Voronoi-based nearest neighbor search for multi-dimensional uncertain databases

Zhang, P; Cheng, R; Mamoulis, N; Renz, M; Züfle, A; Tang, Y; Emrich, T

The 29th International Conference on Data Engineering (ICDE 2013), Brisbane, Australia, 8-11 April 2013. In International Conference on Data Engineering Proceedings, 2013, p. 158-169

2013

http://hdl.handle.net/10722/189614

International Conference on Data Engineering, Proceedings. Copyright © IEEE, Computer Society.; ©2013 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.; This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
Voronoi-based Nearest Neighbor Search for Multi-Dimensional Uncertain Databases

Peiwu Zhang #1, Reynold Cheng #2, Nikos Mamoulis #3, Matthias Renz *4
Andreas Züfle *5, Yu Tang *6, Tobias Emrich *7

#1 The University of Hong Kong, Pokfulam Road, Hong Kong
#2 Ludwig-Maximilians-Universität München, Munich, Germany
#3 The University of Hong Kong, Pokfulam Road, Hong Kong
#4 Ludwig-Maximilians-Universität München, Munich, Germany
#5 Department of Computer Science, Ludwig-Maximilians-Universität München, Munich, Germany
#6 Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong
#7 Ludwig-Maximilians-Universität München, Munich, Germany

Abstract—In Voronoi-based nearest neighbor search, the Voronoi cell of every point \( p \) in a database can be used to check whether \( p \) is the closest to some query point \( q \). We extend the notion of Voronoi cells to support uncertain objects, whose attribute values are inexact. Particularly, we propose the Possible Voronoi cell (or PV-cell). A PV-cell of a multi-dimensional uncertain object \( o \) is a region \( R_\varepsilon \) such that for any point \( p \in R_\varepsilon \), \( o \) may be the nearest neighbor of \( p \). If the PV-cells of all objects in a database \( S \) are known, they can be used to identify objects that have a chance to be the nearest neighbor of \( q \).

However, there is no efficient algorithm for computing an exact PV-cell. We hence study how to derive an axis-parallel hyper-rectangle (called the Uncertain Bounding Rectangle, or UBR) that tightly contains a PV-cell. We further develop the PV-index, a structure that stores UBRs, to evaluate probabilistic nearest neighbor queries over uncertain data. An advantage of the PV-index is that upon updates on \( S \), it can be incrementally updated. Extensive experiments on both synthetic and real datasets are carried out to validate the performance of the PV-index.

I. INTRODUCTION

Nearest neighbor queries are fundamental procedures for many similarity search and location-based query applications for location-based services. In particular, a nearest-neighbor query allows users to retrieve the most similar object to a given query object or to retrieve a location from a geospatial database that is closest to her current location. In recent studies, it is shown that the Voronoi diagram is extremely efficient in exploring a local neighborhood in a geometric space [1].

Given a set of points, a Voronoi diagram uniquely partitions the space into disjoint regions called Voronoi cells, such that each cell is assigned to one single point. The Voronoi cell corresponding to a point \( o \) covers the points in space that are closer to \( o \) than to any other point, as illustrated in Figure 1(a).

Attribute values of a traditional database are often assumed to be exact. This is no longer true for many emerging applications. Consider a system that retrieves positions of pedestrians, vehicles, and buildings from satellite images through human effort (e.g., Wikimapia) and machine learning algorithms. Due to the error-prone nature of the data transmission and extraction procedures, the location values obtained from the images may not be correct. This database, if released to the public, may also be perturbed with noise, in order to alleviate privacy concerns [2]. In natural habitat monitoring, information collected at sensor nodes (e.g., temperature and humidity) can be contaminated with measurement error [3].

In order to satisfy the increasing needs of managing imprecise data, several uncertain databases have been developed [4]–[7].

In this paper, we study the efficient evaluation of the probabilistic nearest neighbor query (PNNQ), a fundamental query operator used in many uncertain databases, by adapting the general concept of Voronoi-based nearest neighbor search. Given a multi-dimensional point \( q \) (e.g., location of a vehicle, a vector of (temperature, humidity, wind speed)), a PNNQ returns the identities of objects whose (qualification) probabilities of being the closest to \( q \) are larger than zero [8].

The problem of evaluating this query in a scalable manner, which is technically challenging, has attracted plenty of research interest (e.g., [8]–[11]). In general, the execution of PNNQ involves the following steps: Step 1: retrieving objects whose qualification probabilities are larger than zero; and Step 2: computing the qualification probabilities of objects obtained in Step 1. Most previous work focused on the efficiency issues of Step 2. Our goal is to propose a scalable solution for enhancing the performance of Step 1.

Fig. 1. Illustrating the PV-cell of (circled) object \( o \), whose uncertainty region is: (a) point; (b) 2D rectangle; and (c) 3D rectangle (where the PV-cell is composed of curved surfaces).

Specifically, we study a solution based on Possible Voronoi cells (or PV-cells in short). To understand this concept, let us consider a database that follows the attribute uncertainty model [8], a model that is commonly used in the database community in the context of uncertain data [9], [10], [12]–[14]. Consider a \( d \)-dimensional domain \( D \), where \( D \subseteq \mathbb{R}^d \). In the model used in this paper the \( d \)-dimensional attribute values of an object is a random variable specified by a given probability distribution [5], [8] called uncertainty pdf. For instance, a location value, obtained by a GPS sensor, can be represented by a Gaussian distribution [15]; the temperature, humidity,
and wind speed values obtained at a sensor node is a three-dimensional attribute with some probability distribution [3]. Here, we adopt the discrete model [13], [14], where o’s uncertainty pdf is represented by a set of d-dimensional points, or “instances”. Each instance is assigned the probability of being the exact representation of o. Since we are interested in retrieving all possible nearest neighbors of a given query object q, i.e. objects with non-zero probability being the nearest neighbor of q, we can use a more simplified approximation of an uncertain object. Following the object representation as proposed in [8], [10], [11], [13], an uncertain object o is defined by an axis-parallel rectangular region u(o) ⊆ D which we call uncertainty region of o. Specifically, u(o) minimally bounds all possible values of o’s attributes o.a. We note that our solution can also be used to handle the case when u(o) is not a rectangle.

Based on the definition of an uncertain object we can now formally define the Possible Voronoi Cell.

Definition 1: Given an uncertain database S, i.e. a set of uncertain objects. The Possible Voronoi Cell (PV-cell) of an uncertain object o ∈ S, denoted by V(o), is a d-dimensional region R, such that for any point p ∈ R, o has a chance, i.e. a non-zero probability, to be the closest to p among all the objects in database S.

Here, “to be the closest to p” means to have the smallest Euclidean distance to p among the objects in S. Notice that when the objects in S are certain points, V(o) reduces to a Voronoi cell of o, as illustrated in the example shown in Figure 1(a).

The example given in Figure 1(b) shows the (2D) uncertainty regions of locations of five objects. Here, the PV-cell of o (in dotted circle) is a region bounded by solid curve segments. To see whether o has a non-zero probability to be the closest to a given query point q, we just check whether q is inside the PV-cell of o. Moreover, if the PV-cells of all objects in S are known, Step 1 of PNNQ evaluation can be performed by retrieving objects in S, whose PV-cells contain q. This approach, as shown by our experiments, is much faster than previous solutions (e.g., [8], [11]). Figure 1(c) shows another example of PV-cells, for uncertainty regions that are 3D rectangles, respectively.

