ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 112

Systems Group, Department of Computer Science, ETH Zurich

Indexing Scalar Product Queries

by

Bojana Dimcheva

Supervised by

Dr. Arijit Khan, Prof. Dr. Donald Kossmann

November 2013–May 2014

inf | Informatik
Computer Science

Acknowledgements

First and foremost, I would like to express my great appreciation to my supervisor Dr. Arijit Khan who provided me with enormous help and guidance thorough out the whole course of this master project. I would also like to thank Professor Donald Kossmann for his encouragement and support in the crucial moments of the development of this thesis.

Contents

Abstract

We present an indexing scheme that is capable of answering a broad range of complex top-k queries that contain a scalar product between two vectors when one of the vectors is present in the database, while the other one arrives at query time. Specifically we provide indexing facilities for the top-k closest points below (above) a hyperplane and the top-k farthest points below (above) a hyperplane, when the hyperplane is described by the scalar product equation $\langle \mathbf{a}, \mathbf{x} \rangle = b$. The parameters $\mathbf{a}, b$ are parameters of the query, whereas the set of points that we query is materializable from the database.

We provide in depth explanation of the algorithms and data structures that compose our indexing scheme. Furthermore, we evaluate and compare the performance of our index with an exhaustive search baseline algorithm, with different types of queries, on synthetic datasets. In addition to the baseline comparison, we compare our indexing scheme with an alternative solution method, that uses completely different approach.

The space consumption of the index is parametrizable, and depending on the performance needs and available memory can range from extremely lightweight to more demanding. The query time complexity is logarithmic on the number of data points in the best case and linear on the number of data points in the worst case.

Chapter 1

Introduction

The scalar product is mathematically defined as a multiplication of two vectors of same size that produces a single number as result. Geometrically the scalar product is defined as the cosine of the angle between the two vectors multiplied by their magnitudes $\langle \mathbf{a}, \mathbf{b} \rangle = ||a||\ ||b||\ \cos\theta$, algebraically it is defined as the sum of the products of the matching coordinates $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_i a_i b_i$. The scalar product has enormous use in mathematics, physics, mechanics etc., and it is one of the most used operations in scientific databases, modern large scale databases for video games, data mining databases, physical simulation databases and many other complex databases. In many of these applications there is an increasing need for querying the storage system with complex query types that involve a scalar product. Unlike other often used operations there is no general indexing scheme available for queries that contain a scalar product. One reason for this might be the fact that it is difficult to derive an indexing scheme that satisfies both the precision requirements of low dimensional data and the efficiency requirements of high dimensional data.

We try to derive a precise indexing scheme based on the Parallel Planar Index from [6], that is capable of scaling easily. In this thesis we are interested in top-k queries of the type: "Given a database $\mathcal{D} = [p_1, p_2, \ldots, p_N]$ of N rows in some application specific domain $\mathbb{D}$ and an application specific known function $\phi : \mathbb{D} \to \mathbb{R}^d$ we are interested in finding the top-k nearest (farthest) points to a scalar product query."

Problem: **[Top-K nearest (farthest) points query]** *Given some k, find the top-k data points $p \in \mathcal{D}$ satisfying $\langle \mathbf{a}, \phi(p) \rangle \leq b$, which also minimize (maximize) $\frac{|\langle \mathbf{a}, \phi(p) \rangle - b|}{|\mathbf{a}|}$.*

We assume that the function ϕ and the points $p \in \mathcal{D}$ are known in indexing time; i.e., the values $\phi(p)$ are either present in the database or they are materializable from it. Both the query parameters $\mathbf{a} \in \mathbb{R}^d$ and inequality parameter $b \in \mathbb{R}$ are known only at query time.

The definition of the problem uses $\leq$ - less than equal constraint but the indexing scheme can support the following operators:

1. $<$ - less than;
2. $\leq$ - less than or equal to;
3. $>$ - greater;
4. $\geq$ - greater than or equal to.

Furthermore, the indexing scheme does not put many constraints on the data units $p \in \mathcal{D}$, they can be data rows, objects, etc. as long as the function ϕ can transform the data points to the d-dimensional real space domain.

It is important to mention that in this thesis, for simplicity, we will only consider data units p for which the transformed points $\phi(p)$ are in the positive part of the space and queries with positive coordinates for all dimensions. The algorithms that are going to be explained rely on this restriction. We will assume that the function ϕ does the task of scaling the coordinates of the points. The method can be generalized though and [10] offers an explanation how to treat points with negative coordinates.

The thesis is divided in 6 chapters. In the next chapter we present the theory behind the Planar index as well as the algorithms and data structures that make up the indexing scheme. We will present all of the Planar index logic using the top-k nearest neighbour search query and explain how it can be generalized to the top-k farthest points query. In the third chapter we explain the H-Hash method, an alternative method that solves a similar problem to the top-k closest points problem and can be extended to solve this problem. In the fourth chapter we explain all the experiment settings that we implemented and show the results of these experiments. For clarity, we do not show the full results for all of these experiments and show extracts, while the full results can be found in the appendix. In the fifth chapter we conclude our study and discuss future research directions and possibilities. Finally the last chapter is the appendix that contains the full results of some of the experiment settings that were too cumbersome to show in the results section.

Some parts of the research motivating this thesis were published as [10], where also extensive study on the Planar index including some real data evaluations can be found.

1.1 Motivation

The top-k query problems described earlier find use in various data mining and machine learning applications as active learning where we need to retrieve the closest or farthest points from a classifier hyperplane. Efficiently and precisely retrieving these points can significantly improve the whole performance of the classification algorithm.

Chapter 2

Planar Index

In this section we will present algorithmic extensions and improvements of the Parallel Planar Index from [6], that improve the performance of the index and enable indexing of top-k nearest neighbour query types.

The chapter is organised as follows. First we present the Parallel Planar Index as presented in [6]. Then we present the possible additions of more components to the index. Finally, we present the extensions crucial for answering top-k nearest neighbour queries.

2.1 Background: Parallel Planar Index

The basic Planar Index was described in a previous master thesis [6] and treats a fairly simpler problem than the top-k nearest neighbour problem that we are interested in. In the current section we will present the Parallel Planar Index as proposed in [6] and show the query answering mechanism for this problem.

2.1.1 Original Planar Problem

Given a database $\mathcal{D} = [p_1, p_2, \ldots, p_N]$ of N rows in some application specific domain $\mathbb{D}$ and an application specific known function $\phi : \mathbb{D} \to \mathbb{R}^d$, the Parallel Planar Index can index functions of the form $\langle \mathbf{a}, \phi(p) \rangle \leq b$ in such a way that it can determine which points satisfy the inequality without giving any notion of the ordering of the points. So, the problem definition can be formulated as:

Problem: *Find all data points $p \in \mathcal{D}$, which satisfy a scalar product inequality: $\langle \mathbf{a}, \phi(p) \rangle \leq b$.*

Similarly as in the top-k nearest neighbour problem, the query parameters $\mathbf{a}$ and b are known only at query time. In a d-dimensional Euclidian co-ordinate system with axes $(x_1, x_2, \ldots, x_d)$ the query can be interpreted as:

Find all data points below or on the query hyperplane:

$$H(q) : \langle \mathbf{a}, x \rangle = \sum_{i=1}^{d} a_i x_i = a_1 x_1 + a_2 x_2 + \ldots + a_d x_d = b$$

2.1.2 Index Creation

[6] proposes having a family of parallel hyperplanes in the d-dimensional space that pass through each point in the dataset. Since these hyperplanes are parallel they all have the same normal vector $\mathbf{n} = (n_1, n_2, \ldots, n_d) \in R^d$. Each hyperplane is fully described with the normal vector and the offset from the origin. We can obtain the offset for each hyperplane by just substituting the values of its point coordinates for each dimension. Following the mathematical representation that we have been using so far the offset can be represented as the scalar product between the normal vector and the point to which the hyperplane belongs $\langle \mathbf{n}, \phi(p) \rangle$. This way we make sure that the hyperplane passes through the point and the normal vector is preserved. The offset value is called the *score* of the point:

$$score\,(p) = \langle \mathbf{n}, \phi(p) \rangle = \sum_{i=1}^{d} n_i \phi_i(p)$$

Having defined the $score\,(p)$ for each point $p \in \mathcal{D}$, we can write the equation that describes the parallel planar index hyperplane for that particular point.

$$H(p) : n_1 x_1 + n_2 x_2 + \ldots + n_d x_d = score\,(p)$$

Figure 2.1 shows a 2-dimensional example of a dataset of 7 points and the hyperplanes passing through them. The *score* is also important because it serves as criteria for sorting all the points in our database $\mathcal{D}$. The points are sorted in ascending order and stored in a random access data structure $\mathcal{A}$. This completes the indexing method of the parallel planar index. The pseudo code for the indexing algorithm follows.

Algorithm 1 Index Construction Algorithm

Require: normal vector $\mathbf{n}$
1: **for all** $p \in \mathcal{D}$ **do**
2: $\quad score\,(p) \leftarrow \langle \mathbf{n}, \phi(p) \rangle$.
3: **end for**
4: $\mathcal{A} \leftarrow$ sort $\mathcal{D}$ according to *score*.

To sum up the index is composed of three data structures:

1. Normal vector $\mathbf{n}$ with a space requirement $\mathcal{O}\,(d)$;
2. Score vector *score* with a space requirement $\mathcal{O}\,(N)$;
3. Sorted array of points $\mathcal{A}$ with a space requirement $\mathcal{O}\,(N)$.

The total space complexity of the index is $\mathcal{O}\,(2N + d)$.

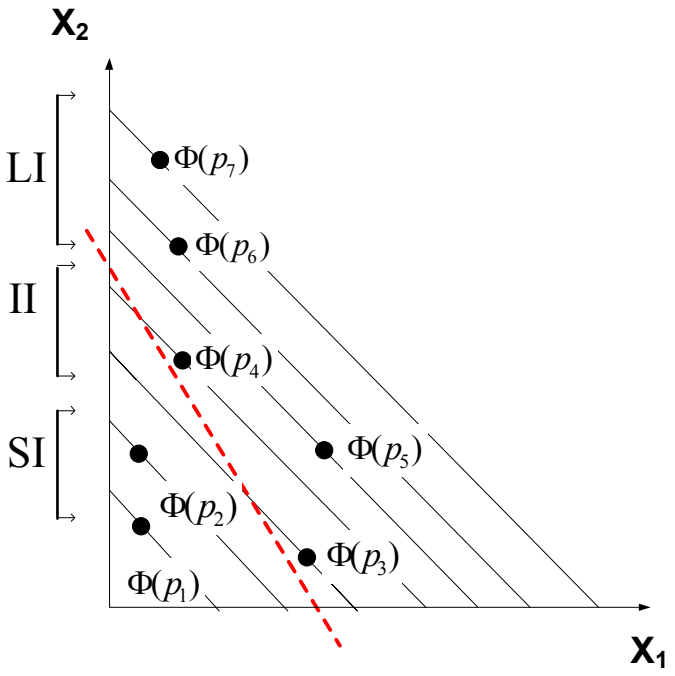


Figure 2.1: Smaller interval SI, larger interval LI and intermediate interval II geometrical interpretation

2.1.3 Query Answering

As mentioned in section 2.1.1 the basic Parallel Planar Index can answer queries of the type "find the set $\mathcal{P}$ where $\mathcal{P} := \{p : \langle \mathbf{a}, p\rangle \leq b\}$". In this section we will use the mentioned example query to show how the query answering mechanism works. Having already defined the index hyperplanes $H(p) : \langle \mathbf{n}, x\rangle = \langle \mathbf{n}, \phi(p)\rangle$ and the query hyperplane $H(\mathbf{q}) : \langle \mathbf{a}, x\rangle = b$, we can proceed defining the relationship between them that allows for pruning of some points.

Let us denote by $I(\mathbf{q}, i)$ the i-th co-ordinate of the intersection point between the query hyperplane $H(\mathbf{q})$ and the axis x_i. Similarly, assume $I(p, i)$ denotes the i-th co-ordinate of the intersection point between the index hyperplane $H(p)$ and the axis x_i. We have, $I(\mathbf{q}, i) = \frac{b}{a_i}$ and $I(p, i) = \frac{\langle \mathbf{n}, \phi(p)\rangle}{n_i}$. Next, we define a partition of the data points into three non-overlapping intervals for efficiently processing our inequality queries.

