Towards Indexing Functions: Answering Scalar Product Queries

Arijit Khan, Pouya Yanki, Bojana Dimcheva, Donald Kossmann
Systems Group, ETH Zurich, Switzerland
{arijit.khan, donaldk}@inf.ethz.ch {pyanki,dbojana}@student.ethz.ch

ABSTRACT

We consider a broad category of analytic queries, denoted by scalar product queries, which can be expressed as a scalar product between a known function over multiple database attributes and an unknown set of parameters. More specifically, given a set of d-dimensional data points, we retrieve all points $\mathbf{x}$ which satisfy an inequality given by a scalar product: $\langle \mathbf{a}, \phi(\mathbf{x}) \rangle \leq b$. We assume that the function $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$ is application specific and known apriori, while the query parameters $\mathbf{a}$ and the inequality parameter b are known only at the time of querying.

Efficiently answering such scalar product queries are essential in a wide range of applications including evaluation of complex SQL functions, time series prediction, scientific simulation, and active learning. Although some specific subclasses of the aforementioned scalar product queries and their applications have been studied in computational geometry, machine learning, and in moving-object queries, surprisingly no generalized indexing scheme has been proposed for efficiently computing scalar product queries.

We present a lightweight, yet scalable, dynamic, and generalized indexing scheme, called the Planar index, for answering scalar product queries in an accurate manner, which is based on the idea of indexing function $\phi(\mathbf{x})$ for each data point $\mathbf{x}$ using multiple sets of parallel hyperplanes. Planar index has loglinear indexing time and linear space complexity, and the query time ranges from logarithmic to being linear in the number of data points. Based on an extensive set of experiments on several real-world and synthetic datasets, we show that Planar index is not only scalable and dynamic, but also effective in various real-world applications including intersection finding between moving objects and active learning.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Search and Retrieval

Keywords

Scalar Product Query, Function Indexing, Planar Index, Moving Object Indexing

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2610493 .

1. INTRODUCTION

In a wide range of complex data analytic applications, the query processing often requires computing an expression that involves multiple columns in the relational database [21]. In this study, we consider a broad category of queries which can be expressed as the scalar product between a known expression (function) over multiple database attributes and an unknown set of parameters. We refer to such queries as the *scalar product* queries, and they can be applied in evaluating complex SQL functions [21], finding intersection between moving-object pairs [33], time-series prediction [5], scientific simulations [25], half-space range searching [1], and also in machine learning [18]. Below, we demonstrate two applications of scalar product queries in processing complex SQL functions and in finding intersection between moving objects.

EXAMPLE 1 (COMPLEX SQL QUERY PROCESSING). *Consider an electricity consumption dataset for individual households. Each data point (household) has 4 important attributes:* active power, reactive power, voltage, *and* current. *The* power factor *over this electricity consumption dataset can be measured as the ratio between* active power *and* (voltage $\times$ current). *Thus, given a relation*

Consumption(ID, Active Power, Reactive Power, Voltage, Current)

we are interested in the following SQL *function,* Critical_Consume *that identifies all households for which the* power factor *is less than an input threshold.*

```
CREATE FUNCTION Critical_Consume (
INPUT double threshold
RETURN ID
FROM Consumption
WHERE Active Power - threshold * Voltage *
Current ≤ 0)
```

The aforementioned SQL function can be modeled as a scalar product query:

$\langle (1 \quad \text{-threshold}), (\text{ActivePower} \quad \text{Voltage} \times \text{Current}) \rangle \leq 0.$

One may note that the above-mentioned scalar product consists of two components — a function over the database attributes: ϕ(Active Power Voltage Current) = (Active Power Voltage $\times$ Current) and a parameter set (1 -threshold). The functional part of the query is known apriori; and hence, can be indexed — while the parametric part is known only at the time of querying. *Can we answer such queries in sublinear time without performing a sequential scan over the entire dataset?* It is worthwhile to mention that Oracle 11.1 release has built-in support for indexing complex SQL functions over multiple attributes [21]. However, their index does not support queries that consist of both complex functions as well as unknown parameters.

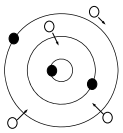


Figure 1: Application of scalar product query: finding intersection between moving-object pairs

EXAMPLE 2 (MOVING-OBJECT INTERSECTION). *Assume two sets of objects are moving in a two dimensional plane as depicted in Figure 1: objects from one set are moving in concentric circles, while objects from the other set are moving in straight lines. For simplicity, we assume that the objects are moving in constant linear or angular velocities. Given an input time-instant t in the future and an input distance S, find the object pairs from the two sets which will be within S unit distance from each other at time t.*

The aforementioned intersection query is critical in scientific simulations, air traffic control, and in massively multi-player online games (MMOGs). A naïve approach to solve such intersection problem will be to compute the distance for each object pair at time t. Let us denote the angular velocity and radius of an object from the first set as r and ω, and the initial position and velocity of an object from the second set by p_x, p_y, u_x, and u_y, respectively. In order to find intersections, we have to verify for each object pair from the two sets whether Equation 1 holds true.

$$AX_1 + BX_2 + CX_3 + DX_4 + EX_5 + FX_6 + GX_7 \leq S^2 \tag{1}$$

Nevertheless, Equation 1 has the form of a scalar product query; and therefore, one can apply our indexing method to evaluate such queries more efficiently. It is easy to verify that the functional part consists of $X_1 = r^2 + p_x^2 + p_y^2 + 2rp_x + 2rp_y$, $X_2 = 2[u_x(r + p_x) + u_y(r + p_y)]$, $X_3 = -2rp_x$, $X_4 = -2rp_y$, $X_5 = -2ru_x$, $X_6 = -2ru_y$, and $X_7 = u_x^2 + u_y^2$. The parametric part, on the other hand, contains $A = 1$, $B = t$, $C = 1 + \sin \omega t$, $D = 1 + \cos \omega t$, $E = t(1 + \sin \omega t)$, $F = t(1 + \cos \omega t)$, and $G = t^2$. We also note that there exist several spatio-temporal indexes for moving-object databases (MOD) [8, 23, 32], but their application is limited to objects moving in straight lines with uniform velocities. Therefore, scalar product queries and the subsequent indexing method proposed in this study are more general and widely-applicable in scenarios such as objects moving in circles or with acceleration.

Scalar product queries naturally arise in a variety of machine learning applications as well, e.g., pool-based active learning [26]: given a classifier hyperplane, the query finds the class labels of all unlabeled data points. More precisely, points in one side of the query hyperplane are labeled positive and points in the other side are considered negative; in addition, the query also requires the identification of the top-k closest positive and negative points to the classifier hyperplane [14, 18]. An important subclass of scalar product queries is the half-space range searching query, which has been studied extensively in computational geometry [1, 2, 19]. In spite of many critical applications, surprisingly no generalized indexing scheme has been proposed to answer scalar product queries in an online and accurate manner.

In this work, we study the problem of fast online computation of scalar product queries in an accurate manner. To this aim, we devise a novel, lightweight, and generalized indexing scheme, called the Planar index, which can answer our queries very efficiently. Our offline technique relies on indexing functions $\phi(\mathbf{x})$ for data points $\mathbf{x}$ with multiple sets of parallel hyperplanes, and pre-computing some information which is linear in the number of the data points. Our online query evaluation consists of finding the optimal set of index hyperplanes for a given query, and then using the pre-computed information to efficiently answer our queries in an exact manner. The key idea of Planar index is to allow very fast pruning of the data points without actually computing the scalar product for them. Our best case query time is logarithmic in the number of data points, which is often the case for carefully designed Planar indices, and is also verified by performing empirical analysis over multiple datasets and considering several real-world applications.

Our contribution and roadmap. Our contributions can be summarized as follows:

- We define the fundamental problem of efficiently answering scalar product queries (Section 3).
- We devise a lightweight, yet effective and generalized indexing scheme, called the Planar index, for efficiently answering scalar product queries in an online and exact manner. (Section 4). The proposed technique is based on indexing function $\phi(\mathbf{x})$ with multiple sets of parallel hyperplanes and pre-computing some information that is linear in the number of data points.
- Based on the Planar index, we develop a fast *pruning-and-verification* strategy. Given a scalar product query, we first determine the optimal set of index hyperplanes (Section 5), and then use our indexed information to accurately accept or reject several data points without even computing the scalar product for them. In Section 6, we show how our proposed Planar index, coupled with a lower-bound-based pruning method, efficiently retrieves the top-k closest points to a query hyperplane.
- We conduct a thorough experimental evaluation using several real-world and synthetic datasets. We also compare Planar index with a naïve sequential scan. Results attest efficiency and accuracy of the Planar index (Section 7).
- We analyze the performance of Planar index in the moving-objects-intersection problem [33] with both uniform (constant velocity) and non-uniform (moving with acceleration) workloads, as well as in active learning [14, 18], and thereby compare our generalized framework with state-of-the-art methods tailored for such specific applications. (Section 7.5).

2. RELATED WORK

We categorize related work as follows.

Half-space range searching. Given a fixed set S of data points in $\mathbb{R}^d$ and a query hyperplane q, the half-space range searching problem asks for the retrieval of all points of S on a chosen side of q. In computational geometry, the half-space range searching problem for d-dimension has been considered in [1,2,19]. We compare their space and time complexity with that of ours in Table 1. Unfortunately, all these previous works study the problem with respect to the *asymptotic* computational complexity — it is very difficult to implement their preprocessing steps, unlike the lightweight Planar index proposed in this work — and to the best of our knowledge, no implementations of [1, 2, 19] exist in the literature. Besides, our Planar index is more general and can be used in a variety of applications beyond half-space range searching, such as moving-objects-intersection computation [33] and hyperplane-to-closest-point finding [18], which is critical in active learning [26].