Unfortunately, computing exact PV-cells is rather complex. To the best of our knowledge, there exists no efficient solution for this problem. In fact any exact algorithm must scale exponentially in the number of dimensions. To make this clear, consider the simple case where the database consists of only two uncertain objects o₁ and o₂. The border between the two corresponding Voronoi cells is piece-wise curvilinear. The number of pieces is linearly correlated with the number of corners of an uncertain object, which is in $O(2^d)$ for rectangles. (see [16] for a discussion on the computation of such Voronoi planes).

To tackle this problem, we define the Uncertain Bounding Rectangle, or UBR, which is an axis-parallel rectangle that tightly approximates the PV-cell P in a conservative way, i.e. completely contains P. We observed from our experiments that if we use an UBR of P which is only slightly larger than an UBR that minimally bounds P, the performance of PNNQ will not be significantly affected. The advantage of such a loose fitting UBR is that it can be quickly obtained as it does not require to compute the exact PV-cell. The main idea is to approximate the PV-cell iteratively in a conservative way based on distance relationships between uncertain objects. Specifically, in each iteration we apply the concept of distance domination following the studies made in [17] in order to check whether the current UBR is still a conservative approximation of the PV-cell. Based on this concept, we propose a Shrink-and-Expand (or SE) algorithm. This solution runs in an iterative manner; in each round, the UBR of P is either enlarged or reduced, until its size is similar to that of the MBR of P. This algorithm, which only needs a logarithmic number of steps, can efficiently derive an UBR.

Based on the UBRs computed by SE, we develop the PV-index, a space-partitioning structure that organizes the UBRs in a systematic manner, so that a PNNQ can be efficiently executed. Through a detailed experimental evaluation on real and synthetic datasets, we show that our solution is efficient and scalable.

We further address the issue of updating the PV-index upon insertion (deletion) of an object to (from) S. A straightforward solution is to rebuild the index from scratch; however, this may not be cost effective, since all UBRs have to be recomputed and inserted to the index. We observe that the UBRs of object o before and after the change are often similar in shape. Based on this intuition, we develop an incremental version of SE, which derives the new UBR by shrinking or expanding the old UBR. We make use of this result to develop an algorithm that efficiently refreshes the PV-index.

The rest of the paper is organized as follows. We discuss related works in Section II. We discuss preliminaries about PNNQs and PV-cells in Section III. Section IV studies how to express a PV-cell by the domination concept, and Section V presents the SE algorithm. We present construction and query algorithms of the PV-index, as well as how it can be incrementally maintained, in Section VI. Our experimental results are presented in Section VII. We conclude the paper in Section VIII.

II. RELATED WORK

There are many approaches primarily using a Voronoi diagram to answer nearest neighbor (NN) queries over points [1], [18], [19]. In this context, the Voronoi diagram has been used to support nearest neighbor queries in geo-spatial applications [19], in spatial data streams [20] and, recently, in distributed spatial environments [21] as well as in spatial network environments [22]. Further database applications based on the Voronoi diagram are wireless database services [23], [24], location-based services [25], [26], and virus spread analysis [27].

Constructing a Voronoi diagram for multi-dimensional points, even in the non-uncertain case, is very costly. Hence, researchers studied its approximate form. In [28], the Voronoi
diagram was approximated as a disjoint set of convex polygonal objects. In [29], Berchold et al. developed a linear optimization algorithm for finding rectangles that tightly bound multi-dimensional Voronoi cells. The authors demonstrated that these rectangles facilitate NN queries on a large database.

Relatively few works studied the use of the Voronoi diagram for uncertain data. In [30], the Voronoi diagram is used to support uncertain data clustering. In [31], [32], the Voronoi diagram is employed to find out all uncertain objects that must be the nearest neighbor of a query point. In [33], the expected Voronoi diagram is proposed to answer nearest neighbor queries based on expected distance of uncertain objects. Recently, [9] proposed the UV-cell, which is a Voronoi cell for a circular uncertainty region. For any point \( p \) inside an object \( o \)’s UV-cell, \( o \) has a non-zero chance to be \( p \)’s nearest neighbor. Hence, the UV-cell is a special case of the multi-dimensional PV-cell studied in this paper. In [34], a Voronoi-diagram-based structure is developed for a “continuous” nearest neighbor query, where a 2D query point is constantly moving. A problem common to [9] and [34] is that their solutions are customized for 2D data – they make an extensive use of intersection and rotation operations of 2D hyperbolic curves. These operations require costly and high-precision matrix computation. As we can see in Figure 1(c), the shape of the PV-cell of a 3D uncertainty region is complex. The construction cost would thus be very high if solutions of [9] or [34] are extended to derive PV-cells with three or higher dimensions. Our approach does not generate a PV-cell. Instead, we compute UBRs, which does not require any intersection and rotation operation. We also study how to update UBRs upon object insertion or deletion; to our best understanding, this has not been addressed before.

As discussed before, evaluating the probabilistic nearest neighbor query (PNNQ) involves two steps:

- For **Step 1**, i.e., retrieving answer objects that have non-zero probabilities of being the query answer, [8] proposed a branch-and-prune solution based on the R-tree. Due to the high I/O cost involved in that solution, [9] proposed the UV-index, which stores UV-cells, in order to obtain answer objects from a 2D database. Our PV-index, on the other hand, uses PV-cells to support the retrieval of multi-dimensional objects. While [9] assumes that the uncertainty of an object is bounded within a 2D circle, we assume that the uncertainty region is a rectangle, which is a common assumption in the uncertain database literature [8], [10], [11], [13]. For 2D uncertain data, the UV- and PV-indexes register a similar performance in our experiments. Nevertheless, the construction time of the PV-index is about 15 times faster than that of the UV-index. The spatial requirement of the PV-index is also much less than that of the UV-index. Another problem of the UV-index is that if any change occurs in the database, it needs to be rebuilt from scratch; however, the PV-index can be incrementally updated.

- For **Step 2**, i.e., computing the probabilities of answer objects, [8] studied a systematic way of computing these probabilities. Since expensive integration operations are involved in this step, a number of efficient methods have been proposed. In [11], efficient methods were proposed to generate answer objects’ probability bounds without performing expensive integration operations.

While we focus on enhancing the performance of Step 1, we will also evaluate how this impacts the overall performance of PNNQ. Other variants of PNNQs, such as group NN [12] and reverse NN [13], [14], have also been studied. In all these works, the R-tree was used to support efficient object retrieval. It would be interesting to see how the PV-index can be used to facilitate these query algorithms.

III. ON POSSIBLE VORONOI CELLS

As we have discussed, PV-cells can be used to evaluate Step 1 of PNNQ, i.e., retrieving answer objects that have non-zero probabilities of being the query answer. Let us now examine them in more detail. In the following we first discuss important properties of a PV-cell. Then, in Section III-B, we show how to approximate a PV-cell.

### A. PV-cell: Basic properties

Given a point \( p \in \mathbb{R}^d \), let \( \text{dist}_{\max}(o, p) \) (\( \text{dist}_{\min}(o, p) \)) be the maximum (minimum) distance of \( o, a \) from \( p \). Suppose that \( S \) contains two uncertain objects, \( o \) and \( o' \). Consider the following \( d \)-dimensional hyperplane, \( H_{o', o} \):

\[
H_{o', o} = \{ p \in \mathbb{R}^d | \text{dist}_{\max}(o', p) = \text{dist}_{\min}(o, p) \} \quad (1)
\]

Figure 2 illustrates \( o, o', H_{O', o} \) (in solid line), and the PV-cell of \( o \) (shaded).

