6] and, thanks

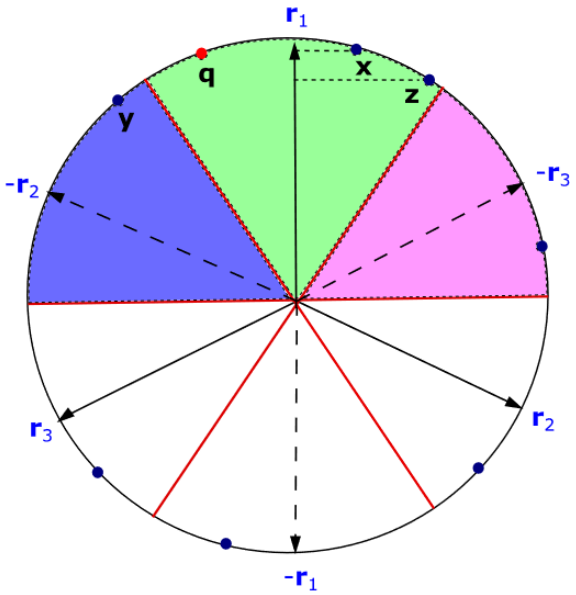


Figure 1.1: An illustration of Falconn with r_1, r_2, r_3. $h(x) = h(z) = h(q) = r_1, h(y) = -r_2$ [3]

to multi-probing, overcoming the bottleneck on indexing space of basic LSH-based solutions.

Given D random vectors $r_i \in \mathbb{R}^d, i \in [D]$ whose coordinates are randomly selected from the standard normal distribution $N(0,1)$ and $sgn(\cdot)$ as the sign function, the cross-polytope LSH family uses the index of the closest or furthest random vector as the hash value and, in particular, it uses cosine similarity to locate this vector.

Since we restricted ourselves to the unit sphere, w.l.o.g., we can assume that $r_1 = argmax_{r_i}\left|q^\top r_i\right|$ and then $h(q)$ is calculated as follows:

$$h(q) = \begin{cases} r_1, & \text{if } sgn(q^\top r_1) \geq 0 \\ -r_1, & \text{otherwise} \end{cases}$$

Essentially, the geometric intuition behind Falconn is that, given D random vectors, if $h(q) = r_1$ (or $h(q) = -r_1$) then q is closest (or farthest) to r_1, as shown in Figure 1.1. If x and q are similar, then they tend to be closest or farthest to the same random vector and, therefore, to be hashed into the same bucket corresponding to that random vector.

Since Falconn also uses the negatives of the D random vectors as hash values, it ends up with a total of $2D$ hash buckets for each hash function. Therefore, when using the concatenation trick discussed in Section 1.1, it builds a total of $(2D)^l$ buckets in each hash table.

1.2.1 Multi-probing

Multi-probing [7] is a method used to reduce the number of hash tables needed, and therefore the space occupation, while preserving time efficiency and high collision probability for close points.

The idea is to use a probing sequence to inspect multiple buckets that are likely to contain the nearest neighbours of a query q. Given the property of locality-sensitive hashing, if an object is close to q but not hashed in the same bucket as q, then it must be in a bucket that is "close by".

The probing sequence can be independent from the query (e.g. step-wise probing), or it can be derived from specific characteristics of the query based on how the hash value of q is computed.

Falconn uses the latter approach, considering the next closest or farthest random vectors to generate the probing sequence for each hash table [3]. Looking at Figure 1.1, since q is next closest to $-r_2$, the bucket corresponding to $-r_2$ will be the next probe. In particular, given a fixed total number $qProbes$ of query probes, Falconn ranks the Gaussian random vectors r_i by computing the distance between the projection values of q on them, therefore constructing the probing sequence. Taking Figure 1.1 as a reference, the ranking score of r_2 for probing is $|q^\top r_1| - |q^\top r_2|$.

1.3 Why Falconn++?

Falconn is an LSH-based approach to ANNS on angular distance with optimal ρ and reduced indexing space.

Unfortunately, we observe that multi-probing does not scale well on high-recall regimes. Since in real-world data sets the distance gaps between nearest neighbours are small, we need a very small approximation factor c. This requirement degrades performance as $\rho \to 1$ for small values of c. That means $qProbes$ and the number of candidates need to be increased, consequentially increasing the number of far away points in the probed buckets different from the query bucket.

To improve Falconn and, in general, any LSH-based solution for ANNS on angular distance, Falconn++ proposes a locality-sensitive filtering mechanism that can efficiently filter out far-away points in any bucket before querying. The proposed mechanism is also locality sensitive, as it filters out far-away points with high probability and close points with low probability.

In particular, Falconn++ uses the theory of concomitants of extreme order statistics to build a locality-sensitive filtering mechanism for angular distance on top of an approach similar to Falconn [3], [8].

Chapter 2

Concomitants of Extreme Order statistics

The Concomitants of Extreme Order statistics (CEOs) is a dimensionality reduction technique born in the context of the problem of approximate maximum inner product search (MIPS) [8]. The maximum inner product search problem is defined as follows: Given a data set $X \subset \mathbb{R}^{d \times n}$ of size n and a query point $q \in \mathbb{R}^d$, report the point $p \in X$ that has the maximum inner product, i.e. $p = argmax_{x \in X} x^\top q$.

Building on the theory of the concomitants of extreme order statistics [9], the CEOs method uses few projections associated with the extreme values of the query signature to estimate inner products.

Intuitively, the MIPS problem has many points of connection to the NNS problem on angular distance; in the next chapter, we indeed see how Falconn++ uses the CEOs property to improve on previous LSH-based solutions for NNS.

In this chapter, we introduce the building blocks of the CEOs technique for MIPS and provide a high-level intuition on the method; then we also introduce the points of connection with Falconn and in the next chapter we finally show how it is used in Falconn++.

2.1 Gaussian random projections

When talking about random projections, we refer to a technique used to reduce the dimensionality of a set of points in the Euclidean space. As a dimensionality reduction technique, it is a building block for CEOs, and therefore also Falconn++.

Given a set of points $X \in \mathbb{R}^{d \times n}$ and a Gaussian random matrix $R \in \mathbb{R}^{D \times d}$, whose elements are randomly sampled from the normal distribution $N(0,1)$, we define the signature of $x \in X$ as the projection of x on the random vectors of the matrix, calculated by Rx.

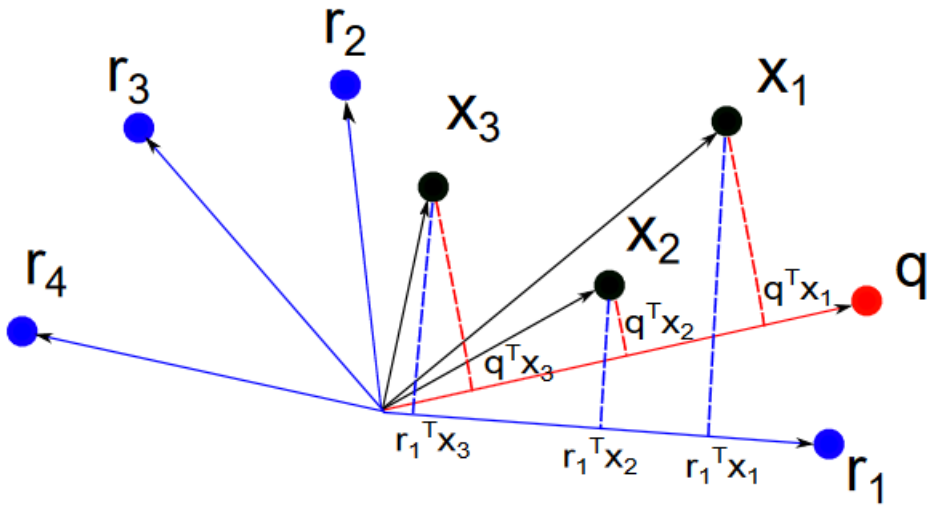


Figure 2.1: A geometric intuition of CEOs: Among 4 Gaussian random vectors r1, r2, r3, r4, the projections on r1 and r4, the closest and furthest vectors to q, preserve the most inner product order and inverse order, respectively [8].

2.2 CEOs for MIPS

The CEOs method is built on top of Gaussian RP, as we still apply D Gaussian random projections to compute the signatures for data and queries. On a high-level intuition, the key difference lies in using only a subset $s \ll D$ of projections to estimate inner products. In particular, this subset s of random projections is associated with the sth extreme values of the query signature.

Looking at Figure 2.1, we get a quick intuition about CEOs: among D random vectors, the closest and farthest to the query q preserve the best inner product order between data points.

The strength of the CEOs method lies in the usage of only the small selection of s projections among the D projections of the data signatures (and the sign of these projections of the query signature), allowing us to precompute and rank inner product estimators for all data before querying. For example, in Figure 2.1, we can precompute and rank $x_j^\top r_1, 1 \leq j \leq 3$ and then simply return x_1 when we find r_1 as the closest random vector to q.

2.2.1 Concomitants of normal order statistics

Let $(Q_i, X_i), i = 1,...,D$ be independent pairs of variates, ordered by Q_i. Then Q_i is called the ith order statistics, while X_i is called the concomitant of the ith order statistics.

In particular, when studying the asymptotic behaviour of the concomitants as $D \to \infty$, we talk about the theory of concomitants of extreme order statistics.

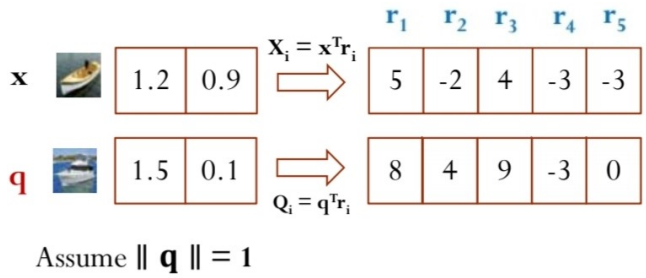


Figure 2.2: Projections of x and q on $D = 5$ random Gaussian vectors [10].

2.2.2 Concomitants of extreme order statistics and random projections

Since the theory of concomitants of extreme-order statistics is a complex topic, we support the theoretical discussion with a practical example: Imagine that we have a set of images with different subjects, e.g., aeroplanes, boats, dogs, etc., and we have a function that maps an image to a point in $\mathbb{R}^2$ based on the subject of the image; then, given a query image q, we can use a similarity metric, i.e. the inner product, to measure the similarity between the subject of q and the subjects of other images from the set. Our final goal is to obtain an inner product estimation process (that is, our similarity metric) between a query image q and another image x that does not depend directly on q, so we can use it to build an efficient MIPS solver. In a more formal way: suppose that we have a set of images represented as points in a 2-dimensional space and, given two points x, q from this set, our goal is to get an estimate of the inner product $x^\top q$, i.e., a similarity metric between image x and q.