Linear constraint queries. In linear constraint queries, the search region is constrained to the intersection of half spaces specified by a set of linear inequalities. The orthogonal range searching problem was discussed in [24], in which the query is a d-dimensional

Table 1: *Time complexity of half-space range search algorithms: n number of data points, d dimensionality of the query space, t cardinality of the answer set, $\epsilon > 0$ any constant, $c = c(d)$ another constant.*

	Query time	*Preprocessing storage*	*Preprocessing time*
Agarwal et. al. [1]	$\mathcal{O}(n^{1-\frac{1}{d}+\epsilon}+t)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Matousek et. al. [19]	$\mathcal{O}(n^{1-\frac{1}{\lfloor d/2 \rfloor}}(\log n)^c + t)$	$\mathcal{O}(n \log \log n)$	$\mathcal{O}(n \log n)$
Arya et. al. [2]	$\tilde{\Omega}(\frac{n^{1-\frac{1}{d+1}}}{m^{\frac{1}{d+1}}}+t)$	$\tilde{\Omega}(m); \quad n \leq m \leq n^d$	$\mathcal{O}(n^{1+\epsilon}+m(\log n)^\epsilon)$
Planar index [this work]	$\mathcal{O}(d \log n + t) \sim \mathcal{O}(dn)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$

axis parallel hyper rectangle; while non-orthogonal range searching queries are considered in [11]. Most studies in linear constraint queries apply spatial data structures such as R-tree and K-D-B tree. As an alternative, one could also apply multiple Planar indices in answering such linear constraint queries.

Nearest neighbor queries. Efficiently finding the top-k nearest points to a given query point has been studied both in low dimension [29] and in high dimension [31]. The problem of finding the nearest subspace from a query point is considered in [3]. Finally, [14, 18] proposed hashing-based approximate methods to find the closest point to a query hyperplane, which has application in active learning [26]. In contrast to the *approximate* methods in [14,18], Planar index efficiently finds the top-k closest points in an *accurate* manner for any input value of k.

Top-k queries with ranking function. The top-k queries retrieve the top-k tuples ordered according to a user-defined ranking function that combines the values from multiple attributes. Fagin et. al. proposed the well-known Threshold algorithm for efficiently computing the top-k queries [10] with monotonic ranking function. The top-k retrieval problem with ad-hoc ranking functions has been studied in [30]. Li et al. optimized the top-k query processing which requires joining of multiple relations [17]. For a survey on top-k query processing, see [13]. A subclass of top-k queries is the linear optimization query, where the sum of linearly weighted attribute values is calculated as the ranking criterion [6, 12]. A very relevant work to ours is [22], which maximizes a scalar product search using a tree-structured index. Nevertheless, our scalar product queries are different from the above-mentioned top-k queries — while the top-k queries identify the top-k data points that maximize a ranking function, our objective is to retrieve all (or, top-k closest) data points which satisfy a given inequality.

Index for moving objects. One of the earliest work in indexing moving objects is the historical R-Tree [20], which indexes the sampled locations of a moving object using an R-tree. Sistla et. al. prototyped moving-object trajectories using a linear function of time [27]. Kollios et. al. [16] applied the dual transform to map a one dimensional trajectory to a point and then used spatial indices to answer window queries. The TPR-tree [23] is an extension of R*-tree to manage moving objects. The B^x-tree [15] indexes moving objects by a B^+-tree using space-filling curves. For a survey on indexing moving-object databases, see [7]. Time-parameterized-join algorithms and moving-objects-intersection queries are studied in [28] and [33], respectively. In all these studies, it has been assumed that objects tend to move in a linear fashion with constant velocities, and an index update is required when an object changes its velocity or direction. Thus, it is difficult to apply state-of-the-art indexing schemes for more complex and non-uniform motions, such as objects moving with acceleration or in a circle.

3. PROBLEM STATEMENT

Given a set of data points in $\mathbb{R}^d$ and an application specific function $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$, we define two novel scalar product queries.

PROBLEM 1 (INEQUALITY QUERY). *Find all data points $\mathbf{x} \in \mathbb{R}^d$, which satisfy a scalar product inequality: $\langle \mathbf{a}, \phi(\mathbf{x}) \rangle \leq b$.*

PROBLEM 2 (TOP-k NEAREST NEIGHBOR QUERY). *Given some k, find the top-k data points $\mathbf{x}$ satisfying $\langle \mathbf{a}, \phi(\mathbf{x}) \rangle \leq b$, which also minimize $\frac{|\langle \mathbf{a}, \phi(\mathbf{x}) \rangle - b|}{|\mathbf{a}|}$.*

Remarks. (1) Both the query parameters $\mathbf{a} \in \mathbb{R}^{d'}$ and the inequality parameter $b \in \mathbb{R}$ are known only at the time of querying. **(2)** Instead of "less than or equal" constraint, one may also have "greater than or equal" constraint in the aforementioned queries. Nevertheless, our indexing scheme is general enough to address both types of constraints. **(3)** When the function ϕ is an identity function, our inequality query (Problem 1) reduces to the half-space range searching problem [1,2,19], while the top-k nearest neighbor problem (Problem 2) becomes identical with the hyperplane-to-nearest-point query [14, 18].

In this study, our objective is to propose a generalized indexing scheme — which is easily maintainable and updatable — and which enables faster processing of both the scalar product queries in an accurate manner.

Since the query parameters $\mathbf{a}$ and inequality parameter b are not known apriori, a naïve approach to solve these problems will be to perform a sequential scan over the entire dataset. Such a naïve scan requires $\mathcal{O}(nd')$ time for the inequality query and $\mathcal{O}(nd' + k \log k)$ time for our top-k nearest neighbor problem, where n is the total number of data points, and d' is the dimensionality of the output of function ϕ. In Section 4, we introduce our indexing technique which helps in efficiently answering both the scalar product queries. For the sake of clarity, we first consider the inequality query (Problem 1) in Sections 4 and 5, then we describe how to answer the top-k nearest neighbor query (Problem 2) in Section 6.

4. THE PLANAR INDEX: OVERVIEW

We shall provide a brief overview of our Planar index in this section. Given an application specific function $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$, let us consider a d'-dimensional Euclidean co-ordinate system with axes $(Y_1, Y_2, \ldots, Y_{d'})$. Recall that our scalar product query has the form $q : \langle \mathbf{a}, \phi(\mathbf{x}) \rangle \leq b$, and assume $\mathbf{a} = (a_1, a_2, \ldots, a_{d'})$. We consider a query hyperplane $H(q)$ in $\mathbb{R}^{d'}$ as follows.

$$H(q) : a_1 Y_1 + a_2 Y_2 + \ldots + a_{d'} Y_{d'} = b \tag{2}$$

The normal to the query hyperplane $H(q)$ is given by the vector $\mathbf{a} = (a_1, a_2, \ldots, a_{d'})$.

4.1 Domain of Query Parameters

The exact query parameter values in a scalar product query are unknown apriori. Nevertheless, over a period of time, it is often easy to identify the domains of those parameters as follows. **(1)** one may learn the domain Δa_i for each query parameter a_i based on the past queries, and dynamically update their domains with time. **(2)** Often, the parameter domains are application specific. For example, in the moving-objects-intersection problem (Example 2), it

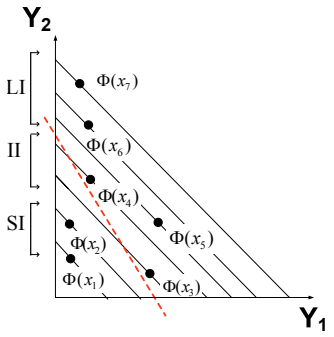


Figure 2: Examples of smaller (SI), intermediate (II), and larger (LI) intervals: the query hyperplane is shown as the dashed line.

is more interesting to find the intersecting pairs at a relatively near time, say from $t = 10$ to $t = 15$. Thus, one can still have an apriori knowledge about the parameter domains for moving-objects-intersection queries, and we update these domains with time. On the other hand, for the complex SQL function over the electricity consumption dataset (Example 1), the power ratio by default lies between 0 and 1. Intuitively, the larger the domains of the query parameters are, the more random the query becomes.

For simplicity, we herein make two assumptions:

- $a_i \neq 0$ for all $1 \leq i \leq d'$. Otherwise, one can simply ignore the corresponding axis during index construction and query processing.
- $a_i > 0\ \forall i$ and $b \geq 0$. Similarly, all of $\phi(\mathbf{x})$ are in the first hyper octant of $\mathbb{R}^{d'}$. Otherwise, we perform a translation of the co-ordinates as discussed in Section 4.5.

4.2 Index Construction

The core of our Planar index is a collection of parallel hyperplanes in $\mathbb{R}^{d'}$ with a unique normal vector $\mathbf{c} = (c_1, c_2, \ldots, c_{d'})$, where each c_i is sampled uniformly from the domain Δa_i of query parameter a_i. We consider n such parallel hyperplanes — one for each data point $\mathbf{x}$, as defined in Equation 3.

$$H(\mathbf{x}) : c_1 Y_1 + c_2 Y_2 + \ldots + c_{d'} Y_{d'} = \langle \mathbf{c}, \phi(\mathbf{x}) \rangle \tag{3}$$

Next, our indexing phase consists of sorting all data points $\mathbf{x}$ in a list $\mathcal{L}$ in ascending order of $\langle \mathbf{c}, \phi(\mathbf{x}) \rangle$ values. Let us denote by $\mathcal{L}(j)$, $1 \leq j \leq n$, the data point $\mathbf{x}$ with the j-th smallest value of $\langle \mathbf{c}, \phi(\mathbf{x}) \rangle$. Clearly, our Planar index has loglinear indexing time and linear space complexity in the number of data points. It is also worthwhile to mention that as the queries change over time, we update the parameter domain Δa_i, which results in deletion of old indices as well as inclusion of new indices.