Figure 3 shows \( \text{dom}(a, b) \) (grey) and \( \neg \text{dom}(a, b) \), separated by \( H_{a, b} \) (dotted line).

The construction of \( H_{o', o} \) is not straightforward. In particular, if the uncertainty regions of \( o \) and \( o' \) are rectangles, the domain space needs to be decomposed into a number of small rectangular partitions [17]. Figure 2 illustrates these partitions in dotted lines. In 2D space, \( H_{o', o} \) consists of straight lines and curves. For \( d \)-dimensional space, finding all the vertices of \( H_{o', o} \) involves solving complex equations. The PV-cell of \( o \), which consists of \( H_{o', o} \), can therefore be “irregular” in shape.

The above observation also applies to a database of more than two objects. To find the PV-cell of an object \( o \), we conceptually find its PV-cell with respect to each of the other objects in \( S \). Then, the PV-cell of \( o \) must be the intersection
of these \(|S| - 1\) PV-cells. Consequently, the shape of \(\mathcal{V}(o)\) can also be quite complex (e.g., Figure 1(b)). In fact, the number of edges required to represent an exact PV-cell increases exponentially in the number of dimension. Thus the space complexity to store a PV-cell, and thus the time complexity of any algorithm using PV-cell must be exponential. An intuition of the exponential number of edges is given as follows: As pointed out by [9], a 2D PV-cell consists of a number of curvilinear one-dimensional surfaces (called edges). As illustrated in Figure 1(c), the PV-cell of a 3D object consists of a number of curvilinear 2D surfaces, each described by a number of curvilinear edges. The PV-cell of a 4D object consists of a number of 3D surfaces, each described by a number of 2D surfaces, each described by a number of edges. Clearly, a \(d\)-dimensional PV-cell is composed of a number of \((d-1)\)-dimensional surfaces, resulting in an exponential number of \(d\)-surfaces. Further information on the complexity of higher dimension Voronoi cells can be found in [35].

Due to this complexity, even in 2D space [9], we next study an approximate form of the PV-cell.

**B. PV-cell: Approximation**

One way to avoid computing the PV-cell is to approximate it with a polygon. Due to its simplicity, a minimum bounding rectangle (MBR), which is a hyper-rectangle that tightly contains a complex spatial object, is often used (e.g., [36]). The authors in [29] also studied the derivation of the MBR of a Voronoi cell. Let us now examine the efficiency of finding the MBR for a PV-cell.

**Lemma 1:** Let \(M(o)\) be the MBR of \(\mathcal{V}(o)\). There does not exist any polynomial-time algorithm for finding \(M(o)\).

**Proof:** (Sketch) Since the surfaces of \(\mathcal{V}(o)\) are concave in shape, \(M(o)\) is determined by the vertices of \(\mathcal{V}(o)\). These vertices are not readily known, since we do not know the exact shape of \(\mathcal{V}(o)\). In fact, we can view the finding of a dimension of \(M(o)\) as a convex optimization problem. In particular, any dimension of \(M(o)\) must be located in the feasible region (or solution space) \(\mathcal{V}(o)\). Since a PV-cell is composed of planes and curved surfaces, \(\mathcal{V}(o)\) cannot be a convex polygon. This implies that the feasible region of this problem is not convex. According to [37], this kind of problems does not have any polynomial-time solution. Correspondingly, no efficient solution exists for \(M(o)\).

The detailed proof of the above lemma can be found in our technical report [38]. We conclude that it is impractical to find \(M(o)\). Hence, we derive the UBR of \(\mathcal{V}(o)\) defined as follows.

**Definition 2:** Given an object \(o\), its **Uncertain Bounding Rectangle** (UBR), denoted by \(B(o)\), is a \(d\)-dimensional rectangle that completely contains \(\mathcal{V}(o)\).

A trivial \(B(o)\) is the domain space \(D\), whereas a UBR that tightly contains \(\mathcal{V}(o)\) is essentially \(M(o)\). Our goal is to develop an efficient algorithm for finding a \(B(o)\), which is only a bit looser compared to the corresponding \(M(o)\). Our experiments show that the UBRs we found are only a bit larger than their corresponding MBRs, and they enable efficient nearest neighbor retrieval. In Sections IV and V, we will study how to derive \(B(o)\). Section VI then explains how to use UBRs to evaluate a PNNQ.

**IV. PV-cell and Dominated Regions**

Our main idea of finding \(B(o)\) is to interpret \(\mathcal{V}(o)\) by using dominated regions. Section IV-A presents the concept of dominated regions. In Section IV-B, we use dominated regions to derive some fundamental properties of \(\mathcal{V}(o)\). These properties form the basis of our solution, which will be discussed in Section V.

For the detailed proofs of the lemmas discussed in this section, please refer to our technical report [38].

**A. Dominated and Non-dominated regions**

Let \(a\) and \(b\) be two uncertain objects, whose uncertainty regions \(u(a)\) and \(u(b)\) are inside the domain \(D\). Then,

**Definition 3:** The **dominated region** of \(a\) over \(b\), denoted by \(\text{dom}(a,b)\), is a subset of \(D\), such that:

\[
\text{dom}(a,b) = \{p \in D | \text{dist}_{\text{max}}(a,p) < \text{dist}_{\text{min}}(b,p)\}.
\]

**Definition 4:** The **non-dominated region** of \(a\) over \(b\), denoted by \(\neg\text{dom}(a,b)\), is \(D - \text{dom}(a,b)\), or

\[
\neg\text{dom}(a,b) = \{p \in D | \text{dist}_{\text{max}}(a,p) \geq \text{dist}_{\text{min}}(b,p)\}.
\]

**Figure 3** illustrates these two regions, which are separated by hyperplane \(H_{a,b}\) (Equation 1). When point \(p\) is inside \(\text{dom}(a,b)\), according to Definition 3, \(a\) is always closer to \(p\) than \(b\). If \(p \in \neg\text{dom}(a,b)\), then \(b\) may be closer to \(p\) than \(a\).

**Lemma 2:** \(\text{dom}(a,b) = \emptyset\) if and only if \(u(a)\) intersects \(u(b)\).

Lemma 2 allows us to quickly determine \(\text{dom}(a,b)\), by checking whether \(u(a)\) intersects \(u(b)\). Notice that \(\text{dom}(a,a) = \emptyset\).

Now, let \(A \subseteq S\) be a subset of \(S\). We introduce two notations to facilitate our discussions.

**Definition 5:** The **non-dominated intersection** of \(A\) over \(o\), denoted by \(I(A,o)\), is the intersection of non-dominated regions of objects in \(A\) over \(o\), i.e.,

\[
I(A,o) = \bigcap_{a \in A} \neg\text{dom}(a,o).
\]

**Definition 6:** The **dominated union** of \(A\) over \(o\), denoted by \(U(A,o)\), is the union of dominated regions of objects in \(A\) over \(o\), i.e.,

\[
U(A,o) = \bigcup_{a \in A} \text{dom}(a,o).
\]

The following result relates \(I(A,o)\) and \(U(A,o)\).