Definition 1 (Smaller interval)**.** *The smaller interval, denoted by SI, consists of all data points p for which the index hyperplane $H(p)$ intersects the axes at points closer to the origin as compared to the intersection points between the query hyperplane $H(\mathbf{q})$ and the corresponding axes. Formally,*

$$SI = \{p : p \in \mathcal{D} \wedge (\forall i)\big(I(p,i) < I(\mathbf{q},i)\big)\}$$

Definition 2 (Larger interval)**.** *The larger interval, denoted by LI, consists of all data points p for which the index hyperplane $H(p)$ intersects the axes at points farther to the origin as compared to the intersection points between the query hyperplane $H(\mathbf{q})$ and the corresponding axes. Formally,*

$$LI = \{p : p \in \mathcal{D} \wedge (\forall i)\big(I(p,i) > I(\mathbf{q},i)\big)\}$$

Definition 3 (Intermediate interval)**.** *The intermediate interval, denoted by II, consists of all data points p which belong to neither the smaller interval nor the larger interval.*

$$II = \{p : p \in \mathcal{D} \wedge (\exists i, i')\big(I(p,i) \leq I(\mathbf{q},i) \wedge I(p,i') \geq I(\mathbf{q},i')\big)\}$$

According to Definition 1 for all the points $p \in SI$ we know that for all dimensions i in the d-dimensional space:

$$\frac{score(p)}{n_i} < \frac{b}{a_i}$$

Which implies that the index hyperplane $H(p)$ will never intersect the query hyperplane $H(\mathbf{q})$ in a point with non-negative coordinates. The similar reasoning can be applied for the points from the large interval LI. Consequently, two interesting observations arise from the definitions of our intervals, which are given below. A formal proof of these observations can be found in [6].

Observation 1. *All data points in the larger interval do not satisfy the inequality query, and therefore, can be rejected.*

Observation 2. *All data points in the smaller interval satisfy the inequality query, and hence, can be accepted.*

Figure 2.1 shows a 2-dimensionally set example with the respective intervals $SI = \{p_1, p_2\}$, $II = \{p_3, p_4\}$ and $LI = \{p_5, p_6, p_7\}$.

Observations 1 and 2 create the basis for accepting and rejecting data points without actually computing the scalar products for them. We only need to evaluate the query for the data points which are in the intermediate interval. Particularly, given an inequality query, we first identify the intersection coordinates $I(\mathbf{q}, i)$ between the query hyperplane $H(\mathbf{q})$ and the corresponding axes. Recall that we have already sorted the data points p in a random access array $\mathcal{A}$ in ascending order of their $\langle \mathbf{n}, \phi(x) \rangle$ values. Now, for each axis x_i, we perform a binary search on $\mathcal{A}$ and find two locations in $\mathcal{A}$ — denoted as $Small(i)$ and $Large(i)$, respectively, and formally defined with the following equations.

$$Small(i) = \max\{j : \mathcal{A}(j) = p \wedge I(p,i) < I(q,i)\}$$
$$Large(i) = \min\{j : \mathcal{A}(j) = p \wedge I(p,i) > I(q,i)\}$$

Algorithm 2 Online Algorithm for the Inequality Query

Require: sorted list $\mathcal{A}$ of p in asc. order of $\langle \mathbf{n}, \phi(p) \rangle$,
query $\langle \mathbf{a}, \mathbf{x} \rangle \leq b$.

```
find intermediate (II) and smaller (SI) intervals. [Binary Search on A]
for all j ∈ SI do
    p ← A(j)
    output p
end for
for all j ∈ II do
    p ← A(j)
    if ⟨a, φ(p)⟩ ≤ b then
        output p
    end if
end for
```

The above-mentioned binary search operations require $\mathcal{O}(d \log N)$ time. Using $Small(i)$ and $Large(i)$ values for all i, the boundaries of SI, LI, and II are computed as follows.

$$j_{min} = \min_{i \in [1,d]} \{Small(i)\}$$

$$j_{max} = \max_{i \in [1,d]} \{Large(i)\}$$

$$SI \leftarrow \mathcal{A}[1 : j_{min}]$$

$$II \leftarrow \mathcal{A}[j_{min} + 1 : j_{max} - 1]$$

$$LI \leftarrow \mathcal{A}[j_{max} : N]$$

Computing the interval boundaries requires $\mathcal{O}(d)$ time. Finally, two sets of data points are returned in the answer set: **(1)** for all data points in the intermediate interval, the scalar product is evaluated, and then those data points which satisfy the given scalar product inequality are reported, as well as **(2)** all data points in the smaller interval are reported. Therefore, the time complexity of the query-processing algorithm is $\mathcal{O}(d(\log N + |\text{II}|))$, where |II| is the cardinality of the intermediate interval.

2.2 Index Normal Vectors

By now it is fairly obvious that the quality of the Planar Index depends solely on the normal vector of the index hyperplanes. If the normal vector of the index hyperplanes is parallel to the normal vector of the query we can prune all of the points. On the other hand, if the normal vector of the index hyperplanes is nearly perpendicular to the normal vector of the query hyperplane we can end up with an intermediate interval II that contains every point in the dataset and empty smaller interval SI and larger interval LI. In this case none of the points will be pruned. In order to speed up the query answering process the cardinality

of the intermediate interval has to be as small as possible. To achieve this a careful selection of the normal vector is crucial.

The first addition that we will propose to the Planar Index is having multiple normal vectors and defining a set of index hyperplanes for each normal vector. Next we will present few ways of generating the normal vectors and discuss some duplicate and near-duplicate removal strategies.

2.2.1 Multiple Sets of Index Hyperplanes

Since the Planar Index is lightweight data structure and does not consume a lot of memory we can create and store multiple such structures and try to reduce the size of the intermediate interval for each query. For some arbitrary number of normal vectors I we can index each point with each of the indexes separately. Furthermore, we can parametrize the number I and increase it or reduce it depending on speed requirements and available memory. As we increase the number of indexes we also increase the probability that there would exist an appropriate index for any given query. Deciding whether some index is appropriate for a given query is not such a trivial task as it looks. Consequently, choosing which index is the best for a given query is a task on its own and we will devote the next section to explaining the index selection method.

Index Selection at Query Time

One brute force approach would be to count the number of points in the intermediate interval for each index for the given query. However, this entails completing the binary search phase of the query answering mechanism for each index, which might be too expensive to compute in query time even if we have a reasonable number of indexes and dimensions. We propose an efficient greedy method for finding the best index, that uses heuristics to minimize the volume of the intermediate interval.

Assuming that the data points are distributed uniformly, the best planar index is the one which minimizes the "volume" of the intermediate interval for a given query. Below we clarify the notion of volume spanned by the intermediate interval in $\mathbb{R}^d$.

Let us denote by $\mathbf{q_i}$ the intersection point between the query hyperplane $H(\mathbf{q})$ and the i-th co-ordinate axis x_i. We recall that the i-th co-ordinate of the intersection point $\mathbf{q_i}$ is denoted as $I(q, i)$. Next, for a planar index with normal vector $\mathbf{n} = (n_1, n_2, \ldots, n_d)$, we consider the set of hyperplanes $H(\mathbf{q_i})$ passing through these intersection points $\mathbf{q_i}$, and parallel to index hyperplanes with normal vector $\mathbf{n}$. Follows the equation of the hyperplane $H(\mathbf{q_i})$.

$$H(\mathbf{q_i}) : n_1x_1 + n_2x_2 + \ldots + n_dx_d = n_iI(q, i)$$

There will be d such hyperplanes for total d intersection points. We find two hyperplanes H_{max} and H_{min} among them which have maximum and minimum

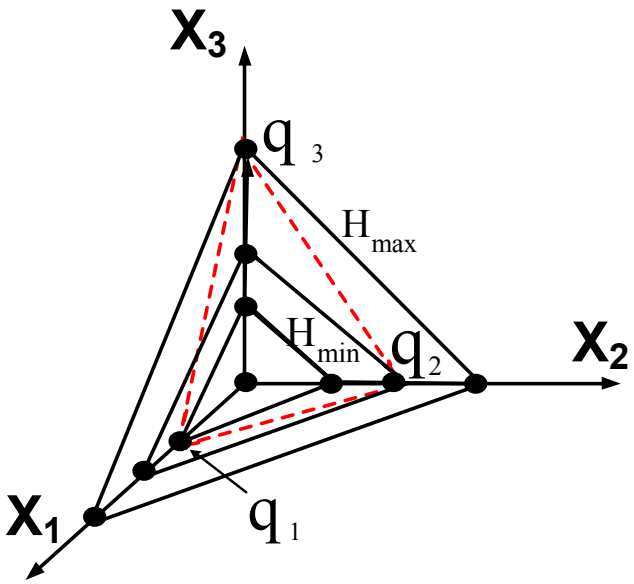


Figure 2.2: Volume of intermediate interval in 3D, the volume is bounded by the H_min and H_max hyperplanes

values of $n_i I(q,i)$, respectively.

$$H_{max} = H(\mathbf{q_{i_1}}) : i_1 = \arg\max_{i \in (1,d)} n_i I(q,i) \tag{2.1}$$

$$H_{min} = H(\mathbf{q_{i_2}}) : i_2 = \arg\min_{i \in (1,d)} n_i I(q,i) \tag{2.2}$$

We are now ready to formally define the volume of the intermediate interval.

Definition 4 (Volume of intermediate interval)**.** *Given a query hyperplane and a planar index, we define the volume of the intermediate interval as the volume of the hyper surface bounded by the two hyperplanes H_{max}, H_{min}, and the co-ordinate axes.*

As an example, Figure 2.2 shows the volume of intermediate interval in three dimension for a given query and a planar index. It is easy to verify that any point which lies on the hyper surface bounded by H_{max}, H_{min}, and the co-ordinate axes is in the intermediate interval. On the other hand, points which are outside the aforementioned hyper surface belong to either the smaller or the larger interval.

Claim 1. *Given a query hyperplane and a planar index, consider the hyper surface bounded by the two hyperplanes H_{max}, H_{min}, and the co-ordinate axes. If a data point lies on this hyper surface, then it is also in the intermediate interval. On the other hand, if some point lies outside this hyper surface, then it is either in the smaller or in the larger interval.*

Proof. First, we shall consider a point p on the hyper surface bounded by H_{max}, H_{min}, and the co-ordinate axes, and also assume that p is not on the boundary of H_{max}. The index hyperplane $H(p)$ passing through p intersects the axis x_i at a distance $\frac{\sum_{k=1}^{d} n_k \phi_k(p)}{n_i}$ from the origin. Since p is on the hyper surface bounded by H_{max}, H_{min}, and the co-ordinate axes, we have the following relation for all $i \in \{1, d\}$.

$$\frac{n_{i_2} I(q, i_2)}{n_i} \leq \frac{\sum_{k=1}^{d} n_k \phi_k(p)}{n_i} \leq \frac{n_{i_1} I(q, i_1)}{n_i} \tag{2.3}$$

We recall that $\frac{n_{i_2} I(q,i_2)}{n_i}$ and $\frac{n_{i_1} I(q,i_1)}{n_i}$ are the co-ordinates of the intersection points of axis x_i with H_{min} and H_{max}, respectively. We shall prove that p is in the intermediate interval by contradiction. Let, if possible, p does not belong to the intermediate interval. Hence, there can be two mutually disjoint events: **(1)** p belongs to the smaller interval, that is, $\frac{\sum_{k=1}^{d} n_k \phi_k(p)}{n_i} < I(q, i)$ for all $i \in \{1, d\}$. or, **(2)** p belongs to the larger interval, that is, $\frac{\sum_{k=1}^{d} n_k \phi_k(p)}{n_i} > I(q, i)$ for all $i \in \{1, d\}$. However, **(1)** contradicts inequality 2.3 when $i = i_2$. Similarly, **(2)** contradicts inequality 2.3 when $i = i_1$. Therefore, $\mathbf{x}$ belongs to the intermediate interval.

Next, we shall prove the second part of our claim, that is, when p is outside the hyper surface bounded by H_{max}, H_{min}, and the co-ordinate axes. Again, there can be two mutually disjoint cases: **(1)** $\frac{\sum_{k=1}^{d} n_k \phi_k(p)}{n_i} > \frac{n_{i_1} I(q,i_1)}{n_i}$ for all $i \in \{1, d\}$. However, following the definition of H_{max} from Equation 2.1, $\frac{n_{i_1} I(q,i_1)}{n_i} \geq I(q, i_1)$. Hence, $\frac{\sum_{k=1}^{d} n_k \phi_k(p)}{n_i} > I(q, i_1)$, which ensures that p is in the larger interval. **(2)** On the other hand, when $\frac{\sum_{k=1}^{d} n_k \phi_k(p)}{n_i} < \frac{n_{i_2} I(q,i_2)}{n_i}$ for all $i \in \{1, d\}$, one can analogously show that p belongs to the smaller interval. This completes the proof. □

Therefore, assuming uniform distribution of the data points, one will select the planar index which reduces the volume of the intermediate interval for a given query. Unfortunately, finding the volume of a hyper surface in higher dimension itself is a very difficult problem. Since the volume of a hyper surface is roughly proportional to the "stretch" of the hyper surface along each axis, we greedily decide the best planar index as the one which minimizes the maximum stretch of the intermediate interval along any axis. We formally define the stretch of the hyper surface in Definition 5.