4.3 Query Processing

Let us denote by $\mathrm{I}(q, i)$ the i-th co-ordinate of the intersection point between the query hyperplane $H(q)$ and the axis Y_i. Similarly, assume $\mathrm{I}(\mathbf{x}, i)$ denotes the i-th co-ordinate of the intersection point between the index hyperplane $H(\mathbf{x})$ and the axis Y_i. We have, $\mathrm{I}(q, i) = \frac{b}{a_i}$ and $\mathrm{I}(\mathbf{x}, i) = \frac{\langle \mathbf{c}, \phi(\mathbf{x}) \rangle}{c_i}$. 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* $\mathbf{x}$ *for which the index hyperplane* $H(\mathbf{x})$ *intersects the axes at points closer to the origin as compared to the intersection points between the query hyperplane* $H(q)$ *and the corresponding axes. Formally,*

Algorithm 1 Online Algorithm for the Inequality Query

Require: sorted list $\mathcal{L}$ of $\mathbf{x}$ in asc. order of $\langle \mathbf{c}, \phi(\mathbf{x}) \rangle$,
query $\langle \mathbf{a}, \phi(\mathbf{x}) \rangle \leq b$.
Ensure: all $\mathbf{x}$ satisfying the query.
1: find intermediate (II) and smaller (SI) intervals. [Binary Search on $\mathcal{L}$]
2: **for all** $j \in$ SI **do**
3: $\mathbf{x} \rightarrow \mathcal{L}(j)$.
4: output $\mathbf{x}$.
5: **end for**
6: **for all** $j \in$ II **do**
7: **if** $\mathbf{x} \rightarrow \mathcal{L}(j)$ satisfies the query **then**
8: output $\mathbf{x}$.
9: **end if**
10: **end for**

$$\mathrm{SI} = \{\mathbf{x} : (\forall i)\big(|\mathrm{I}(\mathbf{x}, i)| \leq |\mathrm{I}(q, i)|\big)\} \tag{4}$$

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

$$\mathrm{LI} = \{\mathbf{x} : (\forall i)\big(|\mathrm{I}(\mathbf{x}, i)| > |\mathrm{I}(q, i)|\big)\} \tag{5}$$

DEFINITION 3 (INTERMEDIATE INTERVAL). *The* intermediate interval, *referred to as* II, *consists of those data points* $\mathbf{x}$ *which belong to neither the smaller interval nor the larger interval.*

$$\mathrm{II} = \{\mathbf{x} : (\exists i, i')\big(|\mathrm{I}(\mathbf{x}, i)| \leq |\mathrm{I}(q, i)|, |\mathrm{I}(\mathbf{x}, i')| > |\mathrm{I}(q, i')|\big)\} \tag{6}$$

EXAMPLE 3. *Figure 2 illustrates an example of three non-overlapping intervals in* $\mathbb{R}^2$. *In Figure 2,* SI=$\{\mathbf{x_1}, \mathbf{x_2}\}$, II=$\{\mathbf{x_3}, \mathbf{x_4}\}$, *and* LI=$\{\mathbf{x_5}, \mathbf{x_6}, \mathbf{x_7}\}$.

One may notice that we consider an absolute value of the co-ordinates of our intersection points. This is to make our query-processing algorithm applicable for queries outside the first hyper octant as well. However, for the sake of simplicity, we defer the discussion of processing queries outside the first hyper octant until Section 4.5.

Two interesting observations arise from the definitions of our intervals, which are given below.

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.*

PROOF. Omitted due to lack of space. □

Observations 1 and 2 create the basis for accepting and rejecting several 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. Nevertheless, there are two important questions at this stage: **(1)** how can we quickly identify the smaller and larger intervals, and subsequently report all data points that satisfy the given inequality query, and **(2)** can we reduce the cardinality of the intermediate interval? For the first question, we propose a binary-search-based efficient query-processing algorithm (Algorithm 1), which is discussed below. For our second question, we propose the usage of multiple Planar indices — the details of which are given in Section 5.

Particularly, given an inequality query, we first identify the intersection co-ordinates $\mathrm{I}(q,i)$ between the query hyperplane $H(q)$ and the corresponding axes. Recall that we have already sorted the data points $\mathbf{x}$ in a list $\mathcal{L}$ in ascending order of their $\langle \mathbf{c}, \phi(\mathbf{x}) \rangle$ values. Now, for each axis Y_i, we perform a binary search on list $\mathcal{L}$ and find two locations in $\mathcal{L}$ — denoted as $\mathsf{Small}(i)$ and $\mathsf{Large}(i)$, respectively, and formally defined in Equations 7.

$$\begin{aligned} \mathsf{Small}(i) &= \max\{j : \mathcal{L}(j) = \mathbf{x}, \mathrm{I}(\mathbf{x},i) \leq \mathrm{I}(q,i)\} \\ \mathsf{Large}(i) &= \min\{j : \mathcal{L}(j) = \mathbf{x}, \mathrm{I}(\mathbf{x},i) > \mathrm{I}(q,i)\} \end{aligned} \tag{7}$$

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

$$j_{min} = \min_{i \in (1,d')} \{\mathsf{Small}(i)\}; \quad j_{max} = \max_{i \in (1,d')} \{\mathsf{Large}(i)\}$$

$$\begin{aligned} &\mathrm{SI} \rightarrow \mathcal{L}[1 : j_{min}] \\ &\mathrm{II} \rightarrow \mathcal{L}[j_{min}+1 : j_{max}-1] \\ &\mathrm{LI} \rightarrow \mathcal{L}[j_{max} : n] \end{aligned} \tag{8}$$

Computing the interval boundaries requires another $\mathcal{O}(d')$ time. Finally, we report two sets of data points in the answer set: **(1)** for all data points in the intermediate interval, we evaluate the scalar product, and then report those data points which satisfy the given scalar product inequality, as well as **(2)** we also report all data points in the smaller interval. Therefore, the time complexity of our online query-processing algorithm is $\mathcal{O}(d'(\log n + |\mathrm{II}|) + t)$, where $|\mathrm{II}|$ is the cardinality of the intermediate interval, and t is the cardinality of our answer set. Here, we emphasize that the size of the intermediate interval can be zero for carefully designed Planar index. Therefore, our best case query-processing time complexity is logarithmic in the number of data points.

4.4 Dynamic Updates of Planar Index

Our Planar index is lightweight, and hence, easily maintainable and dynamically updatable. Let us consider a collection of n data points. Given an update in $\phi(\mathbf{x})$ associated with some data point $\mathbf{x}$, we can reflect such update in our index structure in $\mathcal{O}(d' \log n)$ time. Alternatively, when we dynamically introduce a new Planar index, it requires $\mathcal{O}(nd' \log n)$ time. Also, the storage complexity of our index structure is $\mathcal{O}(n)$. These time and space complexity results attest that Planar index is efficient, dynamic, and scalable.

4.5 Queries outside First Hyper Octant

In order to complete the overview of the Planar index, we shall discuss how to answer scalar product queries outside the first hyper octant. Fortunately, our Planar index is very general, and it can support both data points and queries outside the first hyper octant.

Without loss of generality, let us assume that the inequality parameter b is always non-negative in the query, while the query parameters $(a_1, a_2, \ldots, a_{d'})$ can be both positive and negative. Since, for each a_i, we have an apriori knowledge about its domain Δa_i, it is possible to identify the hyper octant in which a query hyperplane will intersect the co-ordinate axes. Let us denote this hyper-octant as O, and the sign of an axis Y_i in this hyper octant O is denoted as $\mathsf{sign}(O,i)$. Clearly, $\mathsf{sign}(O,i) \in \{+1, -1\}$. For example, the sign of any axis in the first hyper octant is +1.

In order to build the Planar index, we first perform a *translation* operation such that all $\phi(\mathbf{x})$ for data points $\mathbf{x}$ are in the hyper octant O in the modified co-ordinate system. We claim that after such a translation, the query hyperplane still intersects the modified axes in the same hyper octant O.

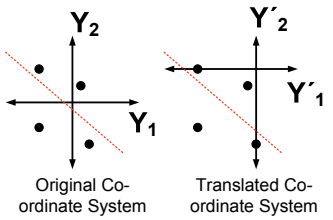


Figure 3: Examples of translation: the query hyperplane intersects the co-ordinate axes in the same hyper octant after translation.