Given D Gaussian random vectors $r_i \in \mathbb{R}^d$ and $x, q \in \mathbb{R}^d$, let $Q_i = q^\top r_i$ and $X_i = x^\top r_i$. For simplicity, assume $\|q\| = 1$.

Therefore, we have:

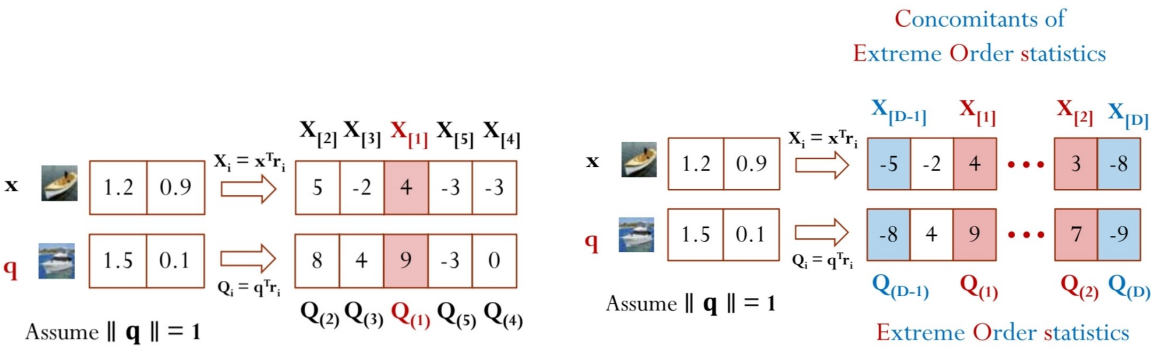
$$Q_i \sim N(0, 1), \quad X_i \sim N(0, \|x\|^2).$$

Essentially, we project x and q onto D Gaussian random vectors and obtain projection values as variates Q_i and X_i, Figure 2.2 describes this operation in our example with the set of images, where x and q are images of boats.

From the values of X_i, Q_i for each Gaussian random vector r_i, we construct D bivariate pairs (Q_i, X_i) from $N(0, 0, 1, \|x\|^2, \rho)$ where $\rho = x^\top q / \|x\|$.

Sorting the D pairs (Q_i, X_i) by the Q_i values, we form the order statistics, where $Q_{(1)}$ is the first (maximum) order statistics and $X_{[1]}$ is the concomitant of the first order statistics.

Figure 2.3a represents the situation in our example: we pair the projection values of q and x on the same random vector r_i together and then sort these pairs in descending order by the value of Q_i. We then call first-order statistics the $\max\{Q_i\}$ (which we refer to as $Q_{(1)}$ from now on) and concomitant of the first-order statistics the X_i (now $X_{[1]}$ as above) associated with



(a) $D = 5$: we form the order statistics. (b) $D \to \infty$: we consider the extreme order statistics.

Figure 2.3: Projection of x and q on D Gaussian random vectors [10].

$Q_{(1)}$, essentially the first pair in descending order contains the first-order statistics $Q_{(1)}$ and its concomitant $X_{[1]}$.

Then, when D is sufficiently large, $Q_{(1)}$ is the extreme order statistics and $X_{[1]}$ is the concomitant of the extreme order statistics.

It is provable that when $D \to \infty$ [8]:

$$E[Q_{(1)}] \xrightarrow{D} \sqrt{2 \log D}, \quad Var[Q_{(1)}] \xrightarrow{D} 0$$

$$E[X_{[1]}] \xrightarrow{D} x^\top q \sqrt{2 \log D}, \quad Var[X_{[1]}] \xrightarrow{D} \|x\|^2 - (x^\top q)^2$$

Since the Gaussian distribution has the property of being symmetric, we can both use $X_{[1]}$ and $-X_{[D]}$, respectively, the concomitant of the extreme order of the maximum $Q_{(1)}$ and minimum $Q_{(D)}$.

Furthermore, we can use more concomitants as they are asymptotically independent and normal [9]:

$$X_{[i]} \xrightarrow{D} N(x^\top q \sqrt{2 \log D}, \|x\|^2 - (x^\top q)^2) \quad 1 \leq i \leq s_0$$

And, as already said, by the Gaussian distribution symmetry, we can use both $X_{[i]}, -X_{[D-i+1]}$ where $1 \leq i \leq s_0$.

In our example, we are now in the situation described in Figure 2.3b: we have a total of $s = 2s_0$ pairs formed by the s_0 maximum extreme order statistics and the s_0 minimum extreme order statistics (and their concomitants).

To achieve our goal, we use those $s = 2s_0$ concomitants to compute an estimate of $x^\top q$

using the following unbiased estimator [8]:

$$\sum_{i=1}^{s_0} X_{[i]} - \sum_{i=1}^{s_0} X_{[D-i+1]}$$

It is important to note that the estimated value of the inner product $x^\top q$ does not use the value of the signature of q.

Recalling our example, this process allows us to get an estimate of the similarity between image x and image q; furthermore, as we see in the next section, this property is also a powerful tool that allows us, for example, to "precompute similarity" between images before querying.

2.2.3 An algorithm for MIPS

It is provable that, thanks to the properties of the concomitants of the s_0th extreme order applied to D random projection, we can precompute estimates of all n inner products for any query and achieve an optimal MIPS solver [8].

In particular, since the estimation process only uses the query signature order and not its value, after performing Gaussian RP on the data points we can precompute and rank all possible combinations of $s = 2s_0$ concomitants before querying.

From the above observations it also follows that the assumption of $\|q\| = 1$ made in Section 2.2.2 does not bear limitations on the CEOs property, since the estimator does not depend on the query signature value. Therefore, the CEOs property holds for the general inner product.

Eventually we end up with a $O(1)$ index lookup with recall guarantees: CEOs propose Algorithm 1 for indexing and Algorithm 2 for querying. [8, Section 3].

Algorithm 1 CEOs: Indexing [8, Algorithm 1]

1: **procedure** INDEXING(Data matrix $X_{d\times n}$, random Gaussian matrix $R_{D\times d}$)
2: Up-project X into D dimensions by computing $X' = RX$
3: For each pair of sets I, J, each containing distinct s_0 projection indexes among D up-projections and $I \cap J = \emptyset$, partially sort $\sum_{i\in I} X'_i - \sum_{j\in J} X'_j$ to add top-k indexes with the largest values to the list L_{IJ}.
4: **return** $O((D/s)^s)$ lists L_{IJ} where I, J correspond to the concomitants of the s_0 maximum and minimum order statistics

Let us introduce an example to give a better high-level understanding of how the algorithms proposed for the optimal MIPS solver work. Recalling the example from Section 2.2.2: suppose that we have a set of images represented as points in a 2-dimensional space $X \in \mathbb{R}^2$ and we want to perform a maximum inner product search on the data set for a certain query image $q \in \mathbb{R}^2$, that is, a similarity metric between images. The process is divided in two phases:

Algorithm 2 CEOs: Querying [8, Algorithm 1]

1: **procedure** QUERYING(query point q, Data matrix $X_{d\times n}$, random Gaussian matrix $R_{D\times d}$, index data structure 1)
2: Up-project q into D dimensions by computing $q' = Rq$.
3: Compute the sets I, J, each of s_0 projection indexes corresponding to s_0 maximum and minimum values of q'.
4: **return** the top-k points from the list L_{IJ}.

1. Indexing, Algorithm 1: build a data structure containing estimates of the top-k inner products for any possible future query.

2. Querying, Algorithm 2: return the precomputed top-k for that query from the data structure.

For the indexing phase, from Algorithm 1 line 2, we start by projecting the data points ($x \in X$) onto our D random Gaussian vectors r_i, obtaining $X_i = x^\top r_i$, where i is the index of the random Gaussian vector. Then, on line 3, for each possible set of $2s_0$ different indexes I and J (such that $I \cap J = \emptyset$), we use the CEOs property discussed in the previous section to precompute estimates of the top-k inner products for any valid pair (I, J) and save them in a data structure.

Recalling the content of Section 2.2.2, we precompute estimates for each possible pair (I, J) because I represents the set of random vector indexes associated with the s_0 maximum order statistics for the projection value of query q on the D Gaussian random vectors, while J represents the set of random vector indexes associated with the s_0 minimum order statistics. Essentially, indexes $i \in I$ are the s_0 maximum order statistics for $Q_i = q^\top r_i$, while indexes $j \in J$ are the s_0 minimum order statistics for $Q_j = q^\top r_j$. Therefore, since we precompute the top-k estimation values for each possible pair (I, J) and save them in a data structure, we can immediately obtain an estimate of the top-k inner product values for any query q. Algorithm 2 implements the querying phase using this process: It first up-projects the query point q on the D Gaussian random vectors to compute the pair (I,J), and then uses the pair (I, J) to return the associated top-k points from the data structures built in the indexing phase (Algorithm 1).

Look at Figure 2.4 (extending the example introduced in Section 2.2.2): with $s_0 = 2$, the specific pair $I = \{1, 3\}, J = \{2, D-1\}$ refers to the random vectors r_1, r_3 as the $s_0 = 2$ maximum order statistics for the projection values of q and to the random vectors r_2, r_{D-1} as the $s_0 = 2$ minimum order statistics. After computing the pair (I, J) for our query q, we can directly obtain the estimated value of the top-$k(k = 1)$ inner product $x_1^\top q = 2$, since we calculate it beforehand using the estimator from Section 2.2.2.

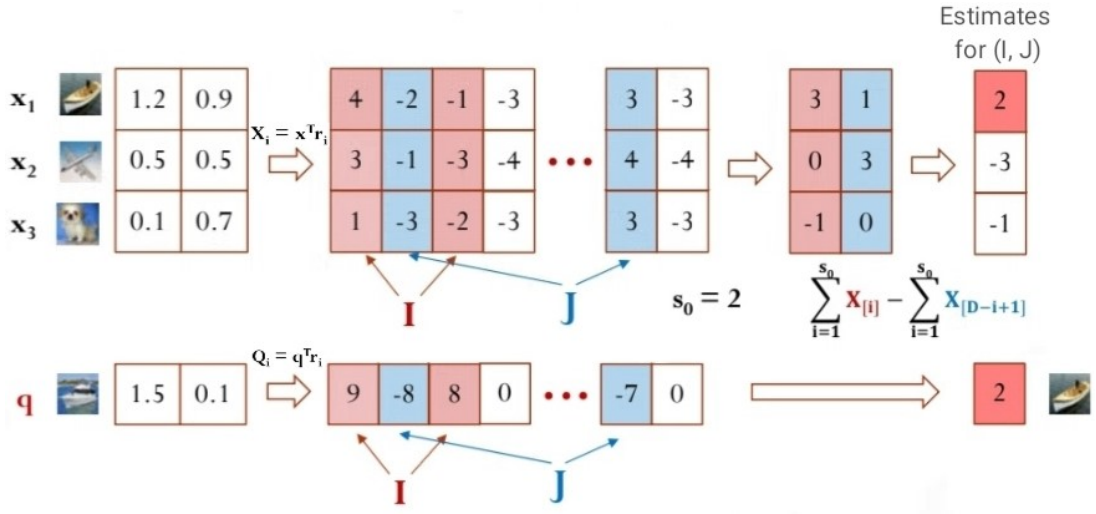


Figure 2.4: Example of application of the CEOs method on solving a MIPS problem [10].