Definition 5. *Given a scalar product query q, the stretch of the intermediate interval due to some planar index (with normal vector $\mathbf{n}$) along the axis x_i is computed as follows.*

$$Stretch(\mathbf{n}, i) = \frac{1}{n_i} \big[\max_{k \in \{1,d\}} n_k I(q, k) - \min_{k \in \{1,d\}} n_k I(q, k) \big] \tag{2.4}$$

The best planar index is selected as the one which minimizes the maximum stretch of the intermediate interval along any axis, i.e.,

$$\arg\min_{\mathbf{n}} \max_{i \in \{1,d\}} Stretch(\mathbf{n}, i) \tag{2.5}$$

The time complexity of the best index selection method is $\mathcal{O}(Id)$.

2.2.2 Generation of normal vectors

The index hyperplanes model the query hyperplanes, so the normal vectors of the index hyperplanes should be generated to match the query normal vectors. In an ideal scenario the queries would come from a known domain and we would generate the normal vectors of the index hyperplanes in this domain. The query domain cannot be known beforehand always though, thus in this section we will propose general techniques for generating the normal vectors as well as techniques for generating from a known domain.

We will start with the simplest case assuming that the coordinates of the normal vector of the query are integers that are drawn from a known discrete uniform distribution $\mathcal{U}[1, RQ]$, where RQ is a variable parameter. The coordinates for different dimensions are completely independent from each other and we assume that the distribution for each dimension has the same RQ parameter for simplicity. In this case the normal vector generation technique is trivial, since we know the distribution and can draw the coordinates for each of the normal vectors from it.

Duplicates removal technique

What might happen in the aforementioned scenario is that the collection of planar indexes might contain duplicates after the generation step. The obvious example is two normal vectors of the planar indexes containing equivalent coordinates, but example 1 shows another case where we have functional duplicates.

Example 1. *Consider an index collection with 2 normal vectors in $d = 2$:*

$$\mathbf{n_1} = (1, 2)$$

$$\mathbf{n_2} = (2, 4)$$

Example 1 shows a collection of 2 normal vectors in a 2-dimensional setting that are not equivalent but have the same direction. If normalized, these vectors would be equivalent. Since the index hyperplanes are described with the following equation

$$\langle \mathbf{n}, \mathbf{x} \rangle = \langle \mathbf{n}, \phi(p) \rangle ,$$

these two normal vectors will yield equivalent index hyperplanes for any given point. We care only about the direction of the normal vectors, not their length;

Algorithm 3 Index generation with duplicates removal technique for normal vectors with integer coordinates

```
1: while index_collection not full do
2:    for all i ∈ [1, d] do
3:       n_i ← U[1, RQ]
4:    end for
5:    max_value ← arg max_{i∈[1,d]} n_i
6:    already_present ← false;
7:    for all multiplier ∈ [1, RQ / max_value] do
8:       if not generated_indexes.insert(multiplier · n) then
9:          already_present ← true
10:         break;
11:      end if
12:   end for
13:   if not already_present then
14:      index_collection.insert(n);
15:   end if
16: end while
```

hence, we should eliminate the functional duplicates as well as the actual duplicates. We propose an index generation algorithm with duplicates removal shown in algorithm 3.

The duplicates removal algorithm uses an additional data structure *generated_indexes* to keep track of all the used indexes, however this structure is only needed in indexing time, as soon as the generation of normal vectors in the indexing step is done the memory can be freed. The actual normal vectors are stored in the *index_collection* data structure for further use at indexing and query time.

Unit vectors under a hypersphere

The aforementioned generation method works well when we know the exact domain of the query normal vectors and this domain contains finite set of discrete values. However, we want to further generalize the planar index structure to support broader range of queries. While the duplicates removal technique successfully eliminates redundant normal vectors in the case where the coordinates of the indexes are integers, expanding this technique to the floating point domain is expensive. The next generation technique that we will present can be used to answer queries that have coordinates from a continuous domain.

There is one useful observation that we already mentioned for the general case: We only care about the direction of the normal vectors. Taking this into account we can go further and explore the option of generating normal vectors that are normalized by length. In an ideal case our normal vectors would be unit vectors that are distributed uniformly on the surface of a hypersphere.

The problem of generating unit vectors that are uniformly distributed on the surface of a hypersphere is called "hyper sphere point picking" in mathematics and it is quite an old problem. In fact, there is a simple and efficient method,

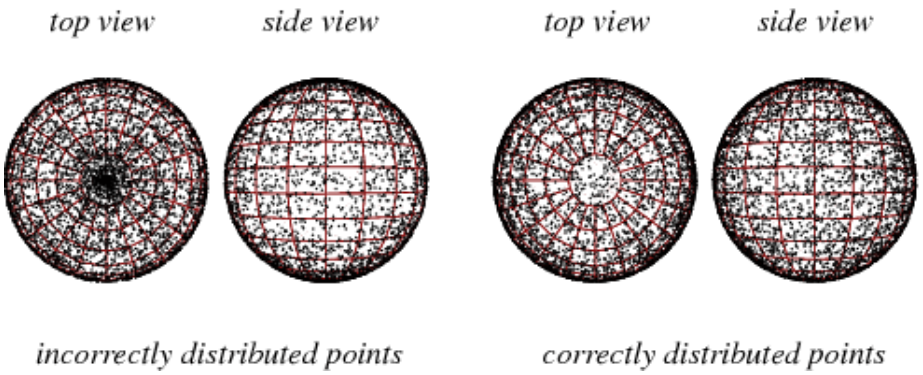


Figure 2.3: Uniform distribution on a sphere visualization (taken from [8])

proposed formally in [1]. If we generate d independent, normally distributed random variables $X_1, X_2, \ldots, X_d$, then the distribution of the vectors:

$$\frac{1}{\sqrt{{X_1}^2 + {X_2}^2 + \ldots + {X_d}^2}} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_d \end{bmatrix}$$

is uniform over the surface $\mathbb{S}^{d-1}$[4][1].

We use this method to generate coordinates for the index normal vectors in the experiment setting with continuous query coordinates.

2.2.3 Dynamic Updates

Even if the normal vectors are generated in such a way that the redundancy is minimal, they might not be the best match for the query distribution. This is especially a problem if we do not know the query distribution in advance. Here we propose a very simple and intuitive approach that enables the Planar index to adapt itself to the received queries. We propose partially rebuilding the index after we have seen some queries. We design these additions such that the additional processing in query-time is minimal.

We have to keep three additional data structures in the index and update them according to the incoming queries:

1. Minimum observed query coordinate for each dimension. The collection has a space requirement $\mathcal{O}(d)$;
2. Maximum observed query coordinate for each dimension. The collection has a space requirement $\mathcal{O}(d)$;
3. Count of visits per each normal vector in the index. The collection has a space requirement $\mathcal{O}(I)$

As we can see the extra space that we need for the dynamic updates is dependent on the dimension and the number of normal vectors, which normally take values much smaller than N the total number of data points, consequently these extensions are a small burden for the index. Furthermore the extra query time processing needed to update these data structures has time complexity of $\mathcal{O}(d)$.

The additional query statistics that we keep help us recreate some normal vectors by invoking the following recreation algorithm.

Algorithm 4 Dynamic index recreation algorithm

```
1: for all n ∈ Planar index do
2:   if visits_count[n] < α then
3:     for all i ∈ [1, d] do
4:       n_i ← U[min_observed_i, max_observed_i]
5:     end for
6:     Invoke duplicates removal method.
7:   end if
8: end for
```

The α parameter is adjustable parameter that can be set according to the number of normal vectors and the expected number of queries answered between two partial recreations.

In our experiments, for simplicity, we invoke this algorithm after we have observed a fixed number of queries, but in a real system the algorithm can be invoked in periods of low traffic, in such way so that the query answering will never suffer because of the partial index recreation.

2.3 Answering Top-K Query Types

In this section we will show how the Planar Index data structure can be extended to answer top-K query types. All of the top-K capabilities are built on top of the already existing capabilities of the Parallel Planar Index. Our Planar index supports both query types, which means that all of the mentioned novelties such as the multiple normal vectors, the different generation techniques and dynamic updates are also applicable for the inequality query types more extensively studied in [6].

As it was already mentioned the main problem of this thesis is retrieving a collection of top-K points that satisfy the inequality query $q : \langle \mathbf{a}, \phi(p) \rangle \leq b$, in which the ordering is determined by the distance from the query:

$$d(q, p) = \frac{\langle \mathbf{a}, \phi(p) \rangle - b}{||\mathbf{a}||}$$

Since we can represent our data as points in a d-dimensional space this leaves space for a geometrical interpretation of the query function. In a d-dimensional Euclidean co-ordinate system with axes $(x_1, x_2, \ldots, x_d)$ the problem can be interpreted as:

Find the top-k closest (farthest) points below or on the hyperplane:

$$\langle \mathbf{a}, x \rangle = \sum_{i=1}^{d} a_i x_i = a_1 x_1 + a_2 x_2 + \ldots + a_d x_d = b$$

We say that the points that satisfy the inequality:

1. $\langle \mathbf{a}, x \rangle < b$ are bellow the hyperplane;
2. $\langle \mathbf{a}, x \rangle = b$ lie on the hyperplane;
3. $\langle \mathbf{a}, x \rangle > b$ are above the hyperplane.

We will proceed explaining the top-K algorithm assuming that the Planar index is already built using some of the mentioned strategies and multiple normal vectors are stored together with all of the needed facilities for each of them. Since we have more than one normal vector the first step in the query answering procedure would be to determine the most suitable normal vector for the query using the algorithm described in section 2.2.1. Lets assume that the normal vector selection algorithm has chosen the normal vector $\mathbf{n}$. The next step would be to use $\mathbf{n}$ and its sorting of the data points to run a binary search that will determine the smaller interval SI, larger interval LI and intermediate interval II.

We would proceed with processing the intermediate interval in such a way that all of the points that satisfy the inequality query are stored into a collection, in our implementation this collection is a priority queue. The priority queue is limited to containing k items, i.e. every insert of the $k+1$-th item is followed by a delete. The priority queue is designed in such a way that the point that is furthest from the query from all the top-K points is always in the head element for easiest inspection and deletion. After the intermediate interval is processed, we continue processing the smaller interval (or the larger interval if we have a different relational operator) following the order of the chosen normal vector. There are two criteria that have to be satisfied before we can be sure that the top-K closest points are really in our priority queue and can be returned. In order to explain these two criteria we have to define the "lower-bound distance" of the index hyperplane $H(p)$ of a data point p to the query hyperplane $H(\mathbf{q})$, which is the lowest possible distance from any point that lies on $H(p)$ to the query hyperplane, in the first hyper octant.

Definition 6 (Lower-bound distance)**.** *Consider a query $q : \langle \mathbf{a}, \phi(p) \rangle \leq b$ and a Planar index with normal vector given by $\mathbf{n} = (n_1, n_2, \ldots n_d)$. For some data point p in the smaller interval, we define the "lower-bound distance" of the index hyperplane $H(p)$ to the query hyperplane $H(\mathbf{q})$ as the smallest value of $\frac{|\frac{a_i}{n_i}\langle \mathbf{n}, \phi(p) \rangle - b|}{||\mathbf{a}||}$ for all $i \in (1, d)$. Formally,*

$$LBS\big(H(p), H(\mathbf{q})\big) = \min_{i \in (1,d)} \frac{\left|\frac{a_i}{n_i}\langle \mathbf{n}, \phi(p) \rangle - b\right|}{||\mathbf{a}||}$$

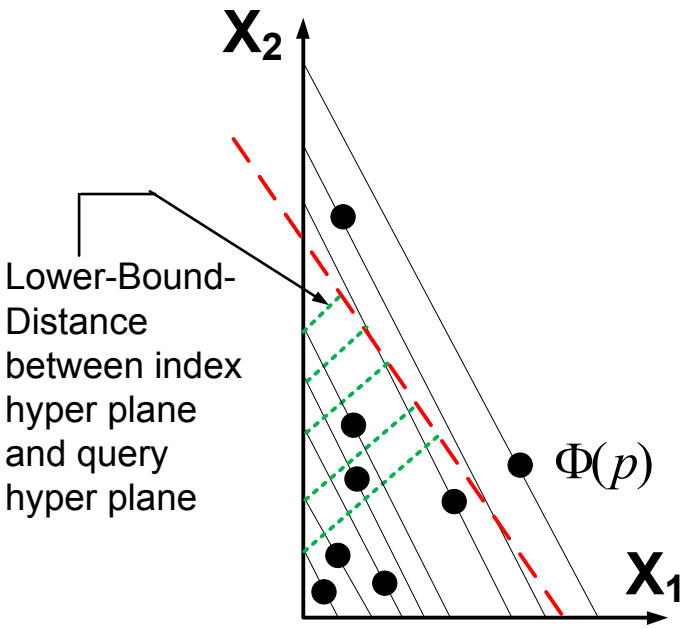


Figure 2.4: Lower-bound distance 2D

Using the lower-bound distance we can finish the algorithm and return the top-K data points when the following two conditions are satisfied:

1. The priority queue containing the top-k closest points is full.
2. $LBS\big(H(p), H(\mathbf{q})\big)$ is greater than the largest distance present in the priority queue.