CLAIM 1. *Consider a query hyperplane $H(q)$ that intersects the co-ordinate axes in the hyper octant O. Perform a translation such that all $\phi(\mathbf{x})$ lie in the hyper octant O in the modified co-ordinate system. After this translation, $H(q)$ will intersect the modified axes in the same hyper octant O.*

PROOF. First, we shall define the translation operation which places all $\phi(\mathbf{x})$ in the hyper octant O. For each $i \in (1, d')$, find the set X_i of data points $\mathbf{x}$ for which $\phi_i(\mathbf{x})$ has a sign opposite to $\mathsf{sign}(O,i)$. Formally,

$$X_i = \{\mathbf{x} : \mathsf{sign}(\phi_i(\mathbf{x})) \neq \mathsf{sign}(O,i)\} \tag{9}$$

From the set X_i, we find the translation parameter δ_i, which is defined as the largest absolute value of $\phi_i(\mathbf{x})$ for any $\mathbf{x} \in X_i$.

$$\delta_i = \max_{\mathbf{x} \in X_i} |\phi_i(\mathbf{x})| \tag{10}$$

Finally, we define our translation operation for all $i \in (1, d')$ as given in Equation 11, where $\phi'_i(\mathbf{x})$ denotes the i-th co-ordinate of $\phi(\mathbf{x})$ in the new co-ordinate system.

$$\phi'_i(\mathbf{x}) = \phi_i(\mathbf{x}) + \mathsf{sign}(O,i)\delta_i \tag{11}$$

It is worthwhile to mention that the aforementioned translation will place all $\phi(\mathbf{x})$ in the hyper octant O in the new co-ordinate system. Now, we shall analyze the effect of this translation operation on the query hyperplane. By using the principles of co-ordinate geometry, the query hyperplane in the new co-ordinate system can be represented as follows:

$$\begin{aligned} &H(q) : a_1 Y'_1 + a_2 Y'_2 + \ldots + a_{d'} Y'_{d'} = b', \\ &\text{where } b' = b + \sum_{i=1}^{d'} [\mathsf{sign}(O,i) a_i \delta_i] \end{aligned} \tag{12}$$

We note that $\mathsf{sign}(O,i)a_i$ is positive, and δ_i is non-negative as well, for all $i \in (1, d')$. Hence, b' is a positive term in Equation 12. Therefore, the query hyperplane $H(q)$ still intersects the co-ordinate axes in the same hyper octant O in the new co-ordinate system. This completes the proof. □

Figure 3 provides an illustration of our translation mechanism in $\mathbb{R}^2$. Once we perform the translation, the query processing can follow our proposed technique as outlined in Algorithm 1. Nevertheless, for the sake of simplicity, hereinafter we shall consider query processing only in the first hyper octant. In our experiments, we applied the aforementioned translation technique to deal with queries and data points outside the first hyper octant.

5. QUERY PROCESSING WITH MULTIPLE PLANAR INDICES

In this section, we introduce multiple Planar indices in order to reduce the size of the intermediate interval during online query processing. For a pre-defined budget b, we index function $\phi(\mathbf{x})$

for each data point $\mathbf{x}$ with b planar indices. Thus, given a scalar product query, our objective is to use the best Planar index to answer our query. We here emphasize that the query-processing time reduces with smaller cardinality of the intermediate interval. Indeed, when our Planar index is parallel to the query hyperplane, the cardinality of the intermediate interval is zero, and the query processing requires only logarithmic time in the number of data points. Since the exact query parameters are not known apriori, by introducing multiple Planar indices, it is more likely that one can find an index hyperplane which is "close" to being parallel to the query hyperplane. Below, we first introduce our techniques to find the best Planar index at the time of querying (Section 5.1), and then we discuss our method to select multiple Planar indices during pre-processing (Section 5.2).

5.1 Best Index Selection at Query Time

One naïve approach to find the best Planar index will be as follows. Given a query, count the number of points in the intermediate interval for each Planar index, and then select the index which generates an intermediate interval with the minimum cardinality. However, such a naïve approach, in the asymptotic sense, has time complexity equal to the largest cardinality of any intermediate interval. This creates a well-known "chicken and egg" problem. In other words, *given a query, is it possible to find the best planar index without actually counting the number of points in the intermediate interval for each index?* This is a difficult problem unless one has apriori information about the distribution of data points. To this aim, we propose two greedy heuristics for finding the best Planar index: **(1)** volume minimization of the intermediate interval, and **(2)** angle minimization with the query hyperplane.

5.1.1 Volume Minimization of 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 $H(q)$ and the i-th co-ordinate axis Y_i. We recall that the i-th co-ordinate of the intersection point $\mathbf{q_i}$ is denoted as $\mathrm{I}(q,i)$. Next, for a Planar index with normal vector $\mathbf{c} = (c_1, c_2, \ldots, c_{d'})$, 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 $\mathbf{c}$. The equation of such a hyperplane $H(\mathbf{q_i})$ is given in Equation 13.

$$H(\mathbf{q_i}) : c_1Y_1 + c_2Y_2 + \ldots + c_{d'}Y_{d'} = c_i\mathrm{I}(q,i) \tag{13}$$

There will be d' such hyperplanes for total d' intersection points. We find two hyperplanes H_{max} and H_{min} among them which has maximum and minimum values of $c_i\mathrm{I}(q,i)$, respectively.

$$H_{max} = H(\mathbf{q_{i_1}}) : i_1 = \arg\max_{i\in(1,d')} c_i\mathrm{I}(q,i)$$
$$H_{min} = H(\mathbf{q_{i_2}}) : i_2 = \arg\min_{i\in(1,d')} c_i\mathrm{I}(q,i) \tag{14}$$

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 4 shows the volume of intermediate interval in three dimension for a given query and a planar index. It is easy

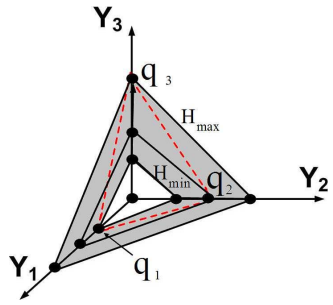


Figure 4: Volume of intermediate interval in 3D: the query hyperplane is marked by dotted lines, and the volume of the intermediate interval is shown as the shaded region.

to verify that any point which lies on the hyper surface bounded by H_{max}, H_{min}, and the co-ordinate axes — except the points on the boundary of H_{min} — 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 2. *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 (except the points on the boundary of H_{min}) 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. Omitted due to lack of space. □

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 our problem statement of selecting the best Planar index in Problem 3.

PROBLEM 3. *Consider a set of r Planar indices. Given a scalar product query q, the stretch of the intermediate interval due to some planar index (with normal vector $\mathbf{c}$) along the axis Y_i is computed as follows.*

$$\mathsf{Stretch}(\mathbf{c},i) = \frac{1}{c_i}\Big[\max_{k\in\{1,d'\}} c_k\mathrm{I}(q,k) - \min_{k\in\{1,d'\}} c_k\mathrm{I}(q,k)\Big] \tag{15}$$

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

$$\arg\min_{\mathbf{c}} \max_{i\in\{1,d'\}} \mathit{Stretch}(\mathbf{c},i) \tag{16}$$

EXAMPLE 4. *Consider a query hyperplane $H(q) : Y_1 + 2Y_2 + 5Y_3 = 10$, and a Planar index with normal vector $(1,1,2)$. The query hyperplane intersects the axes Y_1, Y_2, and Y_3 at points $\mathbf{q_1} = (10,0,0)$, $\mathbf{q_2} = (0,5,0)$, and $\mathbf{q_3} = (0,0,2)$, respectively. Thus, we get $\mathrm{I}(q,1) = 10$, $\mathrm{I}(q,2) = 5$, and $\mathrm{I}(q,3) = 2$. Now, let us consider the hyperplanes passing through the intersection points $\mathbf{q_1}$, $\mathbf{q_2}$, and $\mathbf{q_3}$, respectively, and with the same normal vector $(1,1,2)$ as the Planar index.*

$$H(\mathbf{q_1}) : Y_1 + Y_2 + 2Y_3 = 10$$
$$H(\mathbf{q_2}) : Y_1 + Y_2 + 2Y_3 = 5$$
$$H(\mathbf{q_3}) : Y_1 + Y_2 + 2Y_3 = 4$$

$H(\mathbf{q_1})$, $H(\mathbf{q_2})$, and $H(\mathbf{q_3})$ intersect the axes at points $(10,0,0)$, $(0,10,0)$,$(0,0,5)$; $(5,0,0)$, $(0,5,0)$,$(0,0,2.5)$; and $(4,0,0)$, $(0,4,0)$,$(0,0,2)$, respectively. Therefore, the stretch of the intermediate interval due to this Planar index along axes Y_1, Y_2, and Y_3 are $(10-4)$, $(10-4)$, and $(5-2)$, respectively. Thus, the maximum stretch due to this Planar index along any axis is 6.

Corollary 1 shows that if there exist some Planar index which is parallel to the query hyperplane, our greedy method as proposed in Problem 3 is capable to select the best index.

COROLLARY 1. *If some Planar index is parallel to the query hyperplane, both the volume and the maximum stretch of the intermediate interval is zero.*

Finally, in terms of time complexity, our method finds the best (heuristically) Planar index independent of the data set cardinality. More precisely, given a set of r Planar indices and a scalar product query in $\mathbb{R}^{d'}$, we can find the best Planar index according to problem 3 in $\mathcal{O}(rd')$ time.

5.1.2 Angle Minimization with Query Hyperplane

Our second heuristic method to select the best planar index is quite straightforward, and it works by minimizing the angle between the query hyperplane and some Planar index. Let us consider a scalar product query $q : \langle \mathbf{a}, \phi(\mathbf{x}) \rangle \geq b$ and a Planar index with normal vector $\mathbf{c}$. The angle between the query hyperplane and the Planar index is given by $\cos^{-1}\left(\frac{\langle \mathbf{a}, \mathbf{c} \rangle}{|\mathbf{a}||\mathbf{c}|}\right)$. We greedily select the Planar index as the best index which minimizes the angle with the given query hyperplane.

It is worthwhile to mention that if some Planar index is parallel to the query hyperplane, it makes an angle of zero degree with the query hyperplane. Thus, analogous to our volume minimization-based heuristic, the angle minimization-based technique also selects the best Planar index, when there exists some Planar index which is parallel to the query hyperplane.