2.3 Connection to Falconn

Keeping in mind that CEOs is a similarity estimation technique, rather than a hashing one, we can still see many points of connection with Falconn.

First of all, they both share the usage of random projections; in particular Falconn uses the top-1 closest or farthest random vector index as the hash value for a query q, while CEOs uses the projections values of the data set points on the closest and farthest s_0 random vectors to estimate the maximum inner product.

Another point of connection lies in the fact that Falconn's multi-probing likely probes the buckets corresponding to the $2s_0$ random vectors closest or farthest to q. Since the inner product order is well preserved on the projections onto these $2s_0$ vectors, applying the CEOs property, we can keep only a small fraction of number of points with the largest projection values in Falconn's buckets [3].

Referring to Figure 1.1, we have a high level intuition of the connection by observing the following:

- Falconn: If $r_1 = argmax_{r_i} \left| q^\top r_i \right|$, then:
  - use r_1 corresponding to $Q_{(1)}$ index as hash value
  - use $Q_{(i)}$ and $Q_{(D-i)}$ as probing buckets where $i = 1, ..., s_0$
- CEOs: If $r_1 = argmax_{r_i} \left| q^\top r_i \right|$, then:
  - use $X_{[1]} = x^\top r_1$ to estimate $x^\top q$
  - use $X_{[i]}$ and $X_{[D-i]}$ as estimators of $x^\top q$ where $i = 1, ..., s_0$

Essentially, Falconn++ builds on top of Falconn's LSH and then uses the the CEOs property to construct a locality-sensitive filtering mechanism to further save indexing space while still keeping a high recall ratio.

Finally, it is important to note that the CEO property works on the general inner product, as we use this observation in the next chapter.

Chapter 3

Falconn++

In the Introduction, we talked about how Falconn++, through usage of a locality-sensitive filtering mechanism, can improve on Falconn's approach, saving on indexing space and enhancing query performance.

While still using the same LSH technique of Falconn, on a high level the key difference is that Falconn++ applies the CEOs property to filter out far away points from each bucket before querying.

In this chapter, we discuss the concept of locality-sensitive filtering and the role of the CEOs property. Finally, we introduce Falconn++ as a combination of LSF with CEOs and the cross-polytope LSH of Falconn and discuss some practical add-ons that significantly improve performance.

3.1 Locality-sensitive filtering (LSF)

We call locality-sensitive filtering (LSF) a mechanism that can keep in any hash bucket close points with high probability and far away points with low probability. The definition of LSF is as follows [3]:

For positive $r, c, q_1, q_2 \in \mathbb{R}, 0 < q_2 < q_1 \leq 1, c > 1$, given x, y in the bucket $h(q)$ where h is randomly chosen from a LSH family, an LSF mechanism is a filter with the following properties:

- if $dist(x,q) \leq r$, then $Pr[x \text{ is not filtered}] \geq q_1$
- if $dist(y,q) \geq cr$, then $Pr[y \text{ is filtered}] \leq q_2$

Applying LSF to the buckets of any (r, cr, p_1, p_2)-sensitive LSH gives new collision probabilities between data points, e.g., x, y, and query points, e.g., q. In particular, we can state the following:

- $Pr[x \text{ is not filtered}, h(x) = h(q)] \geq q_1 p_1$
- $Pr[y \text{ is filtered}, h(y) = h(q)] \leq q_2 p_2$

On top of this assume that LSF has the property $\ln(1/q_1)/\ln(1/q_2) \leq \rho$ [3].

In the above statements lies the theoretical strength of Falconn++: after applying locality-sensitive hashing and filtering, we obtain a lower upper bound on the probability of collision between far away points ($q_2 p_2 < p_2$).

Obviously, this comes at a cost, as the lower bound on the probability of collision between close points has also been lowered ($q_1 p_1 \leq p_1$), forcing us to use a greater number of hash tables $O(1/q_1 p_1)$ as opposed to the standard LSH $O(1/p_1)$. In practice, the additional hash tables required do not take up more space when indexing, and actually space usage is improved: by applying our filter after hashing, we remove far away points from the buckets, and, as close points are often less than far away points, we end up having significantly fewer points in the buckets of each hash table.

In summary, thanks to the combination of LSH and LSF, we end up with a new ρ' such that:

$$\rho' = \frac{\ln(1/q_1 p_1)}{\ln(1/q_2 p_2)} \leq \rho,$$

resulting in improvements in query response time and indexing space usage over standard LSH.

3.2 CEOs property applied to LSF

The improvement over standard LSH methods, like Falconn, is not without its challenges, as our chosen LSF mechanism must have $\rho' \leq \rho \approx 1/c^2$ to be worth the trouble. By using the CEOs property to build its filtering mechanism, Falconn++ achieves this condition.

Given sufficiently large D random vectors r_i, without loss of generality (recall Section 1.2), we assume $h(q) = r_1 = argmax_{r_i} q^\top r_i$. Let $X = x^\top r_i, Y = y^\top r_i$ be random variables corresponding to x, y in the bucket $h(q)$.

From being in the unit sphere ($\|x\|, \|y\| = 1$) and the property of CEOs [8], we have:

$$X \sim N(x^\top q\sqrt{2\ln D}, 1 - (x^\top q)^2) \quad Y \sim N(y^\top q\sqrt{2\ln D}, 1 - (y^\top q)^2)$$

Exploiting this property, we obtain the following filtering mechanism [3]:

Given a threshold $t = (1 - r^2/2)\sqrt{2\ln D}$, in each bucket, we keep any point x if $x^\top r_1 \geq t$ and discard any point y if $y^\top r_1 \leq t$.

The following theorem shows that this filtering mechanism is locality-sensitive [3, Section 3.2 - Theorem 1]:

Given sufficiently large D random vectors and $c > 1$, the proposed filtering mechanism asymptotically has the following properties:

- if $\|x - q\| \leq r$, then $Pr[x \text{ is not filtered}] \geq q_1 = 1/2$
- if $\|y - q\| \geq cr$, then $Pr[y \text{ is not filtered}] \leq q_2 = \frac{1}{\gamma\sqrt{2\pi}} \exp(-\gamma^2/2) < q_1$ where $\gamma = \frac{cr(1-1/c^2)}{\sqrt{4-c^2r^2}} \cdot \sqrt{2 \ln D}$

Given a sufficiently large number D of random vectors, this LSF mechanism achieves $\ln(1/q_1)/\ln(1/q_2)$ arbitrarily smaller than $1/c^2$.

3.3 Falconn++

The Falconn++ algorithm descends from combining the cross-polytope LSH of Falconn with the LSF mechanism derived from the CEOs property. Essentially, Falconn++ performs hashing like Falconn, but for each bucket it filters out potentially far away points if their projection values are smaller than $t = (1 - r^2/2)\sqrt{2 \ln D}$.

From [3, Section 3.2 - Theorem 1], we have:

$$\ln(1/q_1) = \ln 2,$$

$$\ln(1/q_2) = \ln(\sqrt{2\pi}) + \ln \gamma + \gamma^2/2 \quad \text{where } \gamma = \frac{cr(1-1/c^2)}{\sqrt{4-c^2r^2}} \cdot \sqrt{2 \ln D}$$

For a large D, we have $\ln(1/q_2) \approx \gamma^2/2 = \frac{(1-1/c^2)^2}{4/c^2r^2-1} \cdot \ln D$. The new exponent ρ' of Falconn++ is strictly smaller than the exponent ρ of Falconn since:

$$\rho' = \frac{\ln(1/q_1p_1)}{\ln(1/q_2p_2)} \approx \frac{\frac{\ln 2}{\ln D} + \frac{1}{4/c^2r^2-1}}{\frac{(1-1/c^2)^2}{4/c^2r^2-1} + \frac{1}{4/c^2r^2-1}} \approx \frac{1}{1 + (1-1/c^2)^2} \cdot \frac{4/c^2r^2 - 1}{4/r^2 - 1} \leq \rho$$

In particular, for small c, $\rho \approx 1/c^2$ and $\rho' \approx 1/2(c^2 - 2 + 1/c^2)$.

Figure 3.1 reports a clear gap between Falconn and Falconn++ exponents.

3.3.1 Falconn++ and the unit sphere

Recalling the content of Chapter 1, placing ourselves on the unit sphere is an important special case, since the Euclidean and angular distance overlap.

However, by normalising the points Falconn++ solves the ANNS problem on angular distance even outside the unit sphere without any loss of accuracy. Obviously, the overlap between the Euclidean and the angular distances does not hold outside the unit sphere case.

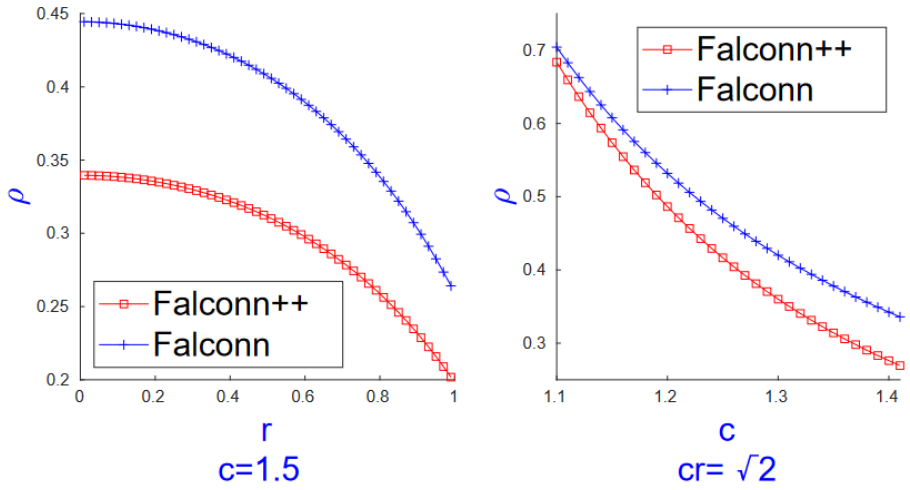


Figure 3.1: A comparison of ρ between Falconn++ and Falconn while fixing $c = 1.5$ and varying r; and fixing $cr = \sqrt{2}$ and varying c. [3]

3.4 Add-Ons

In this section, we discuss some practical implementations and tricks that improve the performance of Falconn++.