We explain the reasoning behind this approach with the following claim.

Claim 2. *Consider two data points p_1, p_2 belonging to the smaller interval SI with respect to* $\mathbf{n}$. *If $\langle \mathbf{n}, \phi(p_1)\rangle > \langle \mathbf{n}, \phi(p_2)\rangle$, then it holds that $LBS\big(H(p1), H(\mathbf{q})\big)$ is smaller than $\frac{|\langle \mathbf{a}, \phi(p_2)\rangle - b|}{||\mathbf{a}||}$.*

Proof. We have to prove that the minimum value of $\frac{\left|\frac{a_i}{n_i}\langle \mathbf{n}, \phi(p_1)\rangle - b\right|}{||\mathbf{a}||}$ is smaller than $\frac{|\langle \mathbf{a}, \phi(\mathbf{p_2})\rangle - b|}{||\mathbf{a}||}$. Since p_1 and p_2 are in the smaller interval, and we are considering queries and $\phi(p)$'s in the first hyper octant, both $\big(\frac{a_i}{n_i}\langle \mathbf{n}, \phi(\mathbf{p_1})\rangle - b\big)$ and $\big(\langle \mathbf{a}, \phi(p_2)\rangle - b\big)$ are negative. Thus, we have to show that the minimum value of $\frac{a_i}{n_i}\langle \mathbf{n}, \phi(p_1)\rangle$ is larger than $\langle \mathbf{a}, \phi(p_2)\rangle$. We shall prove this fact by contradiction. Let, if possible, $\frac{a_i}{n_i}\langle \mathbf{n}, \phi(p_1)\rangle$ is smaller than or equal to $\langle \mathbf{a}, \phi(p_2)\rangle$, for all i. In other words, $a_i\langle \mathbf{n}, \phi(p_1)\rangle$ is smaller than or equal to $n_i\langle \mathbf{a}, \phi(p_2)\rangle$, for all i.

Algorithm 5 Online Algorithm for Top-k Nearest Neighbour Query

Require: sorted list $\mathcal{A}$ of p in asc. order of $\langle \mathbf{n}, \phi(p) \rangle$,
query $\langle \mathbf{a}, \mathbf{x} \rangle \leq b$ and k parameter
/* Process Intermediate Interval */
find intermediate (II) and smaller (SI) intervals. [Binary Search on $\mathcal{A}$]
for all $j \in II$ **do**
  $p \leftarrow \mathcal{A}(j)$
  if $\langle \mathbf{a}, \phi(p) \rangle \leq b$ **then**
    $topk\_pq \leftarrow p$
  end if
end for
/* Process Smaller Interval */
for all $j \in SI$(descending order) **do**
  $p \leftarrow \mathcal{A}(j)$
  if $topk\_pq$ is full **and** $LBS(H(p), H(\mathbf{q})) > topk\_pq.head$ **then**
    terminate
  else
    $topk\_pq \leftarrow p$
  end if
end for

Summing over i in both sides, we get

$$\sum_{i=1}^{d} a_i \langle \mathbf{n}, \phi(p_1) \rangle \leq \sum_{i=1}^{d} n_i \langle \mathbf{a}, \phi(p_2) \rangle = \sum_{i=1}^{d} a_i \langle \mathbf{n}, \phi(p_2) \rangle$$

Since we are in the first hyper octant and all a_i's are positive, the above result is a contradiction to our earlier assumption that $\langle \mathbf{n}, \phi(p_1) \rangle > \langle \mathbf{n}, \phi(p_2) \rangle$. This completes the proof. □

Claim 2 gives us basis for terminating the algorithm and not inspecting all of the points in the smaller interval. If the next point p_1 to process from the smaller interval has a lower-bound distance which is larger than the largest distance in the priority queue, we can be sure that the distance from p_1 to the query is larger than any of the distances in the priority queue. This holds for all other data points p_2 that have smaller score than p_1, i.e., $\langle \mathbf{n}, \phi(p_1) \rangle > \langle \mathbf{n}, \phi(p_2) \rangle$.

It is already visible how the algorithm can be modified to answer the top-k farthest points. We would iterate the sorted points in the opposite order and the pruning logic would rely on defining a "higher-bound distance" similarly as the lower-bound distance. The second condition for terminating in this case would be to stop when the higher bound distance is smaller than the smallest distance in the priority queue.

To estimate the time complexity of the algorithm we first have to take into account that we need $\mathcal{O}(dI)$ time to determine the best index. Then we need $\mathcal{O}(d \log N)$ to find the boundaries of the intermediate interval. We need additional $\mathcal{O}(d|II|)$ time to examine all the points in the intermediate interval, for a cardinality of the intermediate interval $|II|$. If we inspect additional k_1

elements from the smaller interval, the overall time complexity of the top-k algorithm would be $\mathcal{O}\left(dI + d\log N + (|II| + k_1)(d + \log k)\right)$. In the best case for a Planar index that is parallel to the query and there are no points lying on the query $|II| = 0$ and $k_1 = k+1$ the complexity is $\mathcal{O}\left(dI + d\log N + kd + k\log k)\right)$.

Chapter 3

Alternative Methods

Finding appropriate method to compare the Planar index with is a difficult task owing to the fact that there are not many other methods that provide indexing capabilities for the same query type as we define the top-K nearest neighbour query in Chapter 2. One powerful probabilistic method which solves a similar problem is the H-Hash method proposed in [3]. The H-Hash method provides indexing structure for the absolute nearest neighbour to a query hyperplane problem. This problem is an important problem in data mining and active learning, so the H-Hash method is best suited for normalized data, since it assumes that the data is normalized. Also, the H-Hash method assumes query hyperplanes that pass through the origin, which is not the case with the Planar index.

In this thesis, we are mostly interested in non-normalized data and queries that have non-zero offset from the origin. We transform our data to comply with the requirements of the H-Hash method similarly as in [2], which means that we will loose some information and require additional processing. This is why this method is not a perfect match in the quest of a competing method for the Planar index, but to the best of our knowledge, we didn't manage to discover more appropriate ones. We will first present the theory behind locality sensitive hashing, then we will proceed explaining the method as presented in [3]. Finally, in this chapter we will quickly summarize a useful amplification that allows for parametrizing locality sensitive hashing, which we used in our implementation.

3.1 Locality Sensitive Hashing

Locality sensitive hashing [7] is a method of randomized hashing that guarantees that the probability of collision of two data items is inversely proportional to their distance.

A family of locality sensitive hashing functions has to be defined on a set $\mathcal{S}$ with a distance function $d(p, q)$ over any two items $p, q \in \mathcal{S}$. Formally:

Definition 7. *Let $h_{\mathcal{H}}$ be a randomly chosen hash function from the family $\mathcal{H}$. The family $\mathcal{H}$ is called (d_1, d_2, p_1, p_2)-sensitive for $d(\cdot, \cdot)$, when, for any $p, q \in \mathcal{S}$,*

- *if $d(p, q) \leq d_1$ then $Pr[h_{\mathcal{H}}(p) = h_{\mathcal{H}}(q)] \geq p_1$,*
- *if $d(p, q) \geq d_2$ then $Pr[h_{\mathcal{H}}(p) = h_{\mathcal{H}}(q)] \leq p_2$.*

A family of functions $\mathcal{H}$ is only useful if $p_1 > p_2$. Most of the locality sensitive functions proposed solve the nearest neighbour search problem i.e., they provide performance improvement for the problem of finding closest point to a given query point. This is obviously different to our problem of finding top-K closest points below(above) a query hyperplane. The locality sensitive method that we are going to present in the next section solves the problem of finding the closest point to a given query hyperplane, which can be adapted to provide a solution for our problem.

3.2 H-Hash - Hyperplane Hashing based on Angle Distance

The H-Hash(Hyperplane Hashing based on Angle Distance) method was proposed in [3] to solve the problem of retrieving from a dataset $\mathcal{D} = [p_1, p_2, \ldots, p_N]$ the closest point(s) to a given query hyperplane whose normal vector is given by $\mathbf{w} \in \mathbb{R}^d$. It is assumed that the query hyperplane passes through the origin and both the points and the query normal are unit vectors. The Euclidean distance of a point p to a given hyperplane $h_{\mathbf{w}}$ is given by:

$$d(h_{\mathbf{w}}, p) = | \langle \mathbf{w}, p \rangle |$$

The problem can be formalized as:

Problem: *Retrieve data point(s) $p \in \mathcal{D}$ for which $| \langle \mathbf{w}, p \rangle |$ is minimized.*

Every locality sensitive function has to be defined with a distance function $d(\cdot, \cdot)$ and H-Hash defines the distance function as a measure of how far from perpendicular two points p and q are:

$$d_\theta(p, q) = \left(\theta_{p,q} - \frac{\pi}{2}\right)^2$$

Using the distance function the hash function $h_{\mathbf{u},\mathbf{v}} : \mathbb{R}^d \to \{0, 1\}^2$ is defined as:

$$h_{\mathbf{u},\mathbf{v}}(a, b) = [h_{\mathbf{u}}(a), h_{\mathbf{v}}(b)] = [sign(\langle \mathbf{u}, \mathbf{a} \rangle), sign(\langle \mathbf{v}, \mathbf{b} \rangle)]$$

where $h_{\mathbf{u}}(a) = sign(\langle \mathbf{u}, \mathbf{a} \rangle)$ returns 1 if $\langle \mathbf{u}, \mathbf{a} \rangle \geq 0$, and 0 otherwise, and $\mathbf{u}$ and $\mathbf{v}$ are sampled independently from a standard d-dimensional Gaussian: $\mathbf{u}, \mathbf{v} \sim \mathcal{N}(0, I)$[3].

The H-Hash function family $\mathcal{H}$ is defined in as:

$$h_{\mathcal{H})}(\mathbf{z}) = \begin{cases} h_{\mathbf{u},\mathbf{v}}(\mathbf{z},\mathbf{z}) & \text{if } \mathbf{z} \text{ is a database point vector} \\ h_{\mathbf{u},\mathbf{v}}(\mathbf{z},-\mathbf{z}) & \text{if } \mathbf{z} \text{ is a query hyperplane vector} \end{cases}$$

This family is proven to be $(d_1, d_2, \frac{1}{4} - \frac{1}{\pi^2}d_1, \frac{1}{4} - \frac{1}{\pi^2}d_2)$-sensitive for the distance $d_\theta(\cdot,\cdot)$, given that $d_1, d_2 > 0$[3].

It should be taken into account that this problem is slightly different from the problem of finding top-K closest points below(above) a query hyperplane, here the closest point(s) to the query hyperplane is retrieved regardless of the side on which the point is with respect to the query. In our experiments we look at all retrieved points to determine which are on the positive and which on the negative side.

We should also consider the requirements of the H-Hash method. While the requirement that both the query and the point vectors should be unit length is not restrictive, the requirement that the query vectors have to pass through the origin has to be met. We satisfy this requirement by adding one artificial dimension in the data point vectors, with coordinate set to 1, and treat the query vectors as if they are $(d+1)$-dimensional.

3.3 AND-OR Amplification

Any locality sensitive family $\mathcal{H}$ can be amplified using the **(1)**AND-boosting, **(2)**OR-boosting or their combination [5]. In our experiments we use a combination of these two boosting methods, so we will make a short overview of the theory behind these two methods.

The AND-boosting involves forming hash keys by concatenating multiple hash functions (rows) from $\mathcal{H}$. The new hash function $h(x) = [h_1, h_2, \ldots, h_r]$ would contain r hash functions sampled from $\mathcal{H}$. This new hash function would be (d_1, d_2, p_1^r, p_2^r)-sensitive if all of the h_i hash functions are (d_1, d_2, p_1, p_2)-sensitive. In this case two items would be hashed to the same bucket if the results for all of the h_i hash functions match.

The OR-boosting hashes two items to the same bucket if they match for only one of b hash functions (bands) sampled from $\mathcal{H}$. Specifically, for the points p and q and the hash functions $h_1, h_2, \ldots, h_b$, the points will be hashed together if for any $j \in [1, b]$ it holds that $h_j(p) = h_j(q)$. The resulting hashing scheme would be $(d_1, d_2, 1-(1-p_1)^b, 1-(1-p_2)^b)$-sensitive.

If we first do an AND configuration, then an OR, we would get a hashing function that is $(d_1, d_2, 1-(1-p_1^r)^b, 1-(1-p_2^r)^b)$-sensitive.