Given a collection of r Planar indices and a scalar product inequality query in $\mathbb{R}^{d'}$, the angle minimization technique finds the best (heuristically) Planar index in $\mathcal{O}(rd')$ time.

In our empirical analysis, we found that the minimum-volume-based best index selection method usually outperforms the minimum-angle-based best index selection criterion.

5.2 Multiple Planar Index Selection at Preprocessing Time

For a pre-defined budget b, how do we select the initial b Planar indices? We recall that each query parameter a_i is selected from some domain Δa_i. We pick our indices uniformly from the same domains, and later we remove the *redundant* indices — a Planar index is redundant if there exists another Planar index with normal vectors parallel to each other. We note that b planar indices incur space complexity $\mathcal{O}(nb)$, where n is the number of data points.

6. TOP-K NEAREST NEIGHBOR QUERIES

In this section, we shall discuss our solution technique for the top-k nearest neighbor query (Problem 2). Specifically, given a scalar product query $q : \langle \mathbf{a}, \phi(\mathbf{x}) \rangle \leq b$ and an integer k, we are interested in the top-k points which satisfy the scalar product inequality and also minimize $|\langle \mathbf{a}, \phi(\mathbf{x}) \rangle - b|/|\mathbf{a}|$. The top-$k$ nearest neighbor queries are useful in pool-based active learning [26],

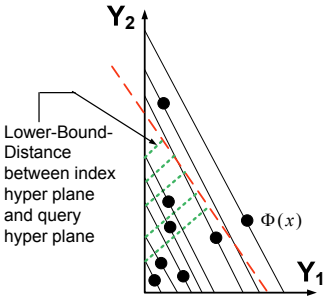


Figure 5: Lower-bound-distance: finding the top-k closest negative points to query hyperplane in 2D.

where given a query hyperplane, one would like to identify the top-k closest positive or negative data points from the query hyperplane [14, 18]. We note that the distance of a point $\mathbf{x}$ from a hyperplane $\langle \mathbf{a}, \mathbf{x} \rangle = b$ is given by $|\langle \mathbf{a}, \mathbf{x} \rangle - b|/|\mathbf{a}|$. Therefore, our top-$k$ nearest neighbor query reduces to the hyperplane-to-nearest-point query when the function ϕ is an identity function.

To answer the top-k nearest neighbor queries efficiently, we use the Planar indexing scheme as introduced earlier. Our online query processing involves a novel pruning-based technique, coupled with a top-k buffer that stores the top-k nearest neighbor points found so far. Particularly, we first consider all data points in the intermediate interval, and if any of them, say $\mathbf{x}$, satisfies the inequality constraint, it is inserted in the top-k buffer based on $\phi(\mathbf{x})$'s distance from the query hyperplane. Next, we consider the data points $\mathbf{x}$ from the smaller interval in descending order of their $\langle c, \phi(\mathbf{x}) \rangle$ values. We terminate our algorithm when the two following conditions are satisfied. **(1)** The top-k buffer is full, and **(2)** for an index hyperplane $H(\mathbf{x})$ corresponding to some data point $\mathbf{x}$ in the smaller interval, if the "lower-bound distance" of $H(\mathbf{x})$ to the query hyperplane $H(q)$ — denoted as $\mathsf{LBS}\big(H(\mathbf{x}), H(q)\big)$, which is illustrated in Figure 5 and formally defined in Equation 17 — is greater than the maximum distance stored in the top-k buffer. An outline of our top-k nearest-neighbor-finding method is given in Algorithm 2.

DEFINITION 5 (LOWER-BOUND DISTANCE). *Consider a query $q : \langle \mathbf{a}, \phi(\mathbf{x}) \rangle \leq b$ and a Planar index with normal vector given by $\mathbf{c} = (c_1, c_2, \ldots c_{d'})$. For some data point $\mathbf{x}$ in the smaller interval, we define the "lower-bound distance" of the index hyperplane $H(\mathbf{x})$ to the query hyperplane $H(q)$ as the smallest value of $\frac{|\frac{a_i}{c_i}\langle \mathbf{c}, \phi(\mathbf{x}) \rangle - b|}{|\mathbf{a}|}$ for all $i \in (1, d')$. Formally,*

$$\mathsf{LBS}\big(H(\mathbf{x}), H(q)\big) = \min_{i \in (1,d')} \frac{\left|\frac{a_i}{c_i}\langle \mathbf{c}, \phi(\mathbf{x}) \rangle - b\right|}{|\mathbf{a}|} \tag{17}$$

Below, we state Claim 3 that forms the basis of our pruning technique employed in Algorithm 2 (see lines 10-11).

CLAIM 3. *Consider two data points $\mathbf{x_1}$, $\mathbf{x_2}$ in the smaller interval such that $\langle c, \phi(\mathbf{x_1}) \rangle$ is larger than $\langle c, \phi(\mathbf{x_2}) \rangle$. Then, it holds that $\mathsf{LBS}\big(H(\mathbf{x_1}), H(q)\big)$ is smaller than $\frac{|\langle \mathbf{a}, \phi(\mathbf{x_2}) \rangle - b|}{|\mathbf{a}|}$.*

PROOF. Omitted due to lack of space. □

Claim 3 provides the theoretical justification of our pruning criteria, which says that if, for some data point $\mathbf{x_1}$ in the smaller interval, $\mathsf{LBS}\big(H(\mathbf{x_1}, H(q))\big)$ is greater than the largest distance stored in our top-k buffer, then we can safely prune all other data points $\mathbf{x_2}$ from the smaller interval that satisfy $\langle c, \phi(\mathbf{x_2}) \rangle < \langle c, \phi(\mathbf{x_1}) \rangle$.

Finally, we analyze the complexity of our top-k nearest neighbor algorithm. In order to find the boundaries of intermediate and larger

Algorithm 2 Online Algorithm for Top-k Nearest Neighbor Query

Require: sorted list $\mathcal{L}$ of $\mathbf{x}$ in asc. order of $\langle \mathbf{c}, \phi(\mathbf{x}) \rangle$, query $\langle \mathbf{a}, \phi(\mathbf{x}) \rangle \leq b$, and an integer k.
Ensure: Top-k nearest neighbor points which satisfy the query.
1: Top-k buffer $\mathcal{B} \rightarrow \phi$
2: find intermediate (II) and smaller(SI) intervals. [Binary Search on $\mathcal{L}$]
/* Process Intermediate Interval */
3: **for all** $j \in$ II **do**
4: **if** $\mathbf{x} \rightarrow \mathcal{L}(j)$ satisfies the query **then**
5: insert $\mathbf{x}$ into $\mathcal{B}$.
6: **end if**
7: **end for**/* Process Smaller Interval */
8: **for all** $j \in$ SI in dsc. order **do**
9: $\mathbf{x} \rightarrow \mathcal{L}(j)$
10: **if** $\mathcal{B}$ full **and** LBS$\big(H(\mathbf{x}), H(q)\big) >$ largest dist. in $\mathcal{B}$ **then**
11: terminate.
12: **end if**
13: insert $\mathbf{x}$ into $\mathcal{B}$.
14: **end for**
15: report data points in $\mathcal{B}$.

intervals, we require $\mathcal{O}(d' \log n)$ time. Since we need to verify all the data points in the intermediate interval before inserting them in the top-k buffer, it requires another $\mathcal{O}(d'|\text{II}|)$ time, where $|\text{II}|$ is the cardinality of the intermediate interval. Next, assume that total k_1 data points from the smaller interval are verified before we can terminate our algorithm. Then, the overall time complexity of our nearest-neighbor-finding algorithm is $\mathcal{O}\big(d' \log n + (|\text{II}| + k_1)(d' + \log k)\big)$. In the best case, that is, when the Planar index is parallel to the query hyperplane, $|\text{II}|$=0, and $k_1 = k + 1$, where k is the top-k value given as the input. Thus, our best case time complexity is $\mathcal{O}\big(d' \log n + d'k + k \log k\big)$.

7. EXPERIMENTAL RESULTS

We present experiments to assess the performance of our Planar index for answering scalar product queries. We evaluate: query-processing efficiency (Section 7.2), index time, memory, and dynamic updates (Section 7.3), and scalability (Section 7.4). Furthermore, we analyze the performance of Planar index in two real-world applications: **(1)** finding intersection between moving objects with both uniform and non-uniform workloads (Section 7.5.1) and **(2)** reporting the top-k nearest points to a query hyperplane (Section 7.5.2), which is critical in pool-based active learning [26]. The code is implemented in C++ and the experiments were performed on a single core of a 100GB, 2.50GHz Xeon server.

7.1 Environmental Setup

Datasets. We involve three synthetic and three real-world datasets, each containing a collection of multi-dimensional data points.

Synthetic. We generate three synthetic datasets by using the generator obtained from [4]. 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).

Image. The real-world *Image* database contains image features extracted from a Corel image collection (`http://corel.digitalriver.com`) with 68,040 photos. We consider two sets of features for our experiments: color moments and co-occurrence texture, and the corresponding datasets are *CMoment* and *CTexture*, respectively. The first dataset is 9 dimensional with attribute values between (-4.15, 4.59), whereas the second dataset is 16 dimensional and its attribute values are in (-5.25, 50.21). Both the datasets are publicly available from `http://archive.ics.uci.edu/ml/datasets.html`.