3.4.1 Scaling

When introducing Falconn++ and its filtering mechanism in the previous sections, we define a threshold $t = (1 - r^2/2)\sqrt{2 \ln D}$ that entails an improvement over Falconn in both the use of index space and the response time to queries. But, as we can see from both intuition and the formula, the threshold depends on the distance to the nearest neighbour r. This detail makes it difficult to find a global optimal t because different queries need different values for r.

In order to overcome the difficulty of selecting t, Falconn++ uses a scaling factor $0 < \alpha < 1$ to control the fraction of number of points in a bucket [3]. After hashing all the points from the data set, for any bucket of size B corresponding to the random vector r_i, we keep only the top-(αB) points with the larger projection values $x^\top r_i$ in the bucket.

As large buckets correspond to dense areas and smaller buckets to sparse areas around r_i, using a scaling factor α is conceptually similar to selecting different values of t in different density areas: larger dense buckets get more potentially far away points filtered out than smaller sparse buckets, mimicking t getting bigger for dense areas (which need small r) and smaller for sparse areas (which need large r). This makes the LSF mechanism of Falconn++ a data-dependent approach without a training phase.

Intuitively, we can argue that we do not need to scale small buckets since we risk losing

high-quality candidates. From this observation we place a limit on scaling, i.e. giving a lower bound k (as in top-k ANNS) on the number of points in each bucket. Obviously this entails increased space usage; however, it achieves a better recall-speed trade-off given the same space complexity configuration [3].

3.4.2 Multi-probing for indexing

Recalling from Section 1.2.1, multi-probing is based on the intuition that randomly perturbing a query point q generates many perturbed objects that tend to be close to q and, consequentially, be hashed into the same bucket with q's nearest neighbours [7], [11].

Obviously, this intuition is symmetrical: In the reverse case, when we perturb q's nearest neighbours, we generate perturbed objects that tend to be hashed into q's bucket. This property can be used for indexing by hashing each data set point into multiple "close" buckets, potentially improving the query performance.

One downside to applying this technique is pretty obvious: let $iProbes$ be the indexing equivalent to $qProbes$, the size of each hash table grows from n to $iProbes \cdot n$, and therefore this technique is usually useless in practise. Additionally, we recognise another downside of this technique in the intuition that high values of $iProbes$ dampen the accuracy of the buckets by adding points that are increasingly far away.

However, this technique of multi-probing for indexing shines when applied to Falconn++, which has the capacity to rank and filter out points in its buckets thanks to the CEOs property.

Falconn++ can use the technique to improve the precision of its query responses without impacting its indexing space usage, as far away points are filtered out. It is easy to note that the amount of space used remains unchanged: after projecting a point x onto the D Gaussian random vectors, we add x into the $iProbes$ buckets corresponding to the top-$iProbes$ closest or furthest random vectors r_i to x and then further scale each bucket by $iProbes$ times, this way keeping the number of points in each hash table to αn[3]. Actually, the situation is trickier when we apply the limit on scaling technique, and in Chapter 4 we discuss the relation between them.

However, we can recognise a limit to the improvement in query performance achieved by increasing $iProbes$: After a certain threshold, the points probed for indexing are likely to be added to farther and farther buckets, eventually being filtered out with high probability. In Chapter 4 we see a practical experiment that highlights this behaviour.

In Algorithm 3 we can see a pseudo code implementation of the technique in Falconn++.

Algorithm 3 Multi-probing for indexing in Falconn++ [3]

1: **procedure** MULTI-PROBE INDEXING(X, α, D random vectors r_i, $iProbes$)
2: **for** each $x \in X$ **do**
3: Compute top-$iProbes$ vectors r_i s.t. $\left|x^\top r_i\right|$ is largest.
4: Add x to the $iProbes$ buckets corresponding to these $iProbes$ vectors.
5: For each bucket of size B corresponding to r_i, keep top-$(\alpha B/iProbes)$ points with the largest absolute projection values on r_i.

3.4.3 Multi-probing for querying

In Section 1.2.1, we see how Falconn builds a query-dependent probing sequences for multi-probing. In Chapter 2 we see the connection between Falconn and the CEOs technique and discuss how Falconn likely probes the $2s_0$ buckets corresponding to the random vectors r_i closest and furthest to the query q on each table.

Falconn++ in his multi-probing criteria for querying is quite similar to Falconn: it uses a query-dependent probing sequence and probes up to $2s_0$ buckets corresponding to the random vectors r_i closest and furthest to the query q. But, unlike Falconn, Falconn++ exploits the CEO property (which asymptotically holds for the $2s_0$ closest and furthest random vectors to q) by ranking the probing sequences based on $\left|q^\top r_i\right|$ where r_i is the random vector corresponding to the probing bucket [3].

3.4.4 Centering data points

As Falconn++ is an instance of Falconn, it does not perform well when data points are skewed on specific areas of the sphere; that is because most of the points will be hashed into the same few buckets. To avoid this occurrence, Falconn++ artificially centers the data points [3].

Taking $c = \sum_i x_i/n$ as the center of the data set X, we map $x_i - c$ to x_i' and therefore diverge X to the whole sphere, solving the problem.

The main doubt that might arise from using this trick comes from observing that $\|x_i'\| \neq 1$, but, as seen in Chapter 2, the asymptotic property of CEOs holds for the general inner product, and it is not restricted to the unit sphere [8]. Additionally, after centering, $argmax_{x \in X} x^\top q = argmax_{x' \in X} x'^\top q$ and, therefore, all the previous statements still hold.

Chapter 4

Experiments

For all of the experiments in this chapter, we use the C++ implementation of Falconn++, available at https://github.com/ninhpham/falconnlsf, on the University of Padova's "Algo" machine.

In particular, we want to test the characteristic features of the algorithm in three experiments by observing the following metrics:

- The trade-offs between recall values and query time.
- The trade-offs between recall values and queries answered per second.
- The memory usage of the data structure used.
- The indexing time.
- The average distance difference from the queries between Falconn++ topK and the true topK for different recall values.

For each experiment, we look at the results of n batches, where each batch represents 20 runs of the algorithm for a specific parameter selection with $qProbes = 1000i, i = 1,...,20$ ($qProbes$ governs the desired recall rate).

As our dataset we use Glove200, available at https://github.com/erikbern/ann-benchmarks/ and set $k = 20$ in all our experiments. To reduce noise, all results are the average of 3 runs of the algorithm, and for each run, we extract 1000 points at random from the test set to build our query set.

4.1 Parameters selection

There are many parameters that contribute to the performance of Falconn++: number of random vectors D, scaling factor α, number of tables L, number of indexing probes $iProbes$ and number of query probes $qProbes$ [3].

First of all, it is important to cite that the implementation of Falconn++ we use concatenates 2 LSH functions and, as each hash function returns $2D$ hash values, the total number of buckets on a hash table is $4D^2$. As we apply multi-probing for indexing, the number of points in each table is $n \cdot iProbes$. Additionally, we expect that the $iProbes$ points probed for indexing are equally distributed in the $4D^2$ buckets, so each bucket has $n \cdot iProbes/4D^2$ points.

Furthermore, recalling the content of Section 3.4.1, we keep $max(k, \alpha B/iProbes)$ points in each bucket.

As we want to have approximately k points in each bucket, the optimal heuristic proposed by Falconn++ is as follows: Set D and $iProbes$ so that $k \approx n \cdot iProbes/4D^2$ [3]

In order for the LSF property to hold, we also need a sufficiently large D: as we work on a high-dimensional data set with large d, $D \approx 2^{\lceil \log_2 d \rceil}$ suffices [3]. The intuition behind this formula is that approximately we need a random vector for each dimension of the data set; e.g., if we are working on Glove200, since $d = 200$, we want $D \approx 2^8 = 256$ as $\log_2(d) \approx 7.64$. Obviously, higher values of D and $iProbes$ achieve a better recall-query time trade-off by sacrificing memory usage, due to the increased size of the hash tables.

In terms of the scaling factor α, thanks to the scaling limit trick, the best performance is obtained with $\alpha = [0.001, 0.1]$ (converted to percentage $0.1 = 10\%$)[3].

It is interesting to note that, when $\alpha = 1$ and $iProbes = 1$, essentially Falconn++ regresses to Falconn.

Finally, the values of L and α are used to govern the desired memory usage, while the role of $qProbes$ is to obtain the desired recall values.

4.2 Preparations

Before we focus on our experiments, we want to spend some words on the steps needed to run them:

1. We convert the Glove200 data set from the .hdf5 format to .txt using the Python module "h5py", by doing so we obtain a train set and test set in $\mathbb{R}^{200}$.

2. The train and test sets must be normalised, as the implementation of Falconn++ we use works on the unit sphere.

3. We need to extract 3 query sets of 1000 random points from the normalised test set, one for each run of the experiments.

4. We need to compute the top-$k = 20$ neighbours for each of our query sets' points using the angular distance.

After performing the above steps, we have everything we need to proceed with our analyses. All other instructions to run the implementation of Falconn++ used are available in the relative repository.

Finally, we want to highlight a tricky notation we later use: when we show a batch's parameter settings, α is expressed as a percentage, so $\alpha = 1$ means that we keep in our buckets only the top 1% points (that is $\alpha = 0.01$ in the usual notation). This is motivated by the fact that the implementation we use of Falconn++ needs an integer value as the scaling factor α, that is, the percentage of points kept in each bucket.

4.2.1 Recall value

For each query operation, the recall value is defined as:

$$\frac{\text{\#top-}k \text{ neighbours returned by Falconn++ } \cap \text{ \#true top-}k \text{ neighbours}}{k}$$

In our experiments, we want to observe the query performance by plotting the response time to the query as a function of the recall value. The recall value is the independent variable since, in practical applications, we want to achieve a specific recall rate while choosing a parameters selection that gives us the best query performance (taking into consideration space usage, too).

As mentioned in Section 4.1, for each experiment, we vary $qProbes$ from 1000 to 2000 to obtain increasing recall values; as $qProbes$ increases, so does the recall value (the more buckets we probe, the higher the number of candidate points and the more likely it is to find the true top-k nearest neighbours).

The query performance of each batch is given by the recall-time trade-off for that parameters selection.

4.3 Experiment 1: testing the bounds on the optimal heuristics for parameters selection

In this first experiment, we want to test the performance of the algorithm when varying D while maintaining the optimal heuristic $k \approx n \cdot iProbes/4D^2$.