Chapter 4

Results

In this section we will present the experimental results involving our implementation of the Planar index for solving the top-k closest points above(below) a query hyperplane. First, we will also include a small-scale experiments set to show the performance of the extended Planar index when solving the original inequality query problem. However, the main goal of this thesis is to solve the top-k problem and all of the consequent experiments will be done for this type of query. We compare the performance of the Planar Index to two methods:

1. Baseline method - exhaustive search over all the data points in the dataset.
2. Hyperplane hash method (H-Hash) for hashing hyperplane queries to near points.

All of the experiments were carried out on a single machine with a Intel Core i5-750 Processor(8M Cache, 2.66 GHz) and 8GB of main memory. For our assessments we implemented the Planar index in the C++ programming language.

For all the experiment sets we use synthetic datasets generated with the generator obtained from [9]. In the **Independent** dataset, all attribute values are generated independently from a pre-defined range with a uniform distribution. The **Correlated** database represents an environment in which points that have higher values in one dimension also have higher values in the other dimensions. In the **Anti-correlated** dataset, points which have higher values in one dimension have lower value(s) in one or all of the other dimensions. The cardinality of each of our synthetic datasets is 1M and we vary the dimensionality of data points from 2 to 14. The range of each attribute lies between (1,100).

All experimental results are averaged over 100 runs.

4.1 Comparison with Baseline (Exhaustive search)

For the baseline comparison, we set up 2 different experiment scenarios that differ in the way that the query and index normal vector coordinates are generated. The first experiment set uses integer coordinates for both the index

and query normal vectors, while the second experiment set uses floating point coordinates for both the index and query normal vectors. The values for the first experiment setting are generated from discrete distributions, i.e. the coordinates are drawn from finite sets of integers. The second experiment setting uses continuous distributions and generates the coordinates as floating point numbers.

For the top-k closest point problem we show the full results for all datasets and varying all parameters in the Appendix. For better visibility, in this chapter we only show extracts of the full results.

For both experiment settings the comparison algorithm in use is a method that performs a sequential scan over all the data points in the dataset.

Algorithm 6 Exhaustive search baseline algorithm

```
1: Top-k priority queue B ← ∅
2: for all i ∈ [1, N] do
3:     scalar_prod = ⟨a, φ(p_i)⟩ − b
4:     if scalar_prod < 0 then
5:         B ← p_i
6:     end if
7: end for
8: return B
```

4.1.1 Queries with Discrete Coordinates

For this experiment setting we consider the problem of finding the top-k nearest points bellow a query given by the equation:

$$\langle \mathbf{a}, \mathbf{x} \rangle < IP \cdot \left(\sum_{i=1}^{d} a_i \max(i) \right)$$

The *inequality parameter* IP is a multiplier that modifies the query selectivity and $max(i)$ denotes the maximum value of the i-th dimension in the data set. If the IP parameter is 0.25 we can expect a small fraction of the data points to be under the query hyperplane, whereas with an IP of 0.75 we can expect a large portion of the data points to be bellow the query hyperplane. All of the plots that we are going to present show results for these two values of the inequality parameter IP. Naturally, we can expect the Planar index' performance to be much better for queries that are more selective($IP = 0.25$), i.e. there are less points below the query hyperplane. This is due to the fact that more points fall into the Larger interval LI and they can be discarded at earlier stages of the query processing.

In Equation 4.1.1, we assume that each query parameter a_i is uniformly selected from a pre-defined domain Δ_i. We denote the size of Δ_i, that is $|\Delta_i|$, as the *randomness of the query* (*RQ*). Particularly, if our data points are d-dimensional, then there are $|\Delta_i|^d$ possible query normal vectors. Here we assume

that the domain is the same for each dimension and the query coordinates are drawn uniformly from the interval $[1, RQ]$. The coordinates of the index normal vectors are also drawn from this interval.

Answering Inequality Queries

The goal of this thesis is to extend the capabilities of the Planar index to answer much broader category of query types denoted as top-K nearest neighbour queries, however this doesn't mean that the index will loose its ability to answer inequality query types like the one described in section 2.1.1. Here we present the results of an experiment done on our framework where the problem setting is as the original problem setting. We want to show that these types of queries also benefit of the extensions made to the Planar index described Chapter 2.

We show results for the following problem, using a synthetic dataset thoroughly described in Chapter 4.

Problem: *Find all data points p from the dataset, which satisfy a scalar product inequality:*

$$\langle \mathbf{a}, \mathbf{x} \rangle < IP \cdot \left(\sum_{i=1}^{d} a_i \max(i) \right)$$

On Figure 4.1 we vary the parameter IP to show the performance of the Planar index in different settings of query selectivity.

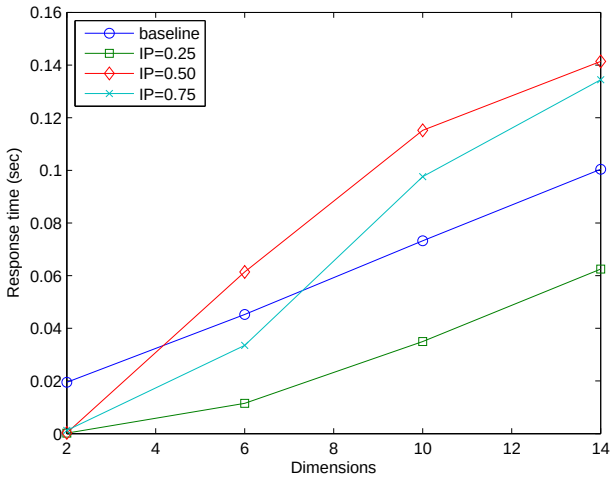
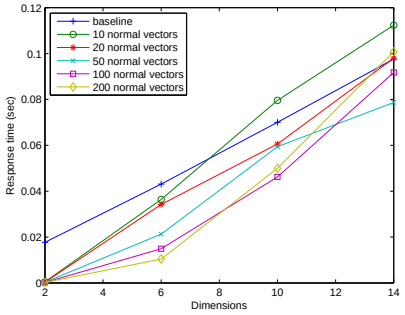


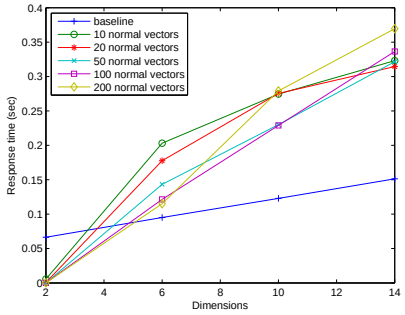
Figure 4.1: Solving the inequality query problem on the Independent dataset with 100 normal vectors, K=1000, RQ=4. The results are for 3 different values of the inequality parameter.

Varying the Number of Normal Vectors

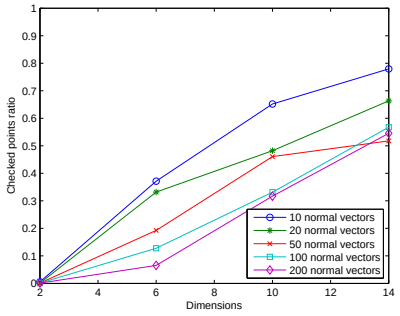
In Figure 4.2 we vary the number of normal vectors that compose the Planar index and we show the effect on the response time compared to baseline and on the *Checked points ratio* $= \frac{inspected\ points}{N}$. For the experiments shown in Figure 4.2 we use the Independent dataset while setting a fixed value for $RQ = 4$. We retrieve the top 1000 points ($K = 1000$). The tests are made with 10,20,50,100 and 200 normal vectors. As expected the response time improves as we increase the number of normal vectors. For highly selective queries ($IP = 0.25$) the Planar index is capable of answering the queries faster than the baseline for up to 13 dimensions with only 20 normal vectors as shown in Figure 4.2-a.



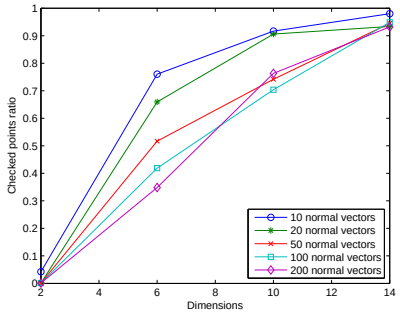
(a) Response time(sec) for IP=0.25



(b) Response time(sec) for IP=0.75



(c) Checked points ratio for IP=0.25



(d) Checked points ratio for IP=0.75

Figure 4.2: Performance of the Planar index solving the top-k problem for 5 different numbers of normal vectors. The randomness of the query RQ is 4, we retrieve the top 1000 closest points from the Correlated dataset.

Although more normal vectors would generally improve the response time, there is a trade off between the number of normal vectors and the indexing time. However, the improvement in the response time diminishes after adding more and more normal vectors. This is also visible in Figure 4.2, when we increase the number of normal vectors from 10 to 20, the performance improves significantly,

but the performance times with 100 and 200 normal vectors seem to converge. It is also important to mention that the memory consumption increases as we increase the number of normal vectors

Figure 4.3 shows the index creation times for the same numbers of normal vectors. As it was already mentioned we are modelling a known query domain and using integer coordinates for the normal vectors. The experiment was run using the duplicates removal technique, this is why the creation time does not increase as the dimensions are increased. For lower dimensions the cardinality of the set of possible normal vectors is smaller, which means that the probability of generating a duplicate vector is higher, which in turn means that more processing time will be needed to regenerate some of the normal vectors. This neutralises the small increase in processing time that stems from the increase in dimension. However, it should be taken into consideration that the change in dimensionality is very small and it should not be assumed that the creation time would stay constant even in the case of dramatic change in dimensionality.

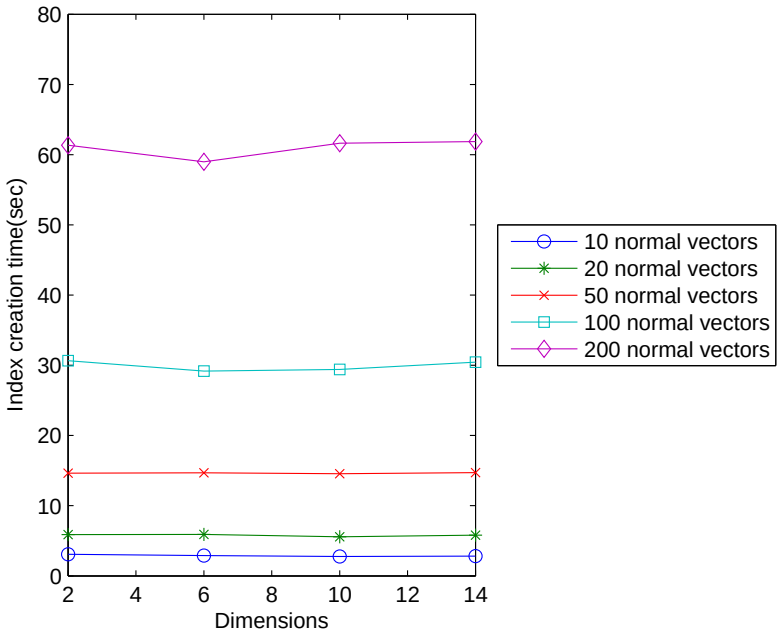
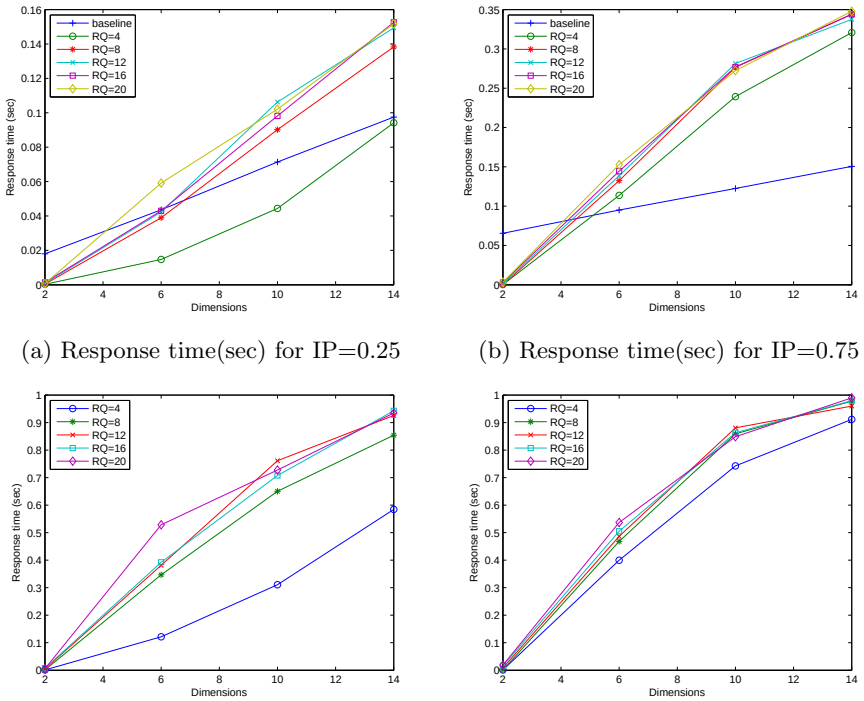


Figure 4.3: Index creation time for normal vectors with integer coordinates

Varying the Randomness of the Query (RQ)

The next set of experiments that we will show involve varying the RQ parameter. For the experimental results shown in Figure 4.4 we use the Independent dataset and set a fixed number of normal vectors of 100, K remains 1000. The

full results for all three datasets can be found in the Appendix. We vary the RQ parameter from 4 to 20 with a step of 4, part of the experimental results are shown in Figure 4.4. The results naturally get worse as we increase the RQ parameter as the cardinality of the set of possible queries increases. For highly selective queries with $IP = 0.25$ the Planar index can outperform the baseline for most RQ values for up to 6 dimensions and for $RQ = 4$ the index outperforms the baseline for all dimensions tested. However, for queries that are not very selective, as expected, the Planar index performs worse. In this case we can see that we can expect the Planar index to outperform the baseline for up to dimensionality of 4.