**Lemma 3:** \(U(A,o) = D - I(A,o)\)

We next study how these concepts can be used to derive some important properties of a PV-cell.

**B. Other Properties of PV-cell**

Lemma 4 below establishes the relationship between the PV-cell of \(o\) and the non-dominated intersection of \(S\) over \(o\):

**Lemma 4:** \(\mathcal{V}(o) = I(S,o)\)

**Proof:** (Sketch) We want to show that 1) for any point \(p \in I(S,o)\), \(o\) has a non-zero chance to be the closest to \(p\) and, 2) for any \(p \notin I(S,o)\), \(o\) has no chance to be the nearest to \(p\). If these two statements hold, then \(I(S,o)\) must be the PV-cell of \(o\).
Fig. 4. Illustrating an iteration of SE (x-dimension, low direction).

We say that $S$ is an V-set of $V(o)$. As we will explain later, V-sets other than $S$ may exist. Formally, we represent the V-set of $V(o)$ by $V_{set}(o)$, with the following definition:

**Definition 7**: The V-set of $V(o)$, denoted by $V_{set}(o)$, is a subset of $S$ such that $V(o) = I(V_{set}(o), o)$.

We also define the candidate V-set of $V(o)$, or simply C-set:

**Definition 8**: The C-set of $V(o)$, denoted by $C_{set}(o)$, is a subset of $S$ such that $V(o) = I(C_{set}(o), o)$.

Thus, $V(o)$ is bounded within the non-dominated intersection of $C_{set}(o)$ over $o$. Notice that when $C_{set}(o) = S$, $C_{set}(o)$ becomes $V_{set}(o)$. Let us now study other properties of $V(o)$.

**Lemma 5**: The uncertainty region of $o$, i.e., $u(o)$, must be completely inside $V(o)$, i.e., $u(o) \subseteq V(o)$.

Based on the above result, we can consider $u(o)$ to be a “lower bound” of $B(o)$. This is used in our UBR construction algorithm that will be detailed in Section V. Next, we have:

**Lemma 6**: $V(o)$ is a connected region.

Hence, $V(o)$ can be bounded by a single rectangle (e.g., $B(o)$). We next explain how to use these lemmas to derive $B(o)$.

V. Generating a UBR

Ideally, $B(o)$ is the MBR of $o$ (i.e., $M(o)$). However, as discussed before, finding $M(o)$ is extremely expensive. We now present the Shrink-and-Expand (or SE) algorithm for efficiently computing a $B(o)$, which is only slightly larger than $M(o)$.

The main idea of SE is to estimate $M(o)$ with the aid of a pair of $d$-dimensional rectangles: the lower bound $l(o)$, which is enclosed by $M(o)$; and the upper bound $h(o)$, which contains $M(o)$. In other words, $M(o)$ is sandwiched between $l(o)$ and $h(o)$. The algorithm iteratively adjusts the size of these rectangles until $M(o)$ is accurately represented by them. Specifically, in each iteration, SE performs either one of the operations:

- **Shrink**: Reduce the size of $h(o)$, by pruning regions that must not be part of $M(o)$, and;
- **Expand**: Increase the size of $l(o)$, by including regions that are assured to be inside $M(o)$.

When the distance between $l(o)$ and $h(o)$ is smaller than some threshold value $\Delta$, SE outputs $h(o)$ as the UBR of $o$. Figure 4(a) illustrates $V(o)$ (in grey), $M(o)$, $l(o)$, and $h(o)$, in 2D space.

Algorithm 1 shows the details of SE. Step 2 executes the procedure chooseCSet, which returns a C-set of $V(o)$. Here, we assume that chooseCSet returns $S$, but we will discuss another implementation of this procedure in Section V-A. Notice that $S$ is indeed a C-set, since according to Lemma 4, $V(o) = I(S, o)$. Step 3 initializes the bounds $l(o)$ and $h(o)$. For $l(o)$, we use the uncertainty region of $o$ (i.e., $u(o)$) as the initial value of $l(o)$. This is correct, because $u(o) \subseteq V(o)$ (Lemma 5), and $V(o) \subseteq M(o)$. For $h(o)$, we use the domain $D$ as its initial value. Next, in every iteration (Steps 4-12), the shrinking of $h(o)$ and the expansion of $l(o)$ are carried out, until the condition defined in Step 4 is satisfied. (We will explain this condition later.) Finally, Step 13 returns $h(o)$ as the UBR of $V(o)$.

We now discuss Steps 4-12 in more detail. Let $\rho = \{\text{low}, \text{high}\}$ be the “direction” of object $o$ along the $j$-th dimension (where $j = 1, \ldots, d$). For example, in Figure 4(a), $\rho = \text{low}$ (high) denotes the left (right) of $o$ along the x-axis. As shown in Steps 5 and 6, shrinking and expansion are done for each direction $\rho$ of dimension $j$. Step 7 computes $i^j_{\text{low}}(o)$, which is a hyperplane in the middle of $h(o)$ and $l(o)$, in direction $\rho$ along the $j$-th dimension. Figure 4(b) illustrates $i^j_{\text{low}}(o)$, where dimensions 1 and 2 denote the x- and y-axes respectively. In Step 8, we consider the region between $i^j_{\text{low}}(o)$ and $h(o)$, denoted by $R^j_h$. Figure 4(b) demonstrates these regions. In Step 9, we test whether $R^j_h$ overlaps $I(C_{set}(o), o)$. According to Definition 8, $V(o)$ is bounded by $I(C_{set}(o), o)$. Hence, if $R^j_h$ does not touch $I(C_{set}(o), o)$, it must also not touch $V(o)$. This $R^j_h$ cannot be part of $M(o)$, and can be safely removed from $h(o)$ (Step 10). Otherwise, we expand $l(o)$ in direction $\rho$, dimension $j$, up to the position of $i^j_{\text{low}}(o)$ (Step 11). Figure 4(c) shows that $h(o)$ is shrunk with the removal of $R^j_{\text{low}}$. The shrinking-and-expanding process is repeated until the maximum distance between the boundaries $h(o)$ and $l(o)$,
In our experiments, by using a small following example.

\[
\text{Problem 1:} \quad \text{The whole database (i.e., } S) \text{ is used to compute } I(\mathcal{C}_{\text{set}}(o), o) \text{ in Step 9, since } \text{chooseCSet} \text{ (Step 2) returns } S. \text{ If } |S| \text{ is large, Step 9 can take a long time to run.}
\]

\[
\text{Problem 2:} \quad \text{Evaluating } R_i^o \cap I(\mathcal{C}_{\text{set}}(o), o) \text{ is costly, since this involves computing intersections of multi-dimensional non-dominated regions, whose shapes can be complex.}
\]

In the worst case, Step 9 has to be executed \(2d \cdot \log(|D|_{\max}/\Delta)\) times. We next study how to tackle Problem 1, in Section V-A. We examine an efficient method for solving Problem 2 in Section V-B.