Table 2: *Dataset characteristics.*

Dataset	# Data Points	# Dimension	Attribute Range
Indp	1,000,000	2 - 14	(1, 100)
Corr	1,000,000	2 - 14	(1, 100)
Anti	1,000,000	2 - 14	(1, 100)
CMoment	68,040	9	(-4.15, 4.59)
CTexture	68,040	16	(-5.25, 50.21)
Consumption	2,075,259	4	(0, 254)

Electric Power Consumption. The *Consumption* dataset consists of electric power consumption measurements for 2,075,259 individual households. Each data point has 4 dimensions: active power (range: 0-11 KWatt), reactive power (range: 0-1 KWatt), voltage (range: 223-254 Volt), and current (range: 0-48 Ampere). This dataset is downloadable from `http://archive.ics.uci.edu/ml/datasets.html`.

Query selection and parameter setting. For the real-world *Consumption* dataset, we consider a complex non-linear SQL query: *find all households for which the* power factor *is less than an input threshold.* Note that power factor is defined as the ratio between active power and (voltage $\times$ current). For details, see Example 1. We select the query parameter "threshold" uniformly from the range (0.100,1.000); i.e., we allow 900 possible query normal vectors.

For the real-world image datasets and for all synthetic datasets, we consider a more generalized form of the scalar-product query:

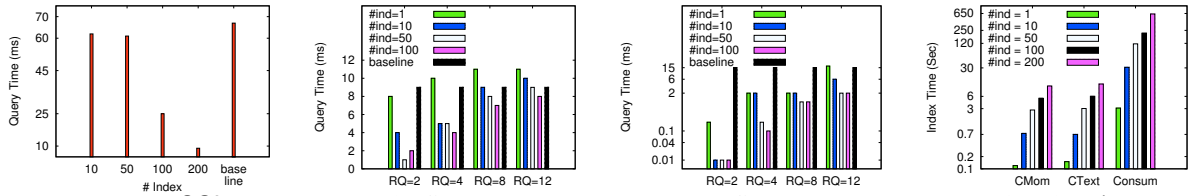
$$\sum_{i=1}^{d} a_i x_i \leq 0.25\big(\sum_{i=1}^{d} a_i \max(i)\big) \tag{18}$$

Here, we assume that our data points $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ are d-dimensional, and $\max(i)$ denotes the maximum value of the i-th dimension in the data set. We multiply the right hand side of our query by an *inequality parameter* 0.25 — this results in a small fraction of data points satisfying our queries. We further vary this inequality parameter in Figure 11, and thereby analyze our query-processing performance with different *query selectivity*. It is worthwhile to mention that if too many data points satisfy the query, the time complexity in the asymptotic sense gets close to $\Theta(n)$, since we need to report all the data points in the result set. Hence, we design our queries in a way such that a small percentage of the data points satisfy these queries.

In Equation 18, 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. Since we do not know the exact query parameters, our objective is to employ only a few number of Planar indices for quickly answering any such query from the potential query set. We vary the randomness of query (RQ) from 2 to 12, while the number of Planar indices is varied from 1 to 200.

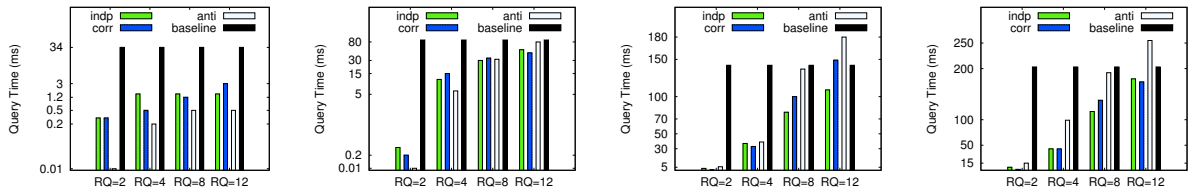
All experimental results are averaged over 100 runs. In all our experiments, we found that the minimum-volume-based best index selection method (Section 5) results in improved query efficiency as compared to its counter part: minimum-angle-based best index selection criterion. Thus, we employed the minimum-volume-based best index selection method in all our experiments.

Competing Method. We compare the performance of our Planar index with a baseline method that performs a naïve sequential scan over the entire dataset.



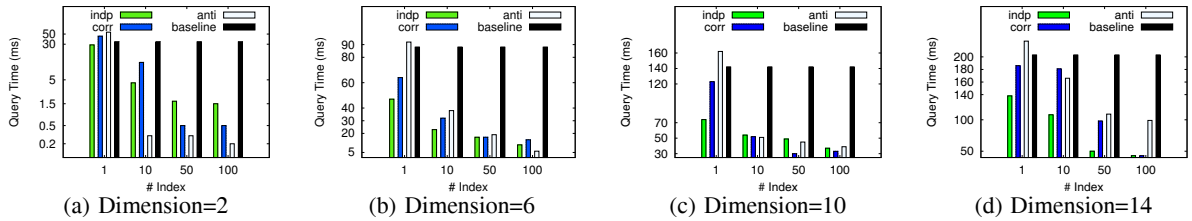
(a) Consumption (SQL func.) (b) CMoment (c) CTexture (d) Index Time

Figure 6: Index and query-processing times using real-world datasets (*Consumption*, *CMoment*, and *CTexture*)



(a) Dimension=2 (b) Dimension=6 (c) Dimension=10 (d) Dimension=14

Figure 7: Query-processing time using synthetic datasets (*Indp*, *Corr*, and *Anti*): # dimensions = 2 $\sim$ 14, and randomness of query (RQ) varied from 2 $\sim$ 12, # index = 100. Baseline running times are for any of the three synthetic datasets.



(a) Dimension=2 (b) Dimension=6 (c) Dimension=10 (d) Dimension=14

Figure 8: Query-processing time using synthetic datasets (*Indp*, *Corr*, and *Anti*): # dimensions = 2 $\sim$ 14 and # index = 1 $\sim$ 100, randomness of query (RQ) = 4. Baseline running times are for any of the three synthetic datasets.

7.2 Query-Processing Efficiency

7.2.1 *Efficiency on Real-World Dataset*

We show the performance of our query-processing technique on real-world datasets in Figure 6. Note that we execute the generalized scalar product query as given in Equation 18 over *CMoment* and *CTexture* datasets, while we evaluate the performance of a non-linear SQL function using the *Consumption* data.

For the evaluation of the SQL function with the *Consumption* dataset, the baseline method requires 62 ms, while our technique with 200 Planar indices takes only 9 ms — thereby, improving the query-processing efficiency by 7 times (see Figure 6(a)).

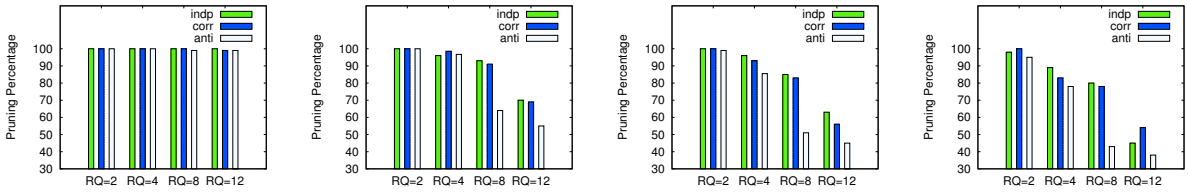
Figure 6(b) shows that with 100 indices, RQ=4, and using the *CMoment* dataset, our query-processing time is 2 times faster than the baseline method — the baseline requires 9 ms, while our query-processing finishes in 4 ms. With the same set of parameters and using the *CTexture* dataset, our method is about 150 times faster: the baseline needs 15 ms, while our query-processing takes only 0.1 ms (Figure 6(c)).

7.2.2 *Efficiency on Synthetic Datasets*

We present in Figures 7 and 8 the query-processing times using three synthetic datasets: *indp*, *corr*, and *anti*, by varying the number of dimensions, number of indexes, and randomness of query (RQ). We observed that with 100 Planar indices and dimensionality up to 6, our query-processing times are 4 orders of magnitude faster than the baseline when RQ=2: 0.01 ms for our method vs. 88 ms using baseline; and it is 14 times faster when RQ=4: 6 ms for our method vs. 88 ms using baseline (see Figure 7(b)). However, as the dimensionality increases, our query-processing time also increases. Nevertheless, with lower query randomness (up to RQ=4), our Planar index-based query-processing times are at least 2 to 3 times faster as compared to the baseline method: 99 ms for our method vs. 203 ms using baseline in Figures 8(c). When both the dimensionality and randomness of query are higher, our query-processing time gets closer to the baseline running time (see Figure 7(d)). This is due to the fact that as RQ increases, the number of possible query normals increases exponentially, and it is difficult to obtain a similar improvement in query times by linearly increasing the number of Planar indices. Rather, it is more beneficial to dynamically update our indices based on the recent queries, as we shall demonstrate in Section 7.3 that our index construction times are very affordable.