(a) Response time(sec) for IP=0.25

(b) Response time(sec) for IP=0.75

(c) Checked points ratio for IP=0.25

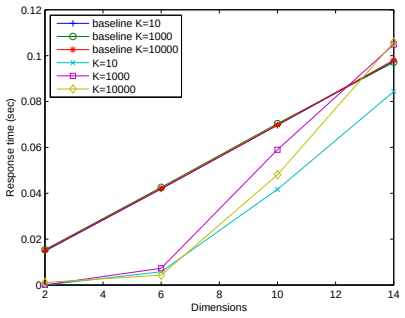
(d) Checked points ratio for IP=0.75

Figure 4.4: Performance of the Planar index solving the top-k problem, varying the RQ parameter. The number of normal vectors is 100, we retrieve the top 1000 closest points from the Independent dataset.

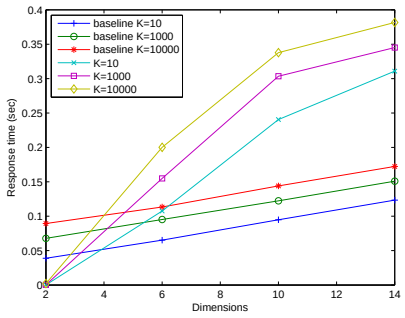
Varying the Top-K parameter

The last parameter that we are going to analyse for the known-domain query type is the top-K parameter. We will discuss the results of the experiments on the Anti-correlated dataset shown in Figure 6.5, while the full results can

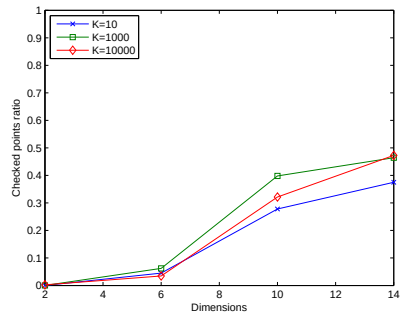
be found in the Appendix. Increasing K results in worse response time since the size of the result priority queue increases. Also more points have to be compared with the contents of the same priority queue. However, the Planar index shows robustness and increasing the K parameter from 10 to 10 000 results in a very small increase in response time. For selective queries the Planar index outperforms the baseline for in most of the cases and the percentage of checked points is less than 50% for all cases. The results are worse for ($IP = 0.75$), where the baseline performs better for dimensionality higher than 4.



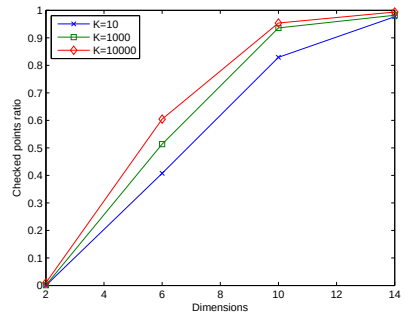
(a) Response time(sec) for IP=0.25



(b) Response time(sec) for IP=0.75



(c) Checked points ratio for IP=0.25



(d) Checked points ratio for IP=0.75

Figure 4.5: Performance of the Planar index solving the top-k problem, varying the top-k parameter. The number of normal vectors is 100, the randomness of the query RQ is 4, we use the Anti-correlated dataset for the shown results.

4.1.2 Queries with continuous coordinates

So far we showed experimental results using integers as coordinates for the query normal vector and we drew each of these coordinates from a discrete uniform distribution over a small set of integers. In this next experiment setting the query normal vector coordinates are generated using floating point numbers

from a continuous uniform distribution. The queries take the form:

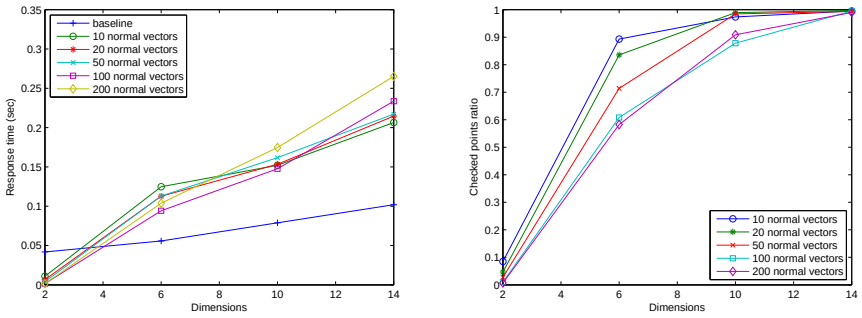
$$\langle \mathbf{q}, \mathbf{x} \rangle < 1$$

$$q_i = \frac{1}{a_i}$$

where a_i is drawn from $\mathcal{U}(0, D \cdot max(i))$. This query setting is very general, furthermore we cannot know anything about the query selectivity in advance. The number of possible query normal vectors is much larger (it is only bounded by the level of precision of representing floating point numbers in a specific implementation) and the probability of having an index normal vector that is equivalent to some query normal vector is very small.

We vary the number of normal vectors for the results in Figure 4.6. In Figure 4.6 we show the results for the Independent dataset and fix the top-K parameter to 1000. The results for all datasets are shown in the appendix.

In this case the Planar index can perform better than the baseline for up to dimensionality of 4, but the processing time grows fast for larger dimensions. We get the best results for 100 normal vectors here. Although the checked points ratio for 100 and 200 normal vectors are very similar, the response time is better for 100 normal vectors, which can be an indicator that the overhead for choosing best index in the case of 200 vectors is too expensive to be justified by the expected improvement in pruning power.



(a) Response time(sec)

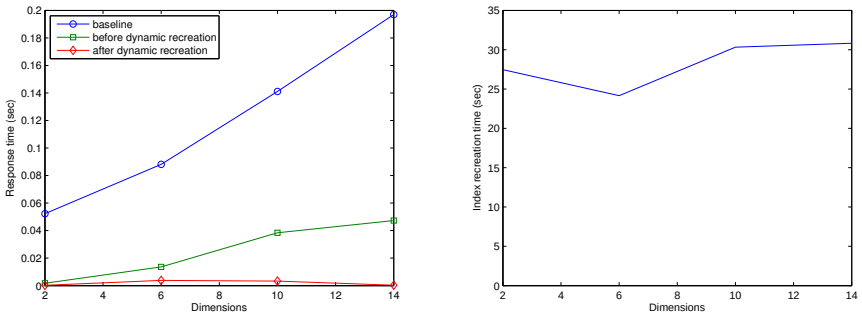
(b) Checked points ratio

Figure 4.6: Performance of the Planar index in a query setting with continuous coordinates (double coordinates), varying the number of normal vectors on the Independent dataset.

4.2 Dynamic updates

The addition of dynamic updates to the Planar index is an easy way to cope with queries that may change their domain. Figure 4.7 shows the performance of the Planar index before some of the normal vectors were recreated and after.

We used the known domain experiment setting described in section 4.1.1 We ran experiments on the Independent dataset, setting the inequality parameter to 0.25, the number of normal vectors to 100 and retrieving the top-1000 nearest points bellow the query. The index normal vectors are generated using the distribution $\mathcal{U}[1, 5]$, while the query normal vectors are generated using the distribution $\mathcal{U}[3, 8]$. We also analyse the recreation time of the Planar index shown in Figure4.7(b).



(a) Response time (sec)

(b) Index recreation time (sec)

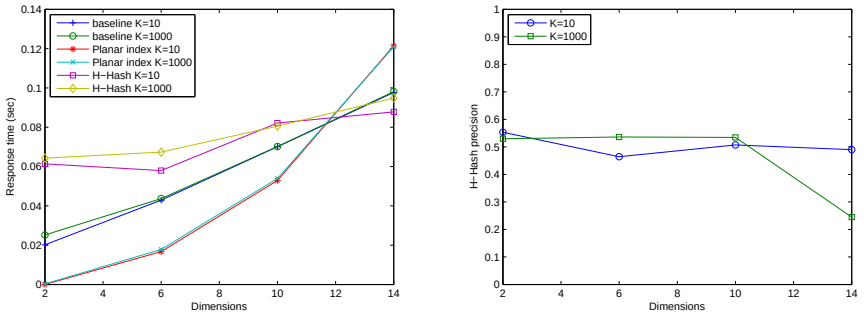
Figure 4.7: Performance of the Planar index with dynamic updates

This experiment was run using the duplicates removal technique and this reflects in the index recreation time as it is visible from the plot. The processing time of the duplicates removal technique depends on the generated index coordinates which means that it involves randomness, so we can expect the standard deviation of the results to be much higher than in a deterministic setting. Also, for lower dimensions the cardinality of the set of possible normal vectors is smaller, which means that the probability of generating a duplicate vector is higher. This explains the smaller running time for 6 dimensions compared to the result for 2 dimensions, since the probability of generating a duplicate in 2 dimensions is very high.

4.3 Comparison with H-Hash

The H-Hash method was presented in Chapter 3. Here we compare our implementation of the H-Hash method which is boosted with AND-OR amplification. This allows for making trade-of between the precision and the performance. By increasing the number of rows we increase the performance, but decrease the precision. By increasing the number of bands we increase the precision but decrease the performance. For our experiments shown in Figure 4.8 we use 50 bands and 3 pairs of rows per band. The queries are generated as known domain queries with $RQ = 4$ and $IP = 0.25$. The Planar index has 100 normal vectors. We use the Correlated dataset for this experiment. H-

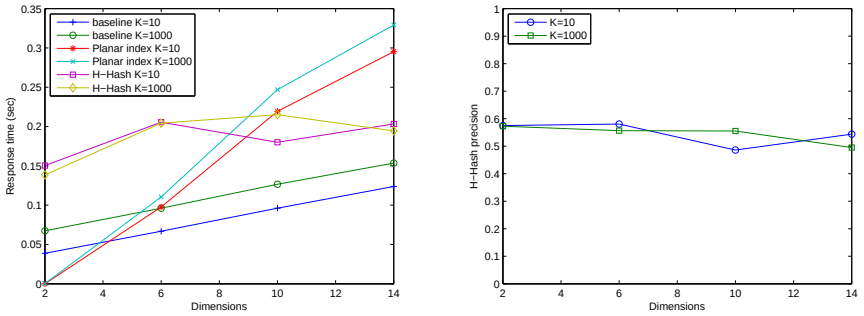
Hash is extremely robust on dimensionality and even though it is not as useful on synthetic data, in the high dimensions outperforms both the baseline and the Planar index. However, because of the probabilistic nature of the H-Hash method a trade of has to be made with the precision of the method.



(a) Response time (sec) (b) H-Hash precision

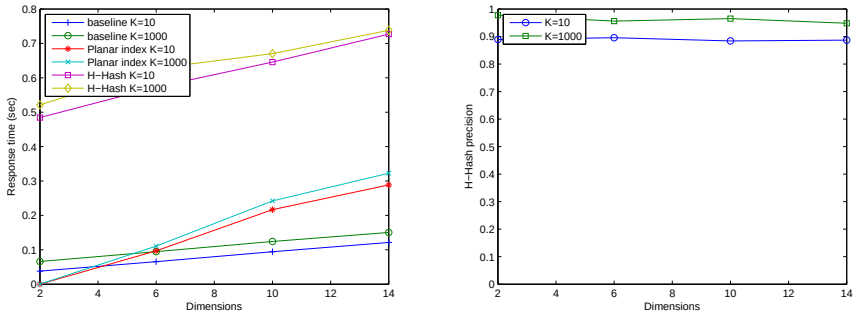
Figure 4.8: Comparison with H-Hash IP=0.25

On figure 4.9 we use the same experiment setting, but we only changed the inequality parameter to 0.75. We can see that H-Hash is more robust on the query selectivity variation than the Planar index. However precision is still a major advantage for the Planar index, since, if we want to get decent precision from H-Hash on our dataset we will get the results from figure 4.10. These results are achieved with 30 bands and 2 pairs of rows for the H-Hash.