In order to do so, we fix the number of hash tables $L = 350$ and the scaling factor $\alpha = 0.01$, according to the optimal range proposed in Section 4.1 and use the following 4 batches of parameter settings:

- Batch1: $L = 350, \alpha = 1, iProbes = 1, D = 2^6 = 64$
- Batch2: $L = 350, \alpha = 1, iProbes = 1, D = 2^7 = 128$

- Batch3: $L = 350, \alpha = 1, iProbes = 4, D = 2^8 = 256$
- Batch4: $L = 350, \alpha = 1, iProbes = 17, D = 2^9 = 512$

Keep in mind that, when we show a batch's parameter settings, α is expressed as a percentage.

We predict that higher values of D give better recall-time trade-offs, as a higher number of Gaussian random vectors translates to closer points in each bucket. Additionally, since we maintain the heuristic with $iProbes \approx 4kD^2/n$, we expect to observe the advantages discussed in Section 3.4.2 for $iProbes > 1$.

On top of that, we foresee that an optimal value for D is approximately equal to or greater than the number of dimensions of the points in the dataset, as proposed by the heuristic $D \approx 2^{\lceil \log_2 d \rceil}$ in Section 4.1.

Obviously, we expect to observe a notable increase in space usage, since the total number of buckets in each hash table is $4D^2$ and we use $iProbes$ to maintain the optimal heuristic of having $\approx k$ points in each bucket.

4.3.1 Results discussion

Figures 4.1, 4.2 and 4.3 report the results of our experiments on varying D while maintaining the optimal heuristic with $iProbes$.

Figure 4.1 highlights that, as expected, higher values of D significantly increase the query performance of Falconn++, particularly in Batch3 and Batch4 where $D \geq 2^{\lceil \log_2 d \rceil}$ and where we get the advantages of using multi-probing for indexing. This point is also proven by 4.3, as we can see that higher values of D make the average distance difference of the query points from the true top20 and Falconn++ top20 smaller and smaller.

Figure 4.2(c) shows the drastic growth in space usage that we expected by increasing D, which is motivated by the interaction between the number of buckets in each hash table being $4D^2$, the higher values of $iProbes$ used to maintain the optimal heuristic and the limit on scaling. The important intuition to take is that we can recognise positive feedback caused by the limit on scaling trick. In each hash table, we have a total of $n \cdot iProbes$ points evenly distributed in $4D^2$ buckets (recall Section 4.1). By scaling the buckets, we counterbalance the additional space usage of multi-probing for indexing. However, since we limit scaling to k and maintain the optimal heuristic to have $\approx k$ points in each bucket, we instead observe a significant increase in space usage for higher values of D due to the significantly higher number of buckets in each hash table. In the next experiments, we further discuss the implications of the limit on scaling trick.

From the above intuition, we clearly see the reason for the drastic increase in the size of the data structure from Batch1 to Batch4: we have a small enough $\alpha = 0.01$ to keep the majority

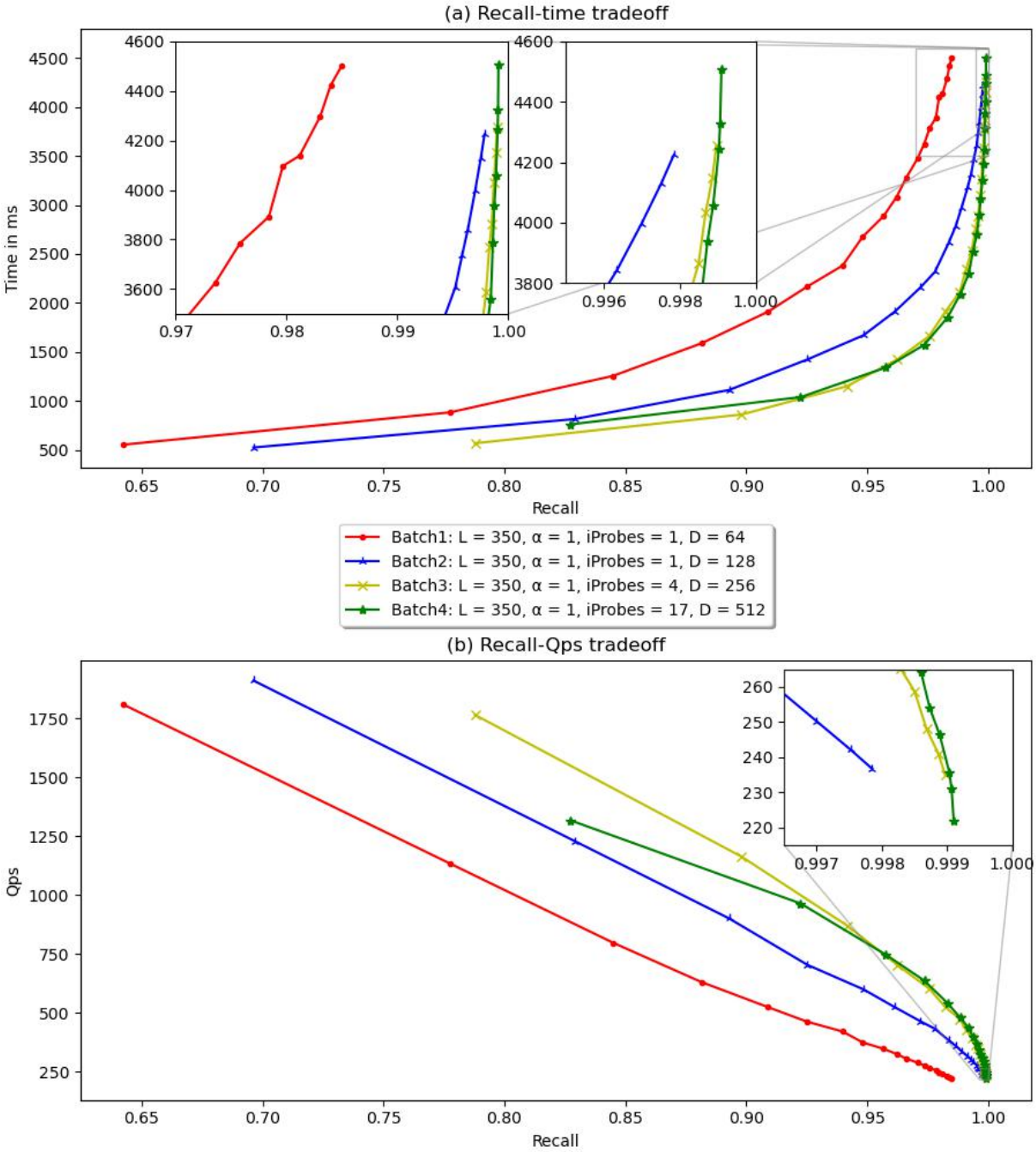


Figure 4.1: Performance of experiment 1: varying D while maintaining the optimal heuristic with $iProbes$.

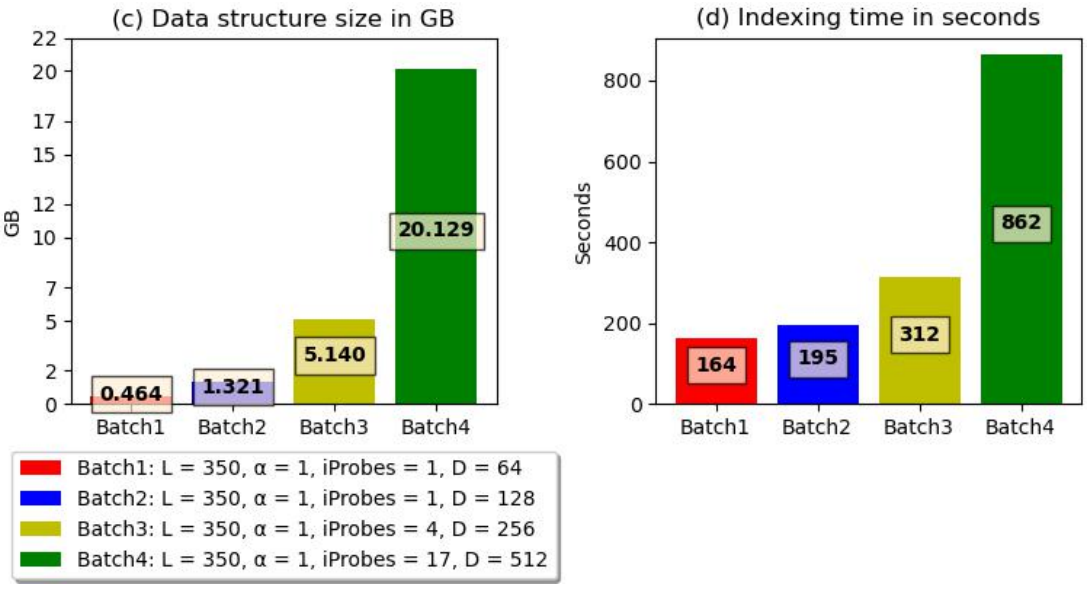


Figure 4.2: Data structure size and indexing time for experiment 1: varying D while maintaining the optimal heuristic with $iProbes$.