A. Designing the chooseCSet Routine

To handle Problem 1, let us consider chooseCSet again, which was made to return \(S\) in the previous section. In fact, this is not necessary. Let us first show the following lemma, the proof of which can be found in the technical report [38].

\[
\text{Lemma 7:} \quad \text{Given an object } o \in S, \text{ any non-null subset } T \text{ of } S \text{ is a } \mathcal{C}\text{-set of } V(o).
\]

Hence, \(S\) is not the only \(C\)-set. Since any subset of \(S\) can be a \(C\)-set, can we just tell chooseCSet to return an arbitrary object \(o' \in S\)? If we do this, \(I(\mathcal{C}_{\text{set}}(o), o)\) is simply \(\neg \text{dom}(o', o)\), and Step 9 of \(S\) can be efficiently executed. Unfortunately, the UBR of \(o\) returned by \(S\) can be much larger than its corresponding PV-cell, as illustrated by the following example.

Example. In Figure 5, \(S = \{o, o_1, \ldots, o_5\}\). The boundary of \(V(o)\) is drawn in bold lines, and one of the \(V\)-sets of \(V(o)\) is \(\{o_2, o_4, o_5\}\). Notice that \(o_1\) and \(o_3\) are not in this \(V_{\text{set}}(o)\).

Recall that \(V(o) = I(S, o)\) (Lemma 4), or the intersection of \(\neg \text{dom}(o', o)\), for every \(o' \in S\). Since \(u(o)\) overlaps \(u(o_1)\), by Lemma 2, \(\neg \text{dom}(o_1, o)\) is D. Hence, \(o_1\) does not affect the shape of \(V(o)\), and \(o_1\) can be excluded from \(V_{\text{set}}(o)\). For \(o_3\), notice that \(\text{dom}(o_3, o)\) (the grey region) does not intersect \(V(o)\). Consequently, \(\neg \text{dom}(o_3, o)\) does not affect \(V(o)\), and \(o_3\) can also be pruned from \(V_{\text{set}}(o)\).

Now, suppose that chooseCSet returns \(o_1\). Then, \(I(\mathcal{C}_{\text{set}}(o), o)\) becomes \(I(\{o_1\}, o)\), or just \(D\). As a result, in Step 9 of \(S\), \(R_i^o \cap D\) is always not null, and \(h(o)\) will not be shrunk at all. Consequently, \(h(o)\), which is initialized to \(D\) (Step 3), will be returned. This UBR returned by \(S\) may not be desirable, since it may be much larger than \(V(o)\) or \(M(o)\).

With a similar argument, when \(C_{\text{set}}(o) = \{o_3\}\), \(h(o)\) cannot be shrunk to tightly bound \(M(o)\).

To ensure that \(S\) returns a small MBR, a thoughtful design of chooseCSet is important. Notice that if \(C_{\text{set}}(o) = V_{\text{set}}(o)\), \(S\) can attain the highest effectiveness. This is because the condition tested in Step 9 becomes \(R_i^o \cap V(o)\) (Lemma 4). If \(\Delta = 0\), the UBR returned is exactly \(M(o)\).

Again, \(S\) is one of the \(V\)-sets of \(o\). To solve Problem 1, however, it is desirable to obtain a \(V_{\text{set}}(o)\) with the minimal size. For example, \(\{o_2, o_4, o_5\}\) is the minimum \(V\)-set of \(o\), denoted by \(V_{\text{set}}^{\text{min}}(o)\), in Figure 5. Unfortunately, it is not easy to derive this set: for every \(s \in S\), we have to compute the boundary \(H_{s,o}\), and check whether \(H_{s,o}\) constitutes \(V(o)\). This is similar to the computation of \(V(o)\), which as discussed in Section III, is extremely expensive.

We propose two implementations of chooseCSet. They derive a small \(V\)-set based on some simple observations about \(V(o)\), as detailed below:

1. Fixed Selection (FS). This algorithm returns \(k\) objects whose mean positions are the closest to the mean position of \(o\). In Figure 5, for instance, if \(k = 2\), then \(C_{\text{set}}(o) = \{o_1, o_2\}\). The FS solution assumes that if object \(a\) is closer to \(o\) than object \(b\), then \(a\) has a higher chance to be included in \(V_{\text{set}}^{\text{min}}(o)\) than \(b\). In Figure 5, \(o_2\) is closer to \(o\) than \(o_3\), and so \(V_{\text{set}}^{\text{min}}(o)\) contains \(o_2\), but not \(o_3\).

Despite of the simplicity of FS, it faces four problems:

- The C-set returned is sensitive to \(k\). If \(k\) is too small, the C-set may not include all members of \(V_{\text{set}}^{\text{min}}(o)\). In the previous example, although \(o_5\) is a member of \(V_{\text{set}}^{\text{min}}(o)\), it is not returned by FS, since \(k = 2\). On the other hand, if \(k\) is too large, the C-set may contain objects that do not belong to \(V_{\text{set}}^{\text{min}}(o)\).
- Since the positions of the objects’ uncertainty regions may not be uniformly distributed, the PV-cells of any two objects can be very different in shape. Thus, the size of the minimum \(V\)-set may not be the same for different objects, and it is not easy to find a single value of \(k\) that is close to the minimum \(V\)-set size of every object.
- Objects far away from \(o\), but are in \(V_{\text{set}}^{\text{min}}(o)\), may not be chosen by FS. In Figure 5, if \(k = 2\), \(o_5\), which is not near to \(o\), will not be selected. However, \(o_5 \in V_{\text{set}}^{\text{min}}(o)\).
- FS does not discard objects whose uncertainty regions

1Specifically, \(|h(o) - l(o)|_d\) is the maximum distance between \(h(o)\) and \(l(o)\), among all dimensions.
overlap \(u(o)\). As explained before, these objects should not be included in a C-set, since it does not affect \(V(o)\) at all. In Figure 5, although \(u(o_1)\) intersects \(u(o)\), \(o_1\) is also returned by chooseCSet.

Let us see how the next solution alleviates the above problems.

2. Incremental Selection (IS). In this approach, \(C_{sett}(o)\) is determined by examining objects in ascending order of distance from \(o\). This not only avoids the problem of setting a fixed \(k\) (in FS), but also allows objects whose uncertainty regions overlap \(u(o)\) to be skipped. Moreover, as discussed next, the search of objects span the whole domain. This increases the chance an object that is in \(V_{sett}^{\text{min}}(o)\) but far away from \(o\) to be selected.

In detail, domain \(D\) is conceptually divided into \(2^d\) disjoint partitions, based on the mean position of \(o\). Figure 5 illustrates the four partitions of a 2D object \(o\) separated by dotted lines. Each partition is associated with a counter variable, which tracks the number of NN’s that has been considered in the partition so far. The solution examines the nearest neighbor of \(o\) one at a time, using the algorithm in [39]. Suppose that the current NN of \(o\) is \(n\). If \(u(n)\) intersects any partition(s), and \(u(n)\) does not intersect \(u(o)\), the counters associated with these partitions will be incremented by one. The algorithm stops either when the counter values of all partitions are at least \(k_{\text{partition}}\), or when \(k_{\text{global}}\) nearest neighbors of \(o\) are examined. Suppose that \(k_{\text{partition}} = 2\) and \(k_{\text{global}} = 10\) in Figure 5. Then, \(o_1\) and \(o_2\) will be retrieved by IS, since they are the only two NN’s of \(o\) in the upper-right partition. On the other hand, \(o_1\) is not returned, since IS detected that \(u(o_1)\) intersects \(u(o)\). In this example, IS returns \(\{o_2, o_3, o_4, o_5\}\), which includes all members of \(o\)’s minimum V-set. The full algorithm of IS can be found in the technical report [38].