(a) Response time (sec) (b) H-Hash precision

Figure 4.9: Comparison with H-Hash IP=0.75



(a) Response time (sec) (b) H-Hash precision

Figure 4.10: H-Hash configuration for high precision

Chapter 5

Conclusion

The indexing scheme that we presented in this thesis is very general on the query types that can be answered. The index can not only solve the top-k query problem for different types of inequalities, but also can use the same data structures for answering both top-k closest points queries and top-k farthest points queries. Among the powerful features of the indexing scheme that we presented here, other than the top-k capabilities, are: different techniques for the generation of the index normal vectors and dynamic updating of the index normal vectors according to the queries received. The extension to multiple normal vectors allows us to parametrize the memory consumption of the index. Additionally we support all the basic capabilities of the Parallel Planar index as proposed by Yanki[6].

We can conclude from our extensive experiments that the Planar index is very capable of answering queries that are highly selective and come from a known discrete domain. The index doesn't handle well queries with very low selectivity, but can still perform well even with these types of queries for 2,3 or 4 dimensional data. The Planar index showed robustness when varying the top-k parameter and its performance didn't suffer significantly even for high values of k. The comparison with the H-Hash indicates that the Planar index is a good approach for non-normalized data.

5.1 Future work

Since the Planar index showed good results in low dimension, but did not scale good enough in high dimension, it is a good candidate for competing with space partitioning methods that are used in many areas of: computer graphics, physical simulation, representations of geographical data and geographic information systems. Comparison with a space partitioning method might give us some insight about even more possibilities of using the Planar index.

Another area in which we can take the research further is the dynamic updates and expanding the query types from which the Planar index can learn

its best configuration. Possible query types might be queries whose coordinates are generated from a different probability distribution than the uniform distribution.

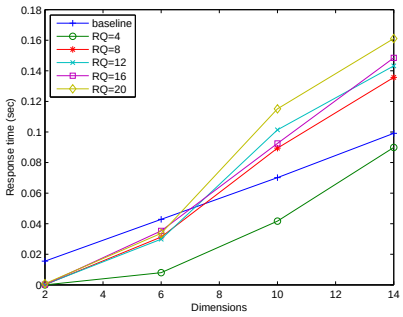
Chapter 6

Appendix

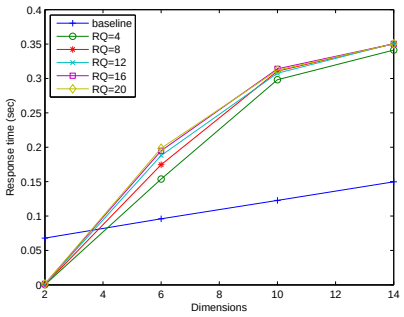
6.1 Experimental results

In this chapter we show the extensive results for the baseline comparison that were too sizeable to show in the results section. The figures are organised by the feature that they show, containing the results for that feature for multiple datasets with different parameters. We show the response time and checked points ration depending on the variation of different parameters.

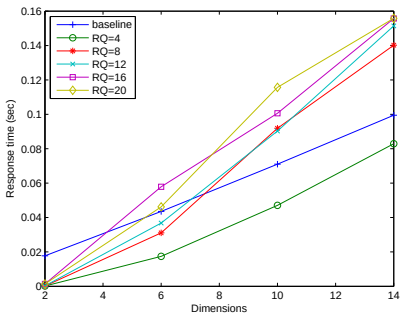
All of the figures maintain a similar layout. In the first row we show the plots for the Anti-correlated dataset, in the second row the plots for the Correlated dataset, and in the third row the plots for the Independent dataset. The columns for the discrete query types contain results for $IP = 0.25$ (in the first column), and results for $IP = 0.75$ (in the second column). The continuous query plots are organised by the measured statistic: response time in seconds(first column); and checked points ratio (second column).



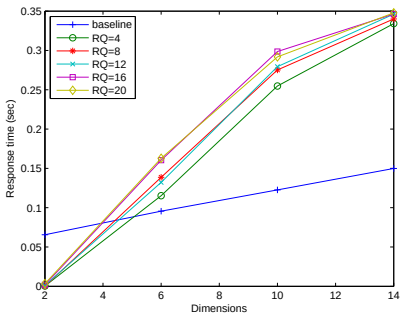
(a) Anti-correlated IP=0.25



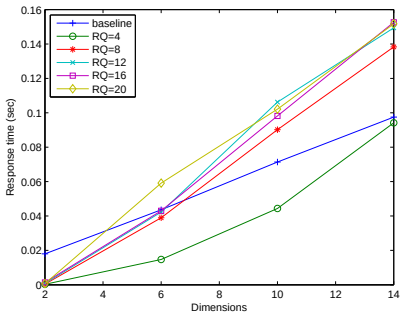
(b) Anti-correlated IP=0.75



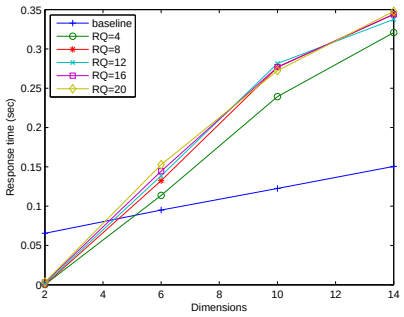
(c) Correlated IP=0.25



(d) Correlated IP=0.75

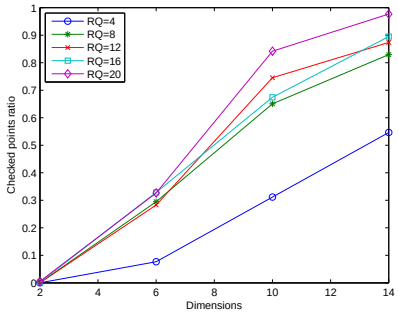


(e) Independent IP=0.25

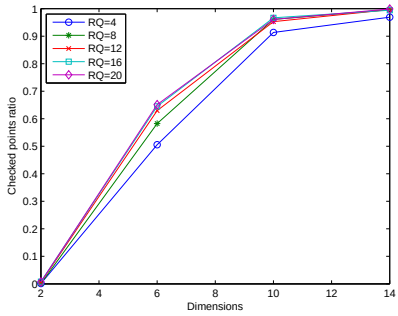


(f) Independent IP=0.75

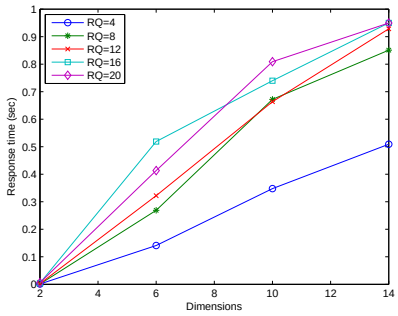
Figure 6.1: Response time Planar Index solving the top-k problem, varying the RQ parameter. The number of normal vectors is 100, we retrieve the top 1000 closest points.



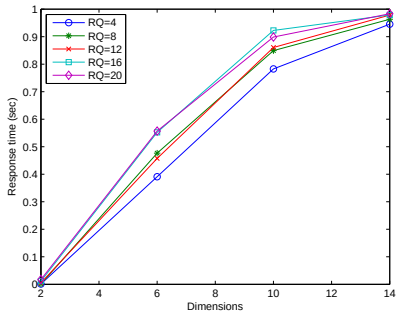
(a) Anti-correlated IP=0.25



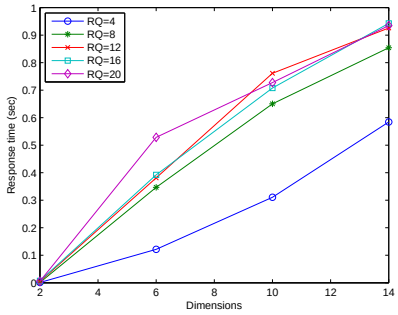
(b) Anti-correlated IP=0.75



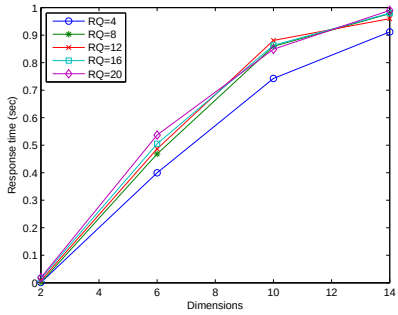
(c) Correlated IP=0.25



(d) Correlated IP=0.75

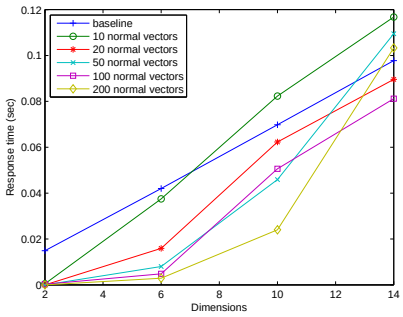


(e) Independent IP=0.25

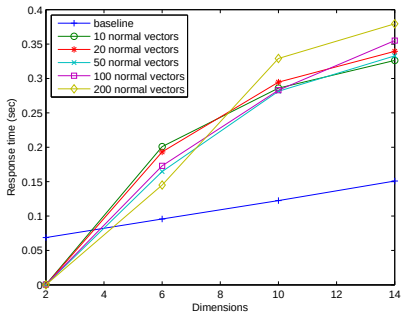


(f) Independent IP=0.75

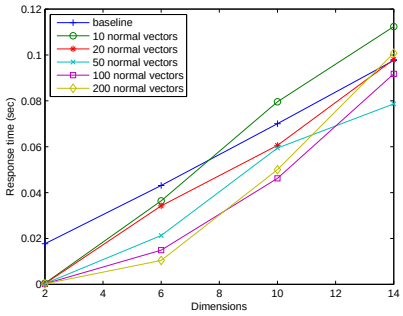
Figure 6.2: Checked points ratio Planar Index solving the top-k problem, varying the RQ parameter. The number of normal vectors is 100, we retrieve the top 1000 closest points.



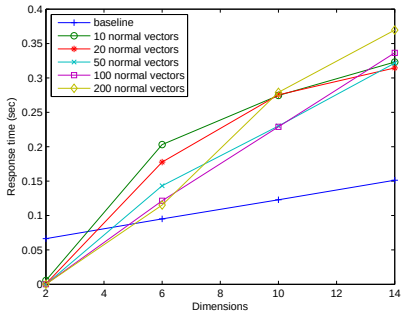
(a) Anti-correlated IP=0.25



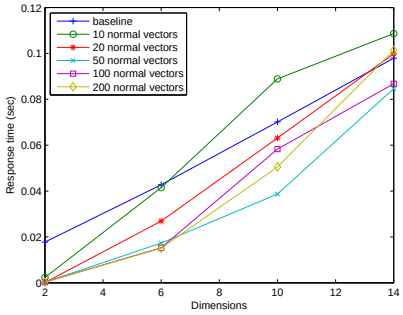
(b) Anti-correlated IP=0.75



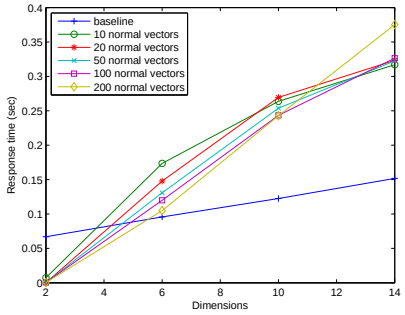
(c) Correlated IP=0.25



(d) Correlated IP=0.75

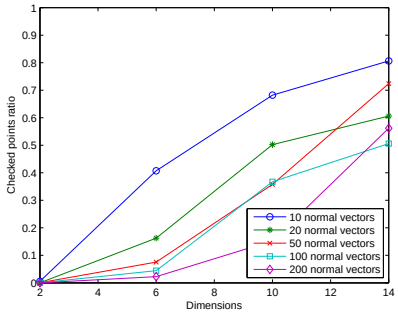


(e) Independent IP=0.25

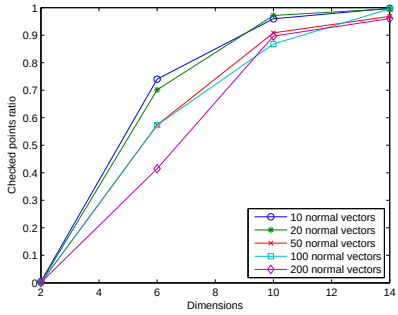


(f) Independent IP=0.75

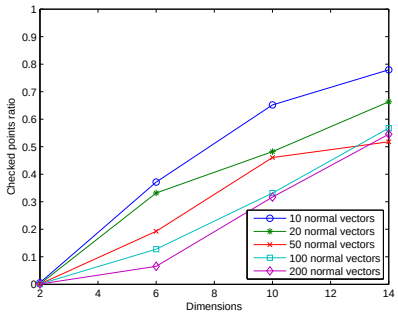
Figure 6.3: Response time Planar Index solving the top-k problem. Results for 5 different numbers of normal vectors. The randomness of the query RQ is 4, we retrieve the top 1000 closest points.