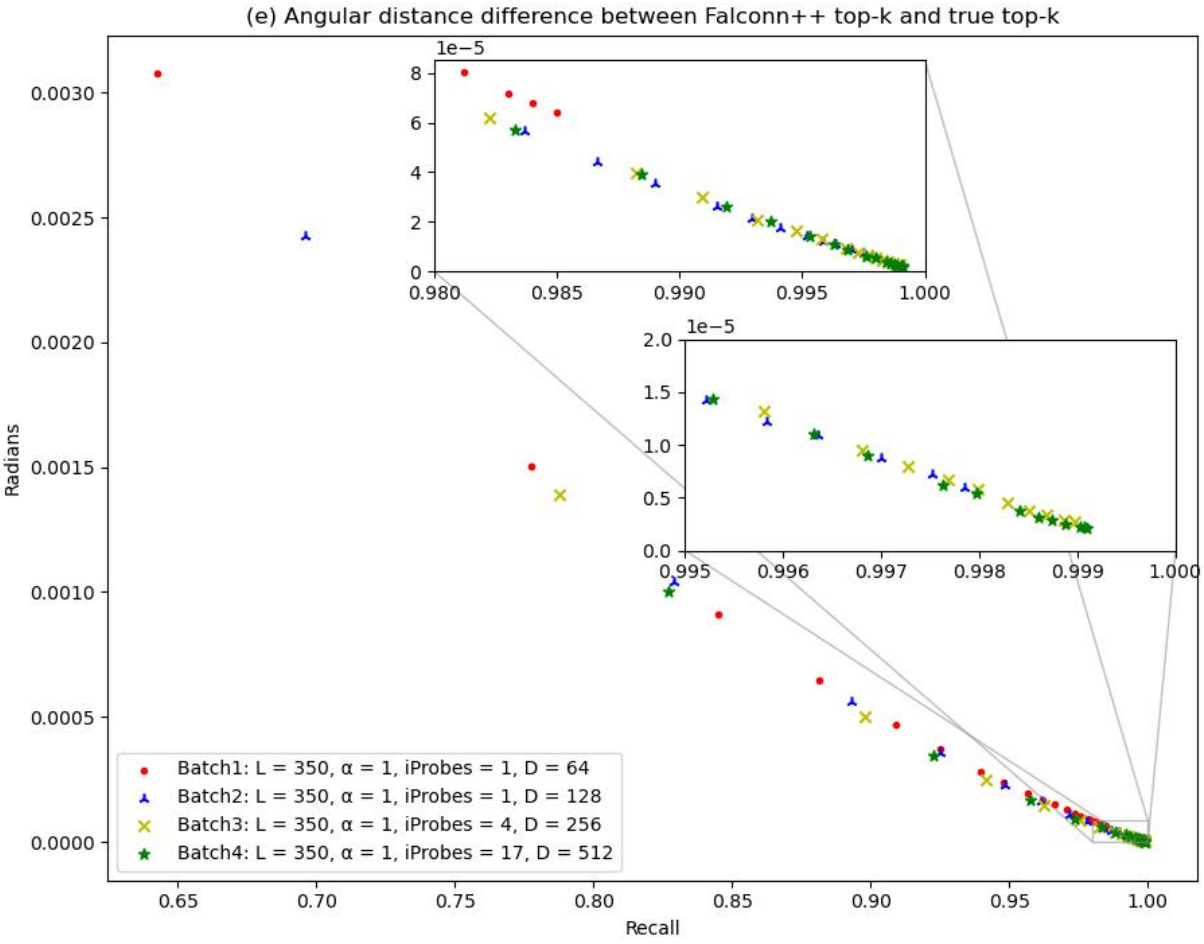


Figure 4.3: Distance difference for experiment 1: varying D while maintaining the optimal heuristic with $iProbes$.

of the buckets on the threshold of k for each batch and since we maintain the heuristic when we increase D, we end up with a significantly larger number of buckets of size $\approx k$.

Furthermore, Figure 4.2(d) also shows a notable increase in the time required to build the data structure, whose origin can be explained by the need to build significantly more buckets in each hash table (recall that each bucket represents a random Gaussian vector).

4.4 Experiment 2: testing the limit on improvement by increasing the number of indexing probes

In this second experiment, we want to test the limit on performance improvement provided by increasing the indexing probes up a certain threshold, as discussed in Section 3.4.2.

As before, we fix the number of hash tables $L = 350$ and the scaling factor $\alpha = 0.01$, according to the optimal range proposed in Section 4.1. We now use the following 4 batches of parameter settings:

- Batch1: $L = 350, \alpha = 1, iProbes = 2, D = 256$
- Batch2: $L = 350, \alpha = 1, iProbes = 4, D = 256$
- Batch3: $L = 350, \alpha = 1, iProbes = 7, D = 256$
- Batch4: $L = 350, \alpha = 1, iProbes = 10, D = 256$
- Batch5: $L = 350, \alpha = 1, iProbes = 1, D = 256$

As always, keep in mind that, when we show a batch's parameter settings, α is expressed as a percentage.

In this experiment, we expect to observe a performance improvement threshold when increasing the number of indexing probes over the value proposed by the optimal heuristic.

We also expect to observe an increased size of the data structure as $iProbes$ increases, for two intertwined reasons:

- We scale each bucket by α, but as we add more and more points to the buckets due to multi-probing for indexing while keeping α fixed, we end up with more total points in the data structure than with lower values of $iProbes > 1$.
- The limit on scaling trick makes it so that buckets of size smaller than k do not get scaled, so points probed for indexing in these small buckets do not get scaled.

It is interesting to note that these considerations refer specifically to practical implementations using the scaling factor α; for example, if we were to use the threshold t (Section 3.2), we might end up with an even smaller increase in space usage (or none at all). However, this is just an interesting theoretical intuition, as in Section 3.4.1 we see the major problems of using the threshold method over the scaling one.

Additionally, we use Batch5 with $iProbes = 1$ as a meaningful comparison, since we expect to observe an improvement in query performance by using the multi-probing for indexing technique.

4.4.1 Results discussion

Figures 4.4, 4.5, 4.6 report the results of our experiments on varying $iProbes$.

In particular, Figure 4.4 proves the following points:

- Using the multi-probing for indexing technique improves the query performance of Falconn++ in general. This follows by comparing the recall - time and queries per second trade-offs between Batch5 with $iProbes = 1$ and the other batches.

- There is a limit to the query performance improvement given by increasing $iProbes$. We see that Batch 1 and Batch2 achieve higher recall rates than Batch3 and, in particular, Batch4. Additionally, in Batch4 ($iProbes = 10$) the algorithm still achieves better recall values than Batch5 ($iProbes = 1$) in general, but performs worse than Batch3 ($iProbes = 7$) and the optimal Batch2 ($iProbes = 4$) and Batch1 ($iProbes = 2$). It is interesting to observe the comparison between Batch2 and Batch1, as both $iProbes$ values can be considered optimal: we observe that Batch2, with $iProbes = 4$ on the higher end of the optimal range, achieves better overall recall rate trade-offs, but asymptotically falls off on high recall regimes (when $qProbes$ assumes higher values) to Batch1, with $iProbes = 2$ on the optimal lower end.

Additionally, Figure 4.6 also proves these points, as Batch5 without multi-probing for indexing achieves the worst mean distance from the query between returned top20 and true top20, while Batch1 and Batch2 with the optimal values achieve the best one.

In Figure 4.5(c) we observe the expected increase in the size of the data structure. However, this increase is contained, especially compared to Experiment 1. This behaviour is explained by the considerations done above. To further deepen the intuition, let us consider the following: As we fix D and L, the number of hash tables and buckets for each table remains constant, and using a small value of $\alpha = 0.01$, we expect a large number of buckets to be scaled to size k. Then, as the number of indexing probes increases, so does the number of points in each bucket, and by the limit on scaling, the smaller buckets get progressively bigger (relatively more than

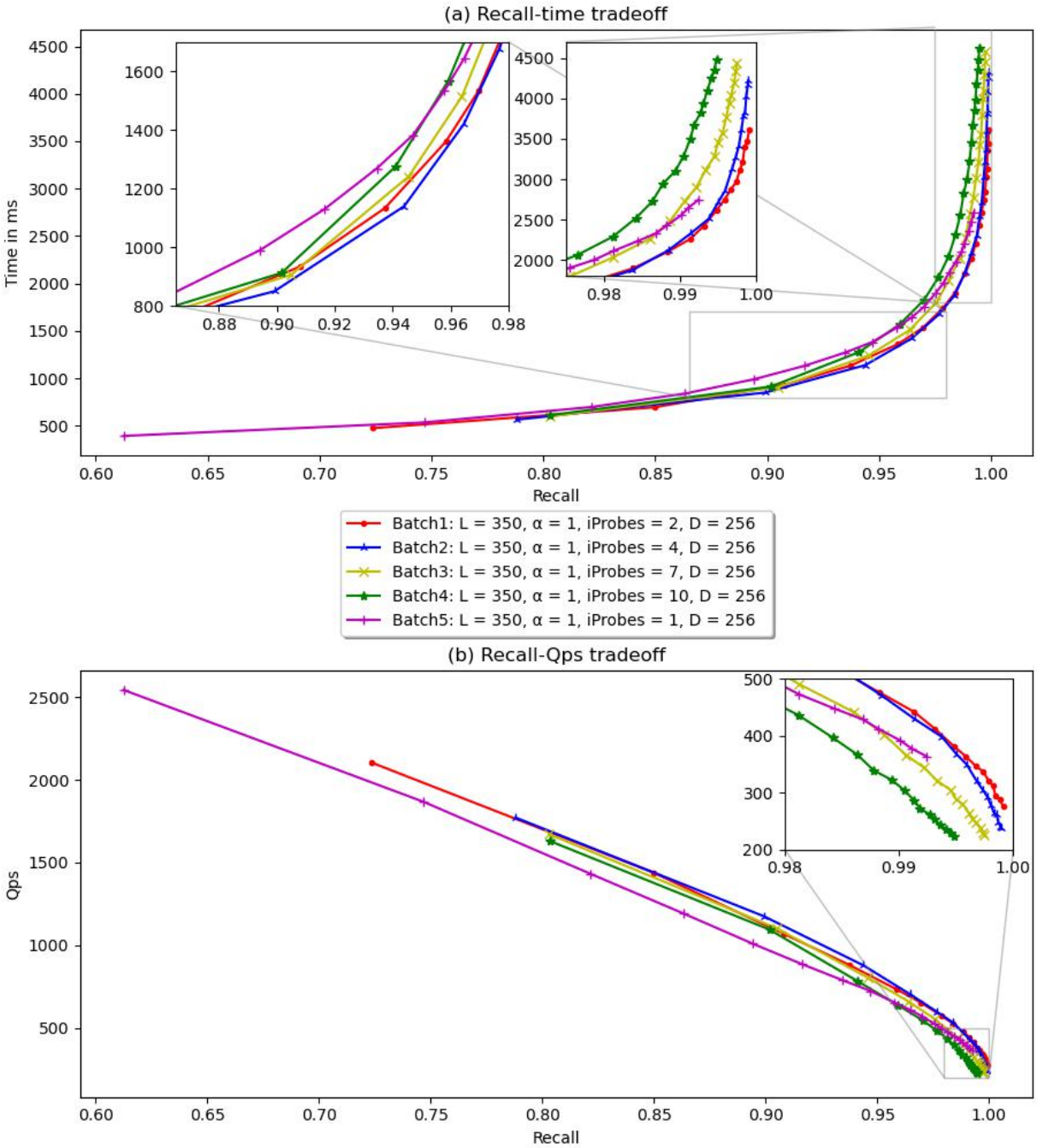


Figure 4.4: Performance of experiment 2: increasing the value of $iProbes$

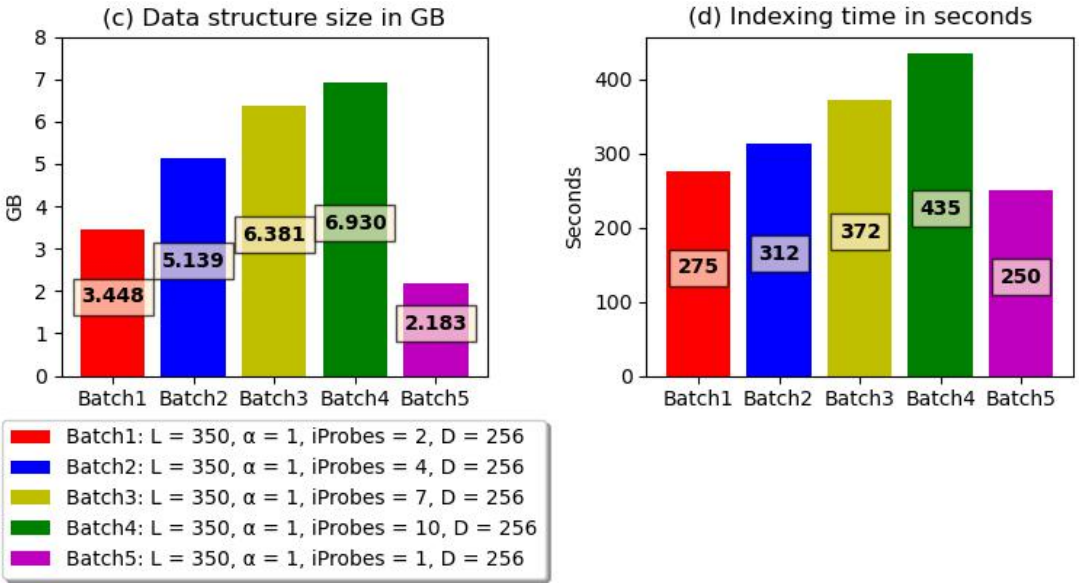


Figure 4.5: Data structure size and indexing time for experiment 2: increasing the value of $iProbes$