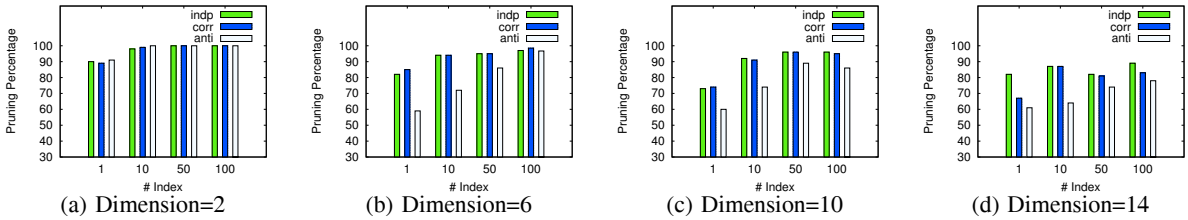
We recall that the key idea of Planar index is to allow very fast pruning of the data points without actually computing the scalar product for them. In Figures 9 and 10, we show the pruning percentage, that is, the percentage of data points that can be accepted or rejected without actually computing the scalar product for them. With 100 indices, dimensions up to 6 and RQ up to 4, we found that almost 90$\sim$100% of the data points can be pruned directly (Figure 10(b)). These results attest high quality of our Planar indexing scheme. It is worthwhile to mention that with high dimensionality and high query randomness (e.g., dimensionality=14 and RQ=12), 100 Planar indices still achieve about 40$\sim$50% pruning of data points (Figure 9(d)), but the corresponding query-processing times get very close to the baseline time (Figure 7(d)). This is because *in our query-processing method, the points in the intermediate interval require a random access — which takes more time; as compared to the fact that all data points are accessed sequentially in the baseline method.*

We note that in higher dimensions ($\geq$ 6), our query-processing requires more time for the *anticorrelated* data, as compared to the other two synthetic datasets. This is because in the *anti-correlated*



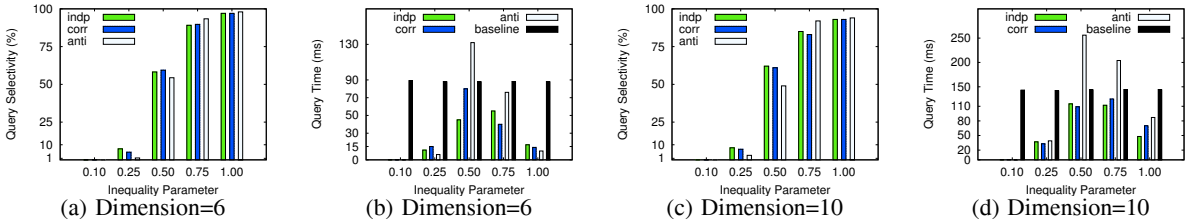
(a) Dimension=2 (b) Dimension=6 (c) Dimension=10 (d) Dimension=14

Figure 9: Pruning percentage for synthetic datasets: # dimensions = 2 $\sim$ 14, and randomness of query (RQ) = 2 $\sim$ 12, # index = 100.



(a) Dimension=2 (b) Dimension=6 (c) Dimension=10 (d) Dimension=14

Figure 10: Pruning percentage for synthetic datasets: # dimensions = 2 $\sim$ 14 and # index = 1 $\sim$ 100, randomness of query (RQ) = 4.



(a) Dimension=6 (b) Dimension=6 (c) Dimension=10 (d) Dimension=10

Figure 11: Query selectivity and query-processing time with varying inequality parameter: synthetic datasets, # index = 100, randomness of query (RQ) = 4. Baseline running times are for any of the three synthetic datasets.

dataset, points which have higher values in one dimension have lower value(s) in one or all of the other dimensions — thus, it generates more data points in the intermediate interval.

7.2.3 *Efficiency with Varying Query Selectivity*

We varied the inequality parameter of our query in Equation 18 from 0.10 to 1.00, and thereby analyze the effect of query selectivity over query-processing time using three synthetic datasets. In Figure 11, we consider RQ=4, number of indices=100, and vary the dimension from 6 to 10. We observe that the query selectivity increases as we increase the inequality parameter (Figure 11(a), 11(c)), while the query-processing time first increases and then decreases, which is more prominent in Figure 11(b). This is because if the inequality parameter is too high or too low, a high percentage of data points could be directly accepted or rejected using our query-processing technique. As expected, our query-processing time is maximum when the inequality parameter is between 0.50$\sim$0.75.

7.3 Index Building Time and Memory Usage

Figure 13(a) shows index construction times with varying dimensionality and varying number of Planar indices using three synthetic datasets. Note that the indexing time is independent of the particular type of synthetic data. The time to construct one index for 1M data points varies from only 2.54 sec to 2.92 sec, for dimensionality of the data points from 2 to 14. These smaller indexing times justify that our indices are dynamically updatable with changes in the query, and *it is more beneficial to update our Planar indices over time as opposed to maintain a large number of indices*.

Figure 6(d) affirms modest index-building times (0.12$\sim$3.11 sec) to construct one Planar index) over three real-world datasets.

Memory Usage. We show the memory consumption by our index structure in Figure 13(b) using three synthetic datasets. Our memory consumption is quite modest: even for 100 indices with 1M data points and dimensionality of each point up to 14, our index structure uses less than 5GB memory. In addition, the memory consumption of our Planar indices is almost independent with respect to the dimensionality of the data points, and it increases linearly with the number of data points and also with the number of indices.

Dynamic Updates in Data Points. In these experiments, we analyze our index-structure-modification time based on dynamic updates in the data points. Figure 13(c) shows that if we have already indexed 1M data points of dimensionality 10, and later 5% of the data points, that is 50K data points, change their values, we can dynamically update our index structure in 170 ms per Planar index — which is equivalent to only 3.4 ms per data point per Planar index. Hence, *it is more beneficial to update our indices dynamically if there is a change in small percentage of data points*.

7.4 Scalability

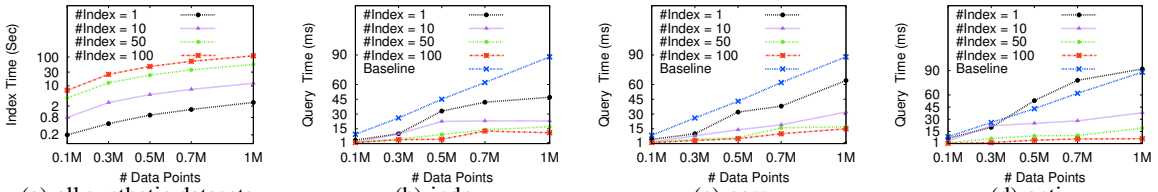
We analyze scalability of our query-processing technique using three synthetic datasets. For this experiment, we consider fragments of the original datasets with number of data points 0.1M, 0.3M, 0.5M, 0.7M, and 1M, respectively. The corresponding index-building and query-processing times are presented in Figure 12.

We observe that the indexing time increases loglinearly with the number of data points, while the query time increases sublinearly. Such results assess high scalability of Planar index construction and the subsequent processing of scalar product queries.

7.5 Applications of Planar Index

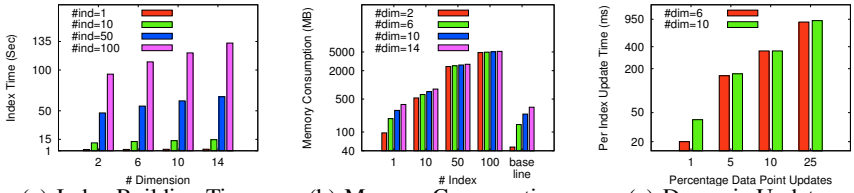
7.5.1 *Moving-Objects Intersection*

We show an application of Planar index in finding intersections between two sets of moving objects. Every object set has cardinality 5K, that is, each query verifies for the intersection of 25M distinct object pairs. We consider three different scenarios with both



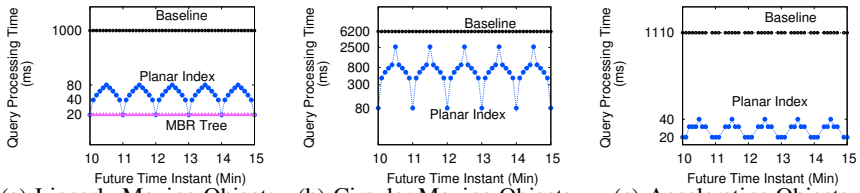
(a) all synthetic datasets (b) indp (c) corr (d) anti

Figure 12: Scalability with varying number of data points using synthetic datasets: # index= 1∼100, randomness of query (RQ) = 4, and # dimensions =6.



(a) Index Building Time (b) Memory Consumption (c) Dynamic Updates

Figure 13: Index construction time, memory usage, and dynamic index updates using synthetic datasets



(a) Linearly Moving Objects (b) Circular Moving Objects (c) Accelerating Objects

Figure 14: Planar index in finding moving object intersection: uniform and non-uniform workloads

uniform (e.g., objects moving with constant linear and angular velocity) and non-uniform (e.g., accelerating objects) workloads.

Objects moving with uniform velocity. We consider a simulating scenario [23, 33] with two sets of objects generated uniformly within a 2D space of 1000 × 1000 mile^2. Every object is moving linearly with a uniform velocity between 0.1∼1 mile/min in both (positive or negative) X and Y-directions. Given a future time-instant t at the time of querying, we want to find all object pairs which will be within 10 miles from each other at time t. The above-mentioned intersection query can be expressed as a scalar product query: $AX_1 + BX_2 + CX_3 \leq 100$, where the functional part consists of $X_1 = (p_x - q_x)^2 + (p_y - q_y)^2$, $X_2 = 2[(p_x - q_x)(u_x - v_x) + (p_y - q_y)(u_y - v_y)]$, $X_3 = (u_x - v_x)^2 + (u_y - v_y)^2$; and the parametric part is given by $A = 1$, $B = t$, and $C = t^2$. Here, p_x, p_y, u_x, and u_y denote the initial X- and Y-coordinates, velocity along X- and Y-directions, respectively, for an object from the first set. We define q_x, q_y, v_x, and v_y analogously.