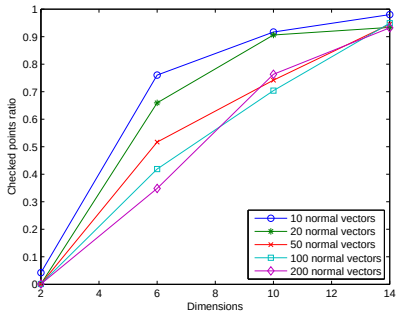
(a) Anti-correlated IP=0.25



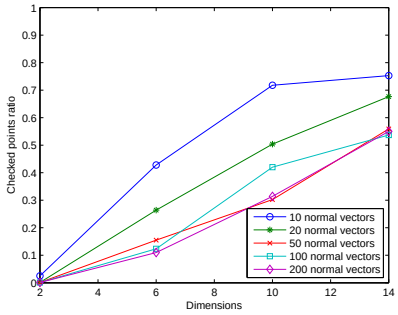
(b) Anti-correlated IP=0.75



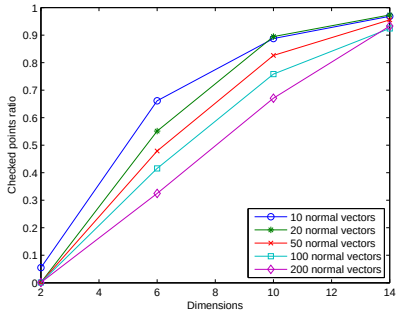
(c) Correlated IP=0.25



(d) Correlated IP=0.75

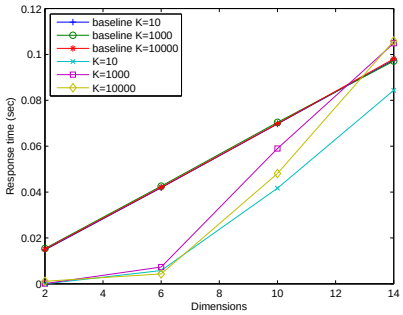


(e) Independent IP=0.25

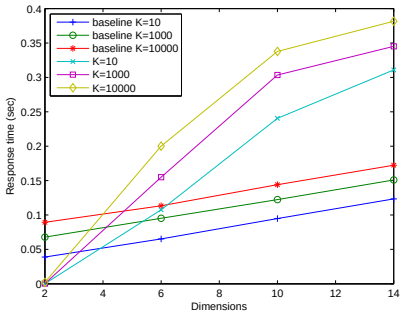


(f) Independent IP=0.75

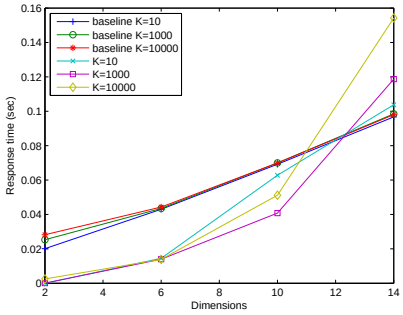
Figure 6.4: Checked Points Ratio Planar Index solving the top-k problem. Results for 5 different numbers of normal vectors. The randomness of the query RQ is 4, we retrieve the top 1000 closest points.



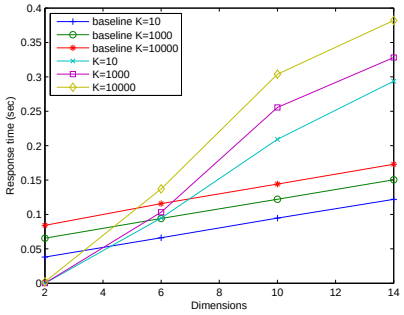
(a) Anti-correlated IP=0.25



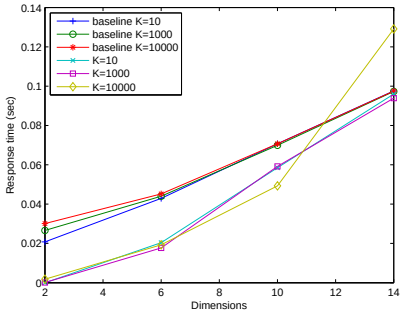
(b) Anti-correlated IP=0.75



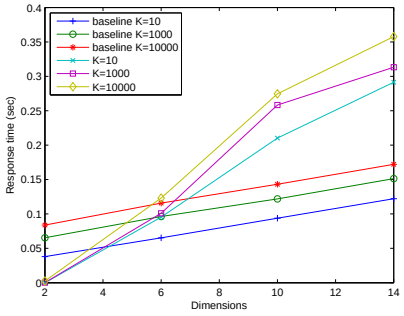
(c) Correlated IP=0.25



(d) Correlated IP=0.75

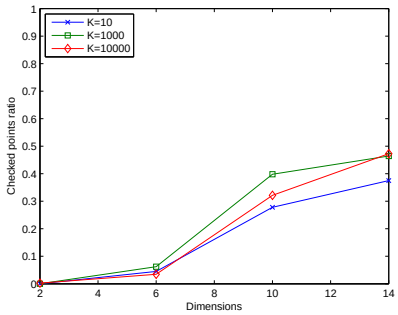


(e) Independent IP=0.25

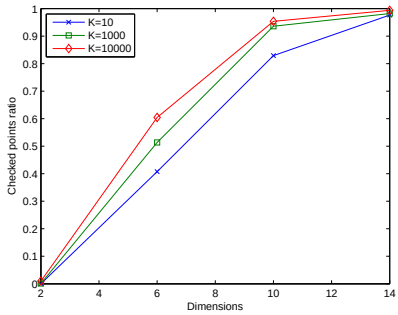


(f) Independent IP=0.75

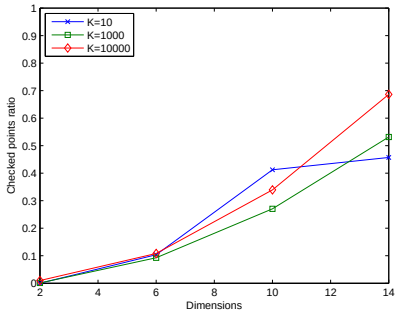
Figure 6.5: Response time Planar Index varying the top-k parameter. The number of normal vectors is 100, the randomness of the query RQ is 4.



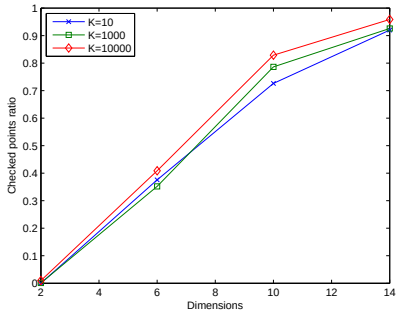
(a) Anti-correlated IP=0.25



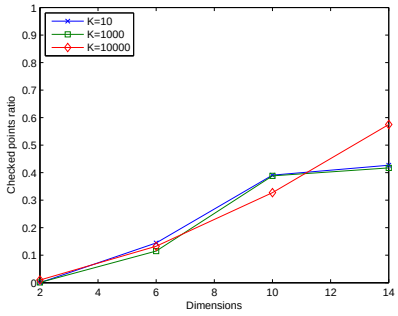
(b) Anti-correlated IP=0.75



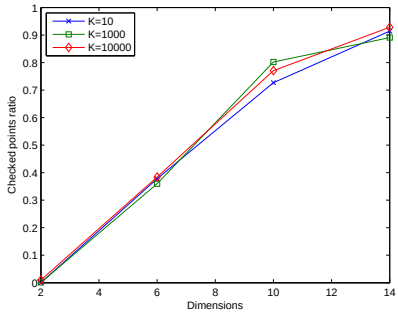
(c) Correlated IP=0.25



(d) Correlated IP=0.75

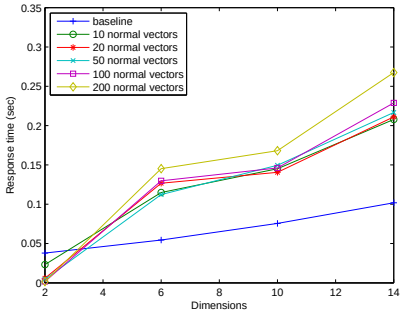


(e) Independent IP=0.25

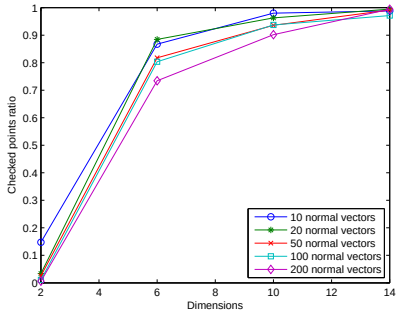


(f) Independent IP=0.75

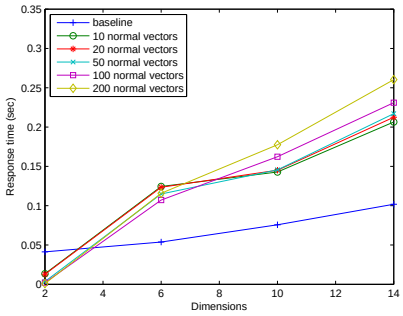
Figure 6.6: Checked points ratio Planar Index varying the top-k parameter. The number of normal vectors is 100, the randomness of the query RQ is 4.



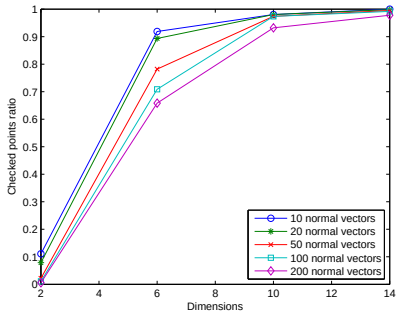
(a) Anti-correlated - Response time



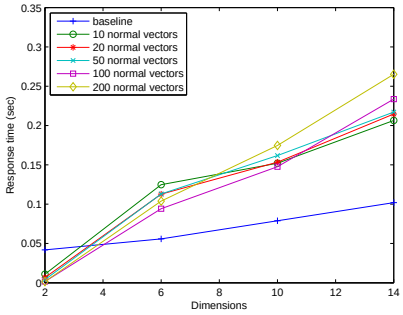
(b) Anti-correlated - Checked points ratio



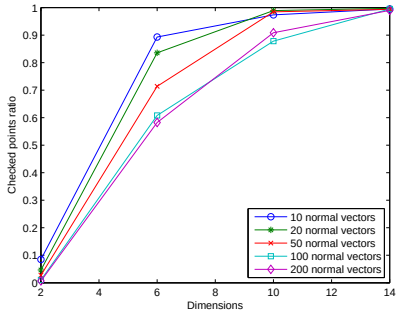
(c) Correlated - Response time



(d) Correlated - Checked points ratio



(e) Independent - Response time



(f) Independent - Checked points ratio

Figure 6.7: Performance of the Planar index in a query setting with continuous coordinates (double coordinates), varying the number of normal vectors. We retrieve top 1000 closest points.

Bibliography

[1] *A Note on a Method for Generating Points Uniformly on N-Dimensional Spheres.*

M. E. Muller, Comm. Assoc. Comput. Mach. 2, 19-20, 1959

[2] *Compact Hyperplane Hashing with Bilinear Functions.* W. Liu, J. Wang, Y. Mu, S. Kumar, and S. Chang, ICML 2012

[3] *Hashing Hyperplane Queries to Near Points with Applications to Large-Scale Active Learning.* P. Jain, S. Vijayanarasimhan, and K. Grauman, NIPS 2010

[4] *Hypersphere Point Picking.*

Weisstein, Eric W. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HyperspherePointPicking.html

[5] *Mining of Massive Datasets.* A. Rajaraman, and J. D. Ullman, Cambridge University Press New York, NY, USA 2011

[6] *Parallel Planar Indexing: An innovative approach for indexing SQL functions with unpredictable parameters.*

P. Yanki, Master's Thesis Nr. 91, Systems Group, ETH Zurich,2013.

[7] *Similarity Search in High Dimensions via Hashing.*

A. Gionis, P. Indyk, and R. Motwani, VLDB 1999

[8] *Sphere Point Picking* Weisstein, Eric W. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/SpherePointPicking.html

[9] *The Skyline Operator.*

S. Borzsonyi, D. Kossmann and K. Stocker, ICDE, 2001

[10] *Towards Indexing Functions: Answering Scalar Product Queries.* A. Khan, P. Yanki, B. Dimcheva, and D. Kossmann, SIGMOD, 2014