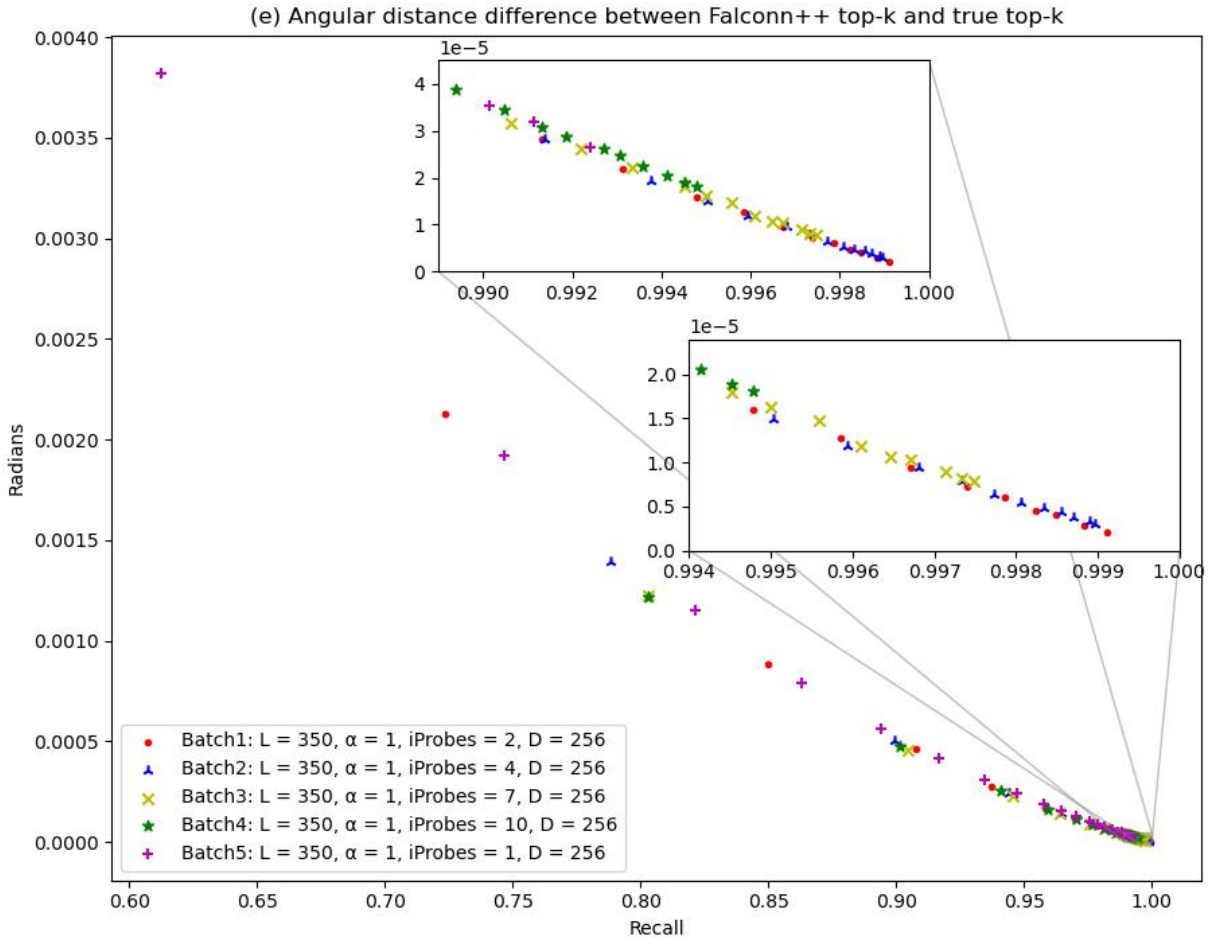


Figure 4.6: Distance difference for experiment 2: increasing the value of $iProbes$

the larger buckets that get scaled). Essentially, we see an increase in space usage by increasing $iProbes$ as the buckets become larger and larger, but this increase is contained as the small value of the scaling factor α counter-balances the growing number of points on larger buckets.

Figure 4.5(d) shows that higher values of $iProbes$ do not drastically impact the indexing time, suggesting that multi-probing for indexing is not a high time-consuming operation; comparing the results with Experiment 1 Figure 4.2(d), we do not see such a spike in indexing time usage as when increasing the value of D.

4.5 Experiment 3: testing the impact of the limit on scaling

In this last experiment, we focus on the scaling factor α and its relations with the number of hash tables L and with the multi-probing for indexing technique.

This time, we fix the number of random Gaussian vectors to $D = 256$, according to the optimal range proposed in Section 4.1, and use the following 5 batches of parameter settings:

- Batch1: $L = 100, \alpha = 13, iProbes = 4, D = 256$
- Batch2: $L = 100, \alpha = 50, iProbes = 1, D = 256$
- Batch3: $L = 250, \alpha = 20, iProbes = 1, D = 256$
- Batch4: $L = 500, \alpha = 10, iProbes = 1, D = 256$
- Batch5: $L = 500, \alpha = 3, iProbes = 4, D = 256$

As always, when we show a batch's parameter settings α is expressed as a percentage.

These batches are selected with the following relation in mind: $\alpha(iProbes \cdot L) = 50$, that follows from these two observations:

1. Applying the scaling factor α to each bucket is equal to scaling each whole hash table. If each hash table has n points in it, after scaling it has $n' = \alpha n$ points.
2. By using the multi-probing technique for indexing, each point is hashed into $iProbes$ buckets instead of just one, e.g., when $iProbes = 4$ one actual hash table can be seen as 4 hash tables merged compared to Falconn.

We expect a better query performance as L increases and α decreases: having more hash tables translates to a higher collision probability for close points, and lower α values further scale the buckets to only keep the very closest points in them. Additionally, a significant performance increase is expected in particular in Batch1 and Batch5, as combining multi-probing for indexing with scaling leads to higher-quality candidates for each query.

From the two statements above, we would expect to observe an approximately constant space usage through the batches. However, we actually predict an increase in space usage caused by applying the limit on scaling trick. Recalling Section 3.4.1, the limit on scaling technique prevents small buckets, of size smaller than the desirable $k = 20$, from being scaled and thus losing high-quality candidates. Theoretically speaking, without the limit, we could obtain an approximately constant space usage throughout the 5 batches. However, our final goal is always the best query performance possible, which is obtained by having buckets with at least k points.

Taking into consideration the scaling limit trick, we expect to observe the following behaviours:

1. Through batches 2 to 4 the space usage is increased by the increasing number of hash tables L: as the number of hash tables grows, so does the number of small buckets that do not get scaled.

2. Between Batch1 and Batch2 the space usage increases as, on average, the multi-probing for indexing technique makes the buckets that do not get scaled bigger by filling them with points probed for indexing. The same behaviour is expected between Batch 5 and Batch4. Recalling the content of Section 4.4 it is interesting to highlight that in Batch1 and Batch5, as we increase $iProbes$, we also decrease α, this makes it so that the effect of multi-probing for indexing gets balanced on the bigger buckets (that do get scaled) by a stricter scaling.

3. When comparing Batch1 to Batch5, we expect a more marked spike in space usage from Batch2 and Batch4, respectively. This is motivated by the observation that, since Batch4 and Batch5 use $L = 500 >> 100$, the number of small buckets that do not get scaled is significantly higher.

4.5.1 Results discussion

Figures 4.7, 4.8 and 4.9 report the results of our last experiment on the scaling factor.

First of all, in Figures 4.7 and 4.9, we can observe that, as predicted, the query performance increases significantly as the number of hash tables L grows and the scaling factor α decreases. The query performance improvement is particularly accentuated in Batch1 and Batch5, where we apply the technique of multi-probing for indexing. Batch1 in high-recall regimes performs significantly better than batches 2 to 4 with a higher number of hash tables, and Batch5 performs better than the others overall.

Then from Figure 4.8(c) we observe the expected behaviours in the size of the data structure: the limit on scaling trick makes the space usage grow with L and $iProbes$.