One benefit of IS over FS is that it does not need to set the value of \(k\) anymore. Although IS needs to determine \(k_{\text{partition}}\) and \(k_{\text{global}}\), our experiments show that the results not very sensitive to these parameters. Another advantage of IS is that objects that are far away from \(o\) and are not considered by FS may also be returned. In Figure 5, for instance, \(o_5\) is ignored by FS, but is returned by IS. This is because IS requires that the number of NN’s found in every partition of \(D\) is at least \(k_{\text{partition}}\).

Remarks. Although IS and FS may not return \(V_{sett}^{\text{min}}(o)\), they are still correct, since according to Lemma 7, any subset of \(S\) can be \(C_{sett}\). For complexity, notice that FS executes a \(k\)-NN query. The worst-case cost of IS is also that of a \(k\)-NN query, with \(k = k_{\text{global}}\). Hence, the worst-case complexity of both solutions is \(O(|S|)\). With the aid of a data structure (e.g., an R-tree of objects’ uncertainty regions for efficient NN retrieval), both IS and FS can be run efficiently. In our experiments, the size of the C-sets returned is usually much smaller than \(|S|\), and so Problem 1 is addressed.

B. Non-Dominated Region Intersection Test

Recall that Step 9 of SE checks whether the intersection of \(R_j^o\) and \(I(C_{sett}(o), o)\) is equal to null. This test, whether there is any intersection at all, can be performed efficiently, even though the task of computing the concrete intersection set is hard. A simple way to perform this test is to compute \(R_j^o \cap I(C_{sett}(o), o)\) directly. As mentioned in Problem 2 (Section V-A), this involves calculating the intersection of multi-dimensional non-dominated regions. Since the shape of these regions can be complex, obtaining their precise intersection points is extremely expensive. To check this condition efficiently, we design a solution that does not compute any intersection of non-dominated regions. The main idea is to use the techniques proposed in [17] denoted as spatial domination and domination count estimation. The concept of spatial domination allows to efficiently decide, for three given rectangles \(A\), \(B\) and \(R\) whether it holds that for any triple of points \(a \in A\), \(b \in B\), \(r \in R\), \(a\) is closer to \(r\) than \(b\). This equals the decision whether \(B\) is completely contained in the region \(\text{dom}(A, R)\). In Figure 6(a), this technique allows to decide that \(R\) is completely contained in the dominating region \(\text{dom}(a, b)\). The concept of domination count estimation, essentially splits object \(R\) into a set of partitions, and applies the concept of spatial domination to each partition individually. If for each partition \(R_i\) of \(R\), it holds that there exists an object \(X \in S\) such that \(X\) spatially dominates \(R_i\) with respect to \(b\), then we can conclude that \(R\) cannot intersect the non-dominating region of \(b\). This test corresponds to testing whether the domination count of \(b\) is greater than zero. An example is given in Figure 6(b), where \(R\) is neither completely contained in \(\text{dom}(a_1, b)\) nor in \(\text{dom}(a_2, b)\). However, it still holds that any point \(r \in R\) is contained in either these regions. The concept of domination count estimation aims at detecting this situation.

Remarks. The intersection test as described above is an approximate solution, specifically not all cases where \(R_j^o\) does not intersect \(I(C_{sett}(o), o)\) are detected. The accuracy primarily depends on the granularity of the partitioning of \(R\). However, the granularity of the partitioning process in turn influences the runtime of the intersection test. Since for each partition of \(R\), the spatial domination test has to be performed for each object in \(C_{sett}(o)\), and the spatial domination test is linear in the dimensionality of the dataset, the runtime of the intersection test is in \(O(|\text{part}(R)| \cdot |C_{sett}(o)| \cdot d)\), where \(|\text{part}(R)|\) denotes the number of partitions of \(R\).

VI. The PV-index

We now discuss how the PV-index uses UBRs to support PNNQ evaluation. Section VI-A presents the querying and
construction of this structure. We explain how to efficiently update the PV-index in Section VI-B. In the sequel, we assume that the UBR of every object in the database has been generated, based on the solutions discussed in Section V.

A. Index Design

The PV-index contains two parts: a primary index, which facilities data pruning, and a secondary index, which stores the UBR and uncertainty information of each object. The primary index is based on a multi-dimensional octree [40], while the secondary index is an extensible hash table [41]. Figure 7 illustrates the PV-index for 2D uncertain objects. In this case, the primary index is a quad-tree, whose root node covers the whole domain. A non-leaf node contains pointers to its $2^d$ child nodes; the region associated with each child node is $1/2^d$ of that of its parent. We do not store the region represented by each child node, because this information can be derived from its parent. A leaf node stores the IDs of objects whose UBRs overlap the region associated with the leaf node. The uncertainty regions of these objects are stored there too. We keep all the non-leaf nodes in the main memory. The leaf nodes are stored in the disk, each of which is a linked list of disk pages. For the secondary index, an entry is accessible by the object ID. For every entry in this entry, we store the object’s UBR, as well as its uncertainty pdf. The secondary index is stored in the disk.

Query Evaluation. The PV-index supports Step 1 of PNNQ evaluation, i.e., retrieval of objects with non-zero qualification probabilities. Starting from the root of the primary index, we access the child nodes whose associated regions contain the query point $q$. This is repeated until we reach the leaf node $n_{leaf}$, whose region contains $q$. The list $L$ of IDs stored in $n_{leaf}$ correspond to objects that may constitute PNNQ answers. Notice that the UBR of any object $o$ in $L$ overlaps the region spanned by $n_{leaf}$. Since $q$ can be inside the PV-cell of $o$, $o$ is possibly a nearer neighbor of $q$. However, some objects in $L$ may not qualify for the answer; for these objects, their PV-cells do not contain $q$. These objects can be pruned by checking whether their minimum distances from $q$ are larger than the minimum of the maximum distances of objects in $L$ from $q$. Objects that remain in $L$ are those whose qualification probabilities exceed zero. Their probabilities are then computed in Step 2, using the uncertainty information stored in the secondary index. We implement Step 2 based on the method in [8]; in practice, any solution mentioned in Section II can be used. Our technical report [38] discusses this in detail.

Index Construction. The PV-index is created by inserting UBRs into it sequentially. Initially, its primary index is a root node with an empty page, and its secondary index is a hash table. We also allocate a fixed amount of main memory to store the non-leaf nodes of the primary index. The UBR $B(o)$ of every object $o \in S$ is then inserted to the index as follows:

1) Perform a range search on the PV-index, using $B(o)$, and locate the leaf nodes whose regions overlap $B(o)$.
2) For every node $n_{leaf}$ obtained in Step 1, if the first page in the list of $n_{leaf}$ is not fully occupied, insert ($ID$ of $o, u(o)$) to it.
3) Suppose that all pages in $n_{leaf}$ are full. If there is not enough main memory to allocate a new non-leaf node, attach a new page to the head of the list in $n_{leaf}$, and insert ($ID$ of $o, u(o)$) to it. Otherwise, make $n_{leaf}$ to be the parent of $2^d$ new child leaf nodes. (Thus, $n_{leaf}$ becomes a non-leaf node.) We then re-insert the UBRs whose corresponding objects were previously contained in $n_{leaf}$, to the new child nodes.
4) Insert an entry ($B(o), u(o), pdf$) to the secondary index.