Since, our index for a future time-instant is good for answering queries over a certain period of time surrounding that time-instant, we apply the "MOVIES" technique [9] — for a short period of time, we use an index to answer the incoming queries. After that, we throw that index away and use a new index. Note that we can apply the "MOVIES" technique due to two reasons. (1) We keep multiple Planar indices corresponding to several time-instants, and select the best index at the time of querying. As an example, we demonstrate below that by keeping indexes for only 6 time-instants, we can efficiently answer queries for a duration of 5 minutes. (2) Our index building requires only 10.8 sec for every time-instant, while an update in one moving object requires only 0.5 ms to update the existing index structure (corresponding to one time-instant). Therefore, it is quite efficient to build a new index as well as dynamically update the existing index structure.

In our simulation, we keep 6 Planar indices at a time, corresponding to the future time-instants: t = 10, 11, 12, 13, 14, and 15 min. We vary the future time-instants in our queries between 10∼15 min, and it can be an intermediate time-instant, e.g., 11.5 min, for which no index exists. We compare our query-processing times with state-of-the-art MBR-tree-based method [33] (which is an improvement over the widely-used TPR-tree [23]), and also with a baseline method which verifies all 5K×5K object pairs. The authors of [33] kindly provided us a C++ implementation.

Figure 14(a) shows that the performance of Planar index is comparable to that of [33] when an index is available for the future time-instant. Otherwise, Planar index-based method is at most 4 times slower than [33]; and therefore, it is beneficial to apply state-of-the-art method [33] for objects moving in straight lines with uniform velocity. However, it is difficult to apply such spatio-temporal indexes for complex and non-uniform motions, such as objects moving in circles or with acceleration. Below, we show two examples when Planar index is quite effective in complex and non-uniform motions, and these results show the generality of Planar index.

Circular moving objects. We consider two sets of objects generated uniformly within a 2D space of 100 × 100 mile^2. One set of objects are moving with a uniform velocity between 0.1∼1 mile/min in both (positive or negative) X and Y-directions. The other set of objects are moving in concentric circles with radius uniformly selected from 1∼100 mile, and angular velocity uniformly selected from 1∼5 degree/min. We use Planar index to find intersecting object pairs at a future time. For details, see example 2. We show the intersection-finding times in Figure 14(b), with the usage of 6 Planar indices corresponding to future time-instants t = 10, 11, 12, 13, 14, and 15 min. Our index building requires only 10.7 sec for every time-instant. We observe that Planar index outperforms the baseline technique by 2.5∼75 times.

Objects moving with acceleration. We simulate a *non-uniform workload* by assuming that objects in one set are moving with acceleration, while the objects in the other set are still moving with constant velocity. The objects are generated uniformly within a 3D space of 1000 × 1000 × 1000 mile^3. Objects from the first

Table 3: *Top-k nearest-neighbor-finding time using* Indp *dataset; # dimensions = 6, RQ=4, # index = 100*

# Top-k	Checked Points/Total Points (%) [Planar index]	Query Time (ms) [Planar index]	Baseline Time (ms)
50	10.97	33	89
1000	11.29	36	89
10000	12.62	42	89

set are moving with an initial velocity between 0.1∼1 mile/min and acceleration between 0.01∼0.05 mile/min^2 in (positive or negative) X, Y, and Z-directions. Objects from the second set are moving with uniform velocity between 0.1∼1 mile/min in (positive or negative) X, Y, and Z-directions. The corresponding intersection query can be formulated as a scalar product query with dimensionality 5: $AX_1 + BX_2 + CX_3 + DX_5 + EX_5 \leq 100$, where $X_1 = (p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2$, $X_2 = 2[(p_x - q_x)(u_x - v_x) + (p_y - q_y)(u_y - v_y) + (p_z - q_z)(u_z - v_z)]$, $X_3 = (u_x - v_x)^2 + (u_y - v_y)^2 + (u_z - v_z)^2 + a_x(p_x - q_x) + a_y(p_y - p_y) + a_z(p_z - p_z)$, $X_4 = a_x(u_x - v_x) + a_y(u_y - v_y) + a_z(u_z - v_z)$, $X_5 = \frac{1}{4}(a_x^2 + a_y^2 + a_z^2)$; and $A = 1$, $B = t$, $C = t^2$, $D = t^3$, $E = t^4$. Here, a_x, a_y, and a_z denote the acceleration of an object from the first set along the X, Y, and Z directions, respectively. The other notations are used as before. We show the intersection-finding times in Figure 14(c), with the usage of 6 Planar indices corresponding to future time-instants $t = 10$, 11, 12, 13, 14, and 15 min. Our index building requires only 11.3 sec for every time-instant. We observe that Planar index outperforms the baseline technique by 25∼50 times. These results show that the Planar indices are very effective even for non-uniform workloads.

7.5.2 *Finding Top-k Nearest Points*

We show another application of Planar index in finding the top-k closest positive or negative points to a query hyperplane, which has application in active learning [26]. We present our results in Table 3 with the *Indp* dataset and query in the form of Equation 18. We observe that 100 Planar indices achieve about 2.5 times speed-up over a sequential scan. It is also worthwhile to mention that our method finds the top-k closest points in an *accurate* manner, as opposed to the *approximate* methods proposed in [14, 18].

8. CONCLUSIONS

In this paper, we studied scalar product queries — a widely-applicable set of analytic queries whose parameters are known only at the time of querying. We defined Planar index, a geometric approach that allows for online processing of scalar product queries in an efficient and accurate manner, as confirmed by an extensive experimental evaluation conducted on various synthetic and real-world datasets. We further show the applications of Planar index in the moving-objects-intersection problem and in active learning.

Future work can be in two directions: since Planar index has high pruning capacity for low-dimensional datasets, it would be interesting to apply various dimensionality reduction techniques as a preprocessing method. One may also use machine learning techniques to dynamically update the indices based on past queries.

9. REFERENCES

[1] P. K. Agarwal, L. Arge, , J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient Searching with Linear Constraints. In *PODS*, 1998.

[2] S. Arya, D. M. Mount, and J. Xia. Tight Lower Bounds for Halfspace Range Searching. *Discrete Comput. Geom.*, 47(4):711–730, 2012.

[3] R. Basri, T. Hassner, and L. Zelnik-Manor. Approximate Nearest Subspace Search. *TPAMI*, 33(2):266–278, 2011.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, 2001.

[5] G. Box and G. Jenkins. *Time Series Analysis: Forecasting and Control*. San Francisco: Holden-Day, 1970.

[6] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The Onion Technique: Indexing for Linear Optimization Queries. In *SIGMOD*, 2000.

[7] S. Chen, C. S. Jensen, and D. Lin. A Benchmark for Evaluating Moving Object Indexes. *PVLDB*, 1(2):1574–1585, 2008.

[8] S. Chen, B. C. Ooi, K.-L. Tan, and M. A. Nascimento. ST^{2}B-Tree: a Self-Tunable Spatio-Temporal B$^+$-Tree Index for Moving Objects. In *SIGMOD*, 2008.

[9] J. Dittrich, L. Blunschi, and M. A. V. Salles. MOVIES: Indexing Moving Objects by Shooting Index Images. *GeoInformatica*, 15(4):727–767, 2011.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[11] J. Goldstein, R. Ramakrishnan, U. Shaft, and J.-B. Yu. Processing Queries by Linear Constraints. In *PODS*, 1997.

[12] V. Hristidis, N. Koudas, Y. Papakonstantinou, Y. Papakonstantinou, and L. J. Ca. PREFER: A System for the Efficient Execution of Multiparametric Ranked Queries. In *SIGMOD*, 2001.

[13] I. F. Ilyas, G. Beskales, and M. A. Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008.

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

[15] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+ Tree based Indexing of Moving Objects. In *VLDB*, 2004.

[16] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. In *PODS*, 1999.

[17] C. Li, K. C.-C. Chang, , I. F. Ilyas, and S. Song. RankSQL: Query Algebra and Optimization for Relational Top-K Queries. In *SIGMOD*, 2005.

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

[19] J. Matousek. Reporting Points in Halfspaces. *Computational Geometry*, 2(3):169 – 186, 1992.

[20] M. A. Nascimento and J. R. O. Silva. Towards Historical R-Trees. In *SAC*, 1998.

[21] Oracle. Oracle Function-based Indexes. In *11g Release*.

[22] P. Ram and A. G. Gray. Maximum Inner-product Search Using Cone Trees. In *KDD*, 2012.

[23] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.

[24] H. Samet. *Applications of Spatial Data Structures - Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.

[25] H. Samet, J. Sankaranarayanan, and M. Auerbach. Indexing Methods for Moving Object Databases: Games and Other Applications. In *SIGMOD*, 2013.

[26] B. Settles. Active Learning Literature Survey. CS Tech. Report, Univ. of Wisconsin–Madison, 2009.

[27] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *ICDE*, 1997.

[28] Y. Tao and D. Papadias. Time-Parameterized Queries in Spatio-Temporal Databases. In *SIGMOD*, 2002.

[29] J. Uhlmann. Satisfying General Proximity/Similarity Queries with Metric Trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[30] D. Xin, , J. Han, and K. C. Chang. Progressive and Selective Merge: Computing Top-k with Ad-Hoc Ranking Functions. In *SIGMOD*, 2007.

[31] P. Y. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *SODA*, 1993.

[32] M. Yiu, Y. Tao, and N. Mamoulis. The B$^{\text{dual}}$-Tree: Indexing Moving Objects by Space Filling Curves in the Dual Space. *VLDB J.*, 17(3):379–400, 2008.

[33] R. Zhang, J. Qi, D. Lin, W. Wang, and R. C.-W. Wong. A Highly Optimized Algorithm for Continuous Intersection Join Queries over Moving Objects. *VLDB J.*, 21(4):561–586, 2012.