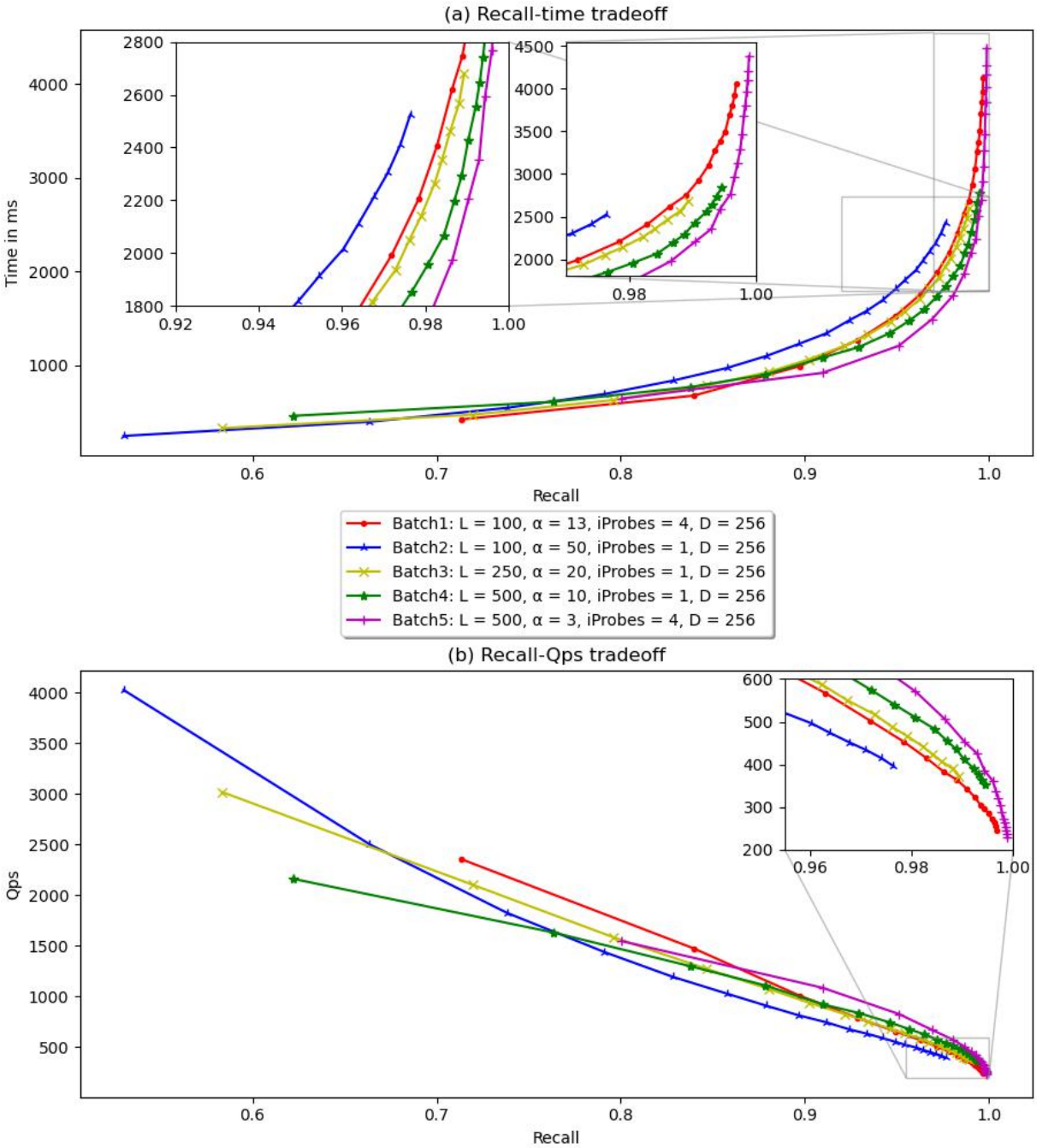


Figure 4.7: Performance of experiment 3: testing the impact of the limit on scaling.

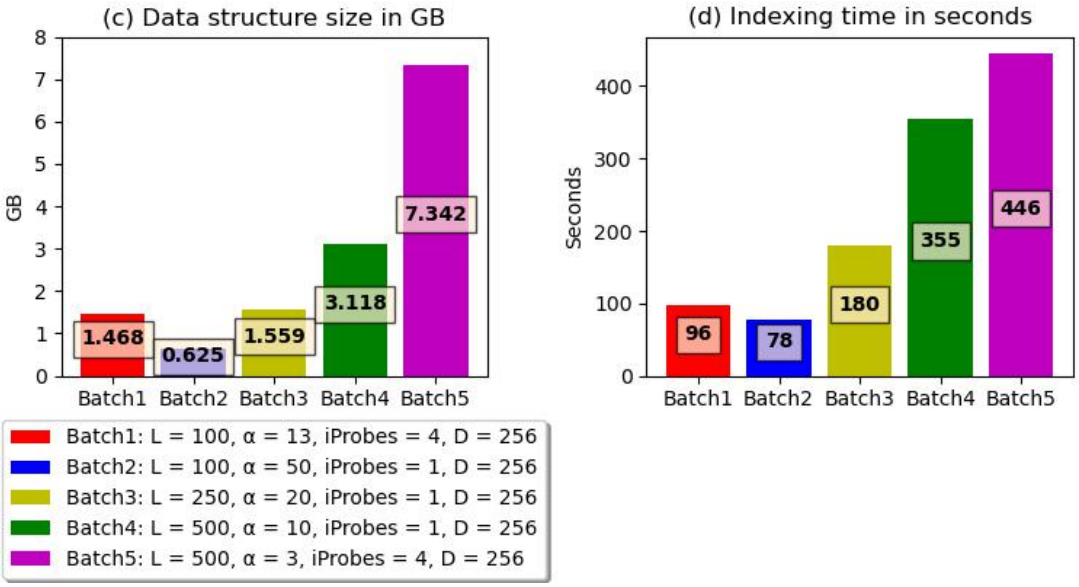


Figure 4.8: Data structure size and indexing time for experiment 3: testing the impact of the limit on scaling.

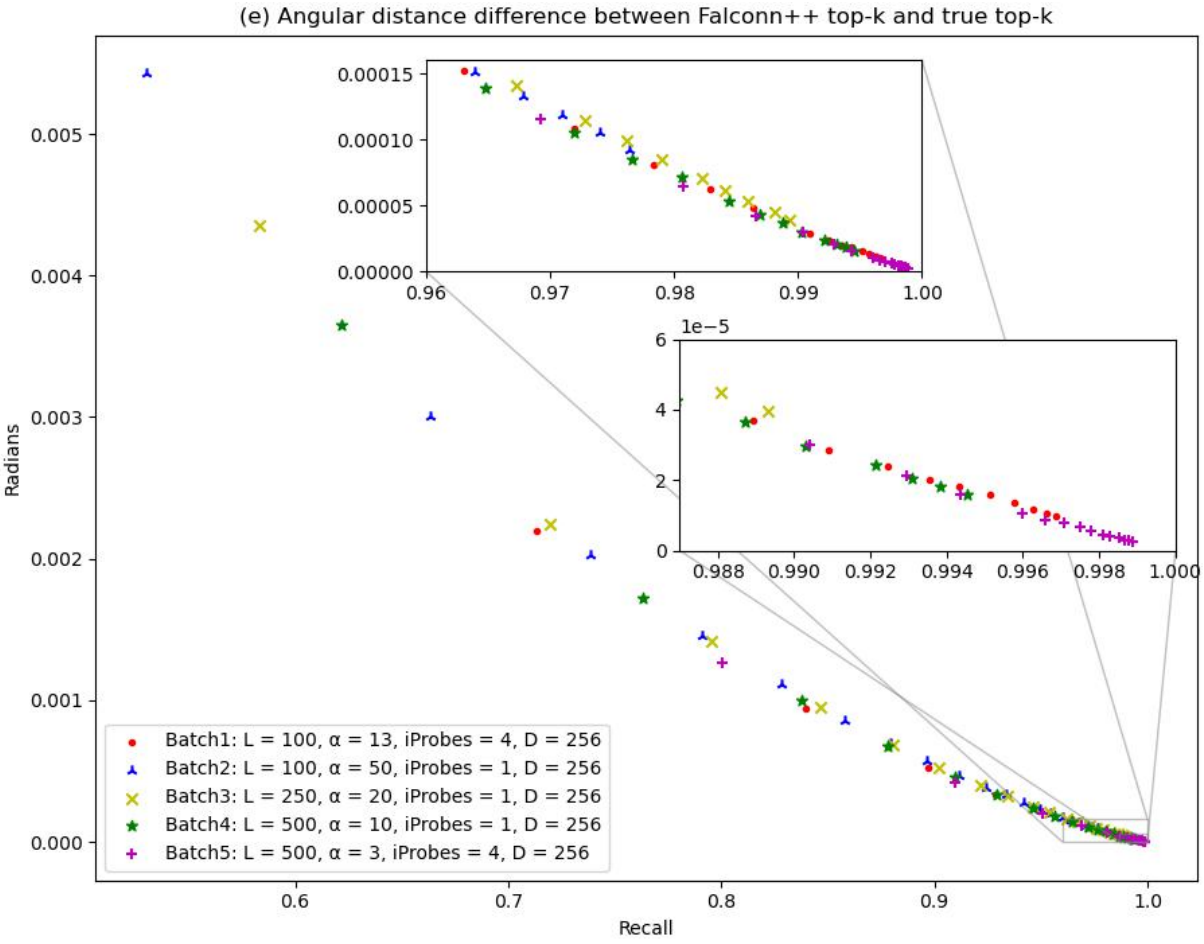


Figure 4.9: Distance difference for experiment 3: testing the impact of the limit on scaling.

Finally, Figure 4.8(d) shows that the greatest impact on indexing time is obtained by increasing the number of hash tables. Combining this observation with the ones done in Experiment 1 and Experiment 2 for the indexing time, we can conclude that the biggest impact is given by increasing the number of hash tables L and/or random vectors D, meanwhile, the impact of multi-probing for indexing, still significant, is contained compared to L and D.

Bibliography

[1] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC '98, Dallas, Texas, USA: Association for Computing Machinery, 1998, pp. 604–613, isbn: 0897919629. doi: 10.1145/276698.276876. [Online]. Available: https://doi.org/10.1145/276698.276876.

[2] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28, Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/2823f4797102ce1a1aec05359cc16dd9-Paper.pdf.

[3] N. Pham and T. Liu, *Falconn++: A locality-sensitive filtering approach for approximate nearest neighbor search*, 2022. arXiv: 2206.01382 [cs.DS].

[4] P. I. Alexandr Andoni, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, 2006.

[5] R. O'Donnell, Y. Wu, and Y. Zhou, *Optimal lower bounds for locality sensitive hashing (except when q is tiny)*, 2009. arXiv: 0912.0250 [cs.DS].

[6] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, *Practical and optimal lsh for angular distance*, 2015. arXiv: 1509.02897 [cs.DS].

[7] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07, Vienna, Austria: VLDB Endowment, 2007, pp. 950–961, isbn: 9781595936493.

[8] N. Pham, "Simple yet efficient algorithms for maximum inner product search via extreme order statistics," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, ser. KDD '21, Virtual Event, Singapore: Association for Com-

puting Machinery, 2021, pp. 1339–1347, isbn: 9781450383325. doi: 10.1145/3447548.3467345. [Online]. Available: https://doi.org/10.1145/3447548.3467345.

[9] H. A. David and J. Galambos, "The asymptotic theory of concomitants of order statistics," *Journal of Applied Probability*, vol. 11, no. 4, pp. 762–770, 1974. doi: 10.2307/3212559.

[10] N. Pham, *Presentation on simple yet efficient algorithms for maximum inner product search via extreme order statistics*, Presentation slides, 2021. [Online]. Available: https://github.com/NinhPham/CEOs/blob/master/paper/Slide-KDD21.pdf.

[11] R. Panigrahy, *Entropy based nearest neighbor search in high dimensions*, 2005. arXiv: cs/0510019 [cs.DS].