Since both the region of a node and $B(o)$ are rectangles, checking whether they overlap is easy. Let $M$ and $K$ be the sizes of the main memory and disk page respectively. Then, the PV-index has at most $[M/2^{d+2}] \cdot (1 + 2^d)$ nodes. The construction cost of the index is $O((M + cost_{SE}) \cdot |S|)$, where $cost_{SE}$ denotes the time complexity of $SE$. Evaluating Step 1 of PNNQ requires a cost of $O(\log(M/2^{d+2}) + |S|/K)$. For details, please refer to the technical report [38].

B. Updating the PV-Index

We now study the maintenance of the PV-index. After a database has been changed, its associated PV-index also needs to be refreshed, in order to allow queries to be answered correctly. A simple yet expensive solution is rebuild the index from scratch. We now introduce an incremental solution, which only modifies part of the index. This solution supports two common operations: object insertion and deletion.

Change of PV-cell. Our approach is based on understanding how a PV-cell is impacted by an update on database $S$. We found that a PV-cell may remain unchanged after an update is applied to $S$. Specifically, let $o'$ be the object to be inserted to (or removed from) $S$, and $S'$ be the resulting database. Also, let $V(X, o), M(X, o)$ and $B(X, o)$ be the PV-cell, MBR, and UBR of $o$ derived from database $X$ respectively, with $X \in \{S, S'\}$. We say that an object $o$ (where $o \in S \land o \neq o'$) is affected, if $V(S, o)$ and $V(S', o)$ are different upon a database update. Lemma 8 lists the conditions for $o$ to be not affected:

---

2Each uncertainty pdf is discretized by 500 samples in our experiments. Moreover, the region spanned by a leaf node can be touched by up to $|S|$ UBRs. We thus store this in the disk.

3Alternatively, an R-tree can be used to implement the primary index. We choose the octree, because its grids do not overlap. This enables efficient evaluation of point query $q$. 
Figure 8. Illustrating deletion of \( o' \): (a) UBRs of \( o' \) (dotted) and objects that may be affected (bolded); (b) UBRs of an affected object before and after deletion (dotted and bolded rectangles).

**Lemma 8:** Object \( o \) is not affected if:
1. \( \forall(S, o) \cap \forall(S, o') = \emptyset \) (for deletion of \( o' \)); or
2. \( \forall(S, o) \cap \forall(S', o') = \emptyset \) (for insertion of \( o' \)); or
3. \( u(o) \cap u(o') \neq \emptyset \)

Condition (1) says that \( o \) is not affected if upon removal of \( o' \) from \( S \), the PV-cells of \( o \) and \( o' \) derived from \( S \) do not intersect. In Condition (2), if \( o' \) is inserted to \( S \), and \( \forall(S, o) \) does not overlap \( \forall(S', o') \), the PV-cell of \( o \) does not change. As shown in Condition (3), if the uncertainty regions of \( o \) and \( o' \) do not intersect, then once \( o \) is unaffected. These conditions can be proved using the domination results in Section IV, as detailed in our technical report [38].

We can thus use the above conditions to discard unaffected objects, whose PV-cells (and UBRs) do not change. For any object that may be affected, the following summarizes how their PV-cells evolve:

**Lemma 9:** The PV-cell of an affected object \( o \):
- Cannot be smaller than before, if \( o' \) is deleted from \( S \);
- Cannot be larger than before, if \( o' \) is inserted to \( S \).

To show that the above is correct, we use Lemma 7. The detailed proof is in our technical report [38]. Lemma 9 allows us to use the old UBR of \( o \), i.e., \( B(S, o) \) to derive the new one, i.e., \( B(S', o) \). We next explain how to do this efficiently.

**An Incremental Solution.** We can now describe our incremental update algorithm. We assume that \( S \) has been updated to \( S' \). Let us first describe a four-step solution for handling the deletion of \( o' \).

1. **Step 1** Retrieve \( B(S', o') \) from the secondary index, using the ID of \( o' \).
2. **Step 2** Identify the set \( A \subseteq S \) of objects that may be affected.
3. **Step 3** Compute the new UBRs of objects in \( A \).
4. **Step 4** Refresh the PV-index with the new UBRs.

We next describe the details of Steps 2-4:

**Step 2** We first issue a range query on the primary index, using the range \( B(S, o') \), and obtain leaf nodes whose space in \( D \) overlap \( B(S, o') \). Since the UBRs of the objects found in these nodes overlap the regions represented by these nodes, the PV-cell of these objects may also touch \( B(S, o') \). Let the set of all objects found in these nodes be \( A \). Then, from the primary index, we obtain the uncertainty regions of these objects, and exclude any object \( o \in A \) where \( u(o) \cap u(o') \neq \emptyset \). Next, from the secondary index, we retrieve their UBRs, and discard any object \( o \) where \( B(S, o) \cap B(S, o') = \emptyset \), or correspondingly, \( \forall(S, o) \cap \forall(S, o') = \emptyset \). Notice that these discarded objects satisfy Conditions (1) or (3) of Lemma 8. The remaining objects in \( A \) may be affected. Figure 8(a) shows the UBRs of \( o' \) and the objects in \( A \), in dotted and bolded rectangles, respectively.

**Step 3** For every \( o \in A \), \( \forall(S, o) \), as well as \( B(S, o) \), may be changed after the update. To obtain \( B(S', o) \), we use a slightly-changed version of \( SE \): in Step 3, rather than initializing the lower bound \( l(o) \) to \( u(o) \), we set \( l(o) \) to be \( B(S, o) \), which was stored in the secondary index. As discussed in Lemma 9, \( \forall(S', o) \) cannot be smaller than \( \forall(S, o) \); hence, \( B(S', o) \) may also be bigger than \( B(S, o) \). We can thus use \( B(S, o) \) as \( l(o) \). 4

Since \( B(S, o) \) cannot be smaller than \( u(o) \), the gap between \( h(o) \) and \( l(o) \) is smaller in Step 3 of \( SE \). As a result, the shrinking-and-expansion process runs more quickly.

**Step 4** We first remove \( B(S, o') \) from the primary index. This is done by locating the leaf nodes whose regions overlap \( B(S, o') \) found in Step 2. We then remove all entries related to \( o' \) from these nodes. We also delete the entry of \( o' \) from the secondary index. For every affected object \( o \in A \), we extract the sets \( N \) and \( N' \) of leaf nodes, whose regions overlap \( B(S, o) \) and \( B(S', o) \) respectively. The entries of \( o \) are then inserted to the set of nodes in \( N' - N \). Figure 8(b) shows the (grey) set of leaf nodes \( (N' - N) \) where the entry of an affected object has to be inserted. Notice that since \( B(S, o) \) is covered by \( B(S', o) \), we do not have to handle the nodes in \( N \). The UBR information of \( o \) in the secondary index is updated accordingly. 5

**Insertion** can be handled in a similar manner:

**Step 1** Retrieve \( B(S', o') \) by running \( SE \) on \( S' \).

**Step 2** Identify the set \( A \) of affected objects, by issuing a range query \( B(S', o') \) on the primary index. For any \( o \in A \), remove \( o \) from \( A \) if \( u(o) \cap u(o') \neq \emptyset \) or \( B(S, o) \cap B(S', o') = \emptyset \) as stated in Lemma 8, \( o \) satisfies Conditions (2) or (3), and is an unaffected object.

**Step 3** For every \( o \in A \), obtain \( B(S', o) \) by running a modified version of \( SE \), where \( h(o) \) is set to \( B(S, o) \), instead of \( D \). This is correct, because Lemma 9 states that \( \forall(S', o) \) cannot be larger than \( \forall(S, o) \); also \( \forall(S, o) \) is completely inside \( B(S, o) \). Since now \( B(S, o) \) is smaller than \( D \), \( SE \) can start with a smaller \( h(o) \) and yield a UBR more efficiently.

**Step 4** For every \( o \in A \), retrieve the sets \( N \) and \( N' \) of leaf nodes, whose regions overlap \( B(S, o) \) and \( B(S', o) \) respectively. Remove entries of \( o \) from the set of nodes in \( N - N' \). Then, insert the UBR of \( o' \) to the PV-index, using the index construction algorithm described in Section VI-A.

**Complexity.** For both deletion and insertion, the worst-case cost of incremental is \( O((M + cost_{SE}) \cdot |S|) \). Our technical report [38] describes how to derive this result, for both insertion and deletion. Notice that this is the same as the cost of rebuilding the PV-index. In our experiments, however, incremental is about two orders of magnitude faster than constructing the index from scratch.

4 Even if \( B(S, o) \) is larger than \( M(S', o) \), \( SE \) is still correct, since \( h(o) = D \) is always an upper bound of \( M(S', o) \).

5 We choose not to update the non-leaf nodes in the primary index, because this can trigger a lot of update operations. Our approach still returns correct query answers efficiently, as shown in our experiments.
We now report our results. Section VII-A describes the experiment setup. In Section VII-B, we compare the query performance of different indexes. We then present a detailed analysis of the PV-index in Section VII-C.

### A. Setup

We have evaluated our approaches on synthetic and real datasets. Synthetic data are generated by using Theodoridis et al.’s data generator.\(^6\) The mean attribute values of uncertain objects are uniformly distributed in domain \(D = [0, 10K]^d\), where \(d = 3\) by default. The length of an attribute’s uncertainty region, \(u(o)\), in each dimension is uniformly distributed in \([1, |u(o)|]\), where \(|u(o)|\) denotes the maximum length of \(u(o)\) in a dimension. We adopt the discrete model\(^1\), \(^2\), by representing an object’s uncertainty pdf with 500 points randomly sampled within the uncertainty region, each of which exists with a probability of 1/500. The number of instances in our experiments is in the order of 10\(^7\), and sizes of these datasets are within 0.2 and 1 GB. Table I lists the values of parameters used in our experiments.

For the three real datasets used, two of them, called roads (30k) and rlines (36k), contain 2D rectangular regions.\(^7\) The third one, named airports, records 3D coordinates (i.e., latitude, longitude, and height) of 20k airports in the US.\(^8\) A airport location was collected by GPS devices, whose measurement error is a 10m-radius sphere.\(^9\) These uncertainty regions are represented by their corresponding minimum bounding rectangles. The uncertainty pdf of each object in these datasets is a normal distribution, with mean equal to the object’s reported location, and variance equal to 1. This pdf is again discretized by 500 samples.

Each PNNQ is generated by randomly selecting a query point from \(D\). We compare three indexes, namely R-tree, UV-index, and PV-index, in terms of their performance in retrieving objects with non-zero qualification probabilities (i.e., PNNQ Step 1). For R-tree, objects are indexed by an R*-tree\(^42\) with a fanout of 100. This R-tree is also used to build UV- and PV-indexes. For UV-index, we implement the solution of [9] for 2D uncertain data. The default settings of [9] are used. For PV-index, the default values of \(k, k_{\text{partition}},\) and \(k_{\text{global}}\) are 200, 10, and 200 respectively. The IS strategy is used to implement chooseSet by default. The non-leaf nodes of these three indexes are all stored in 5 Mb of main memory, while their leaf nodes and object information are kept in 4k disk pages.\(^9\) For computing the actual probabilities of the objects (i.e., PNNQ Step 2), we implement the solution in [8]; the details of this step can be found in the technical report\(^38\).

Each data point reported is an average of 50 runs. Unless stated otherwise, our discussion refers to synthetic datasets. We test our solutions on a PC with an Intel Core2 Duo 2.83GHz processor and 2GB RAM. The source codes of our implementation are publicly available.\(^10\)

### B. Query Performance

We first compare the query time \(T_q\) required by R-tree and PV-index, on 3D datasets. Figure 9(a) shows that under a wide range of database size \(|S|\), PV-index is 38 – 40% faster than R-tree. To understand why, let us consider Figure 9(b), which displays the major components of \(T_q\) (Step 1, i.e., object retrieval, or OR), and Step 2 (i.e., probability computation, or PC). While the amount of time spent on PC is the same for both methods, the time invested by PV-index on OR is about 1/6 of R-tree. Notice that OR involves traversing non-leaf nodes (in main memory) and leaf nodes (in the disk), for both methods. The time required for visiting non-leaf nodes is very small (less than 0.1ms). However, as illustrated by Figure 9(c), the cost of accessing leaf nodes for PV-index is only 20% of that of R-tree. Given a query point \(q\), in PV-index only one leaf node and its list pages has to be accessed. Due to the overlapping nature of the bounding rectangles in R-tree, \(q\) may be contained by the regions associated with one or more nodes. Thus, PV-index is much faster than R-tree in the OR phase, and this leads to a superior query performance.\(^11\)

Figure 9(d) shows that for both PV-index and R-tree, \(T_q\) increases with the size of uncertainty region, \(|u(o)|\). This is because the chance that an object contributes to a PNNQ answer increases. Again, since PV-index has a better I/O performance, it is consistently faster than R-tree.

**Dimensionality.** Figure 9(e) shows that PV-index is 20 – 40% faster than R-tree in different dimensionality \(d\). This is because the time spent on the OR phase (i.e., \(T_{OR}\)) by PV-index is less than R-tree (Figure 9(f)). The improvement is due to the fact that PV-index performs better than R-tree in terms of I/O (Figure 9(g)). Moreover, when \(d \geq 3\), the fraction of time \(T_q\) spent by R-tree on OR is over 60%. Thus, the performance of PNNQ can be improved significantly by the decrease in \(T_{OR}\). Although PV-index and UV-index perform similarly, UV-index only supports 2D data.

Observe from Figure 9(e) that \(T_q\) is minimal at \(d = 3\). When \(d\) increases, \(D\) becomes bigger, and so objects in \(D\) are more separated. Since fewer objects satisfy the PNNQ, the time cost of PC, \(T_{PC}\), drops with \(d\). However, the time cost of OR, \(T_{OR}\),

<table>
<thead>
<tr>
<th>parameter</th>
<th>values (synthetic)</th>
<th>values (real)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>S</td>
<td>)</td>
</tr>
<tr>
<td>(d)</td>
<td>2, 3, 4, 5</td>
<td>2, 3</td>
</tr>
<tr>
<td>(</td>
<td>u(o)</td>
<td>)</td>
</tr>
<tr>
<td>(\Delta)</td>
<td>0.1, 0.3, 1, 10-1000</td>
<td>1</td>
</tr>
<tr>
<td>(k_{\text{max}})</td>
<td>20, 40, 100, 200, 400</td>
<td>100</td>
</tr>
</tbody>
</table>

**TABLE I**

Parameters and their default values (in bold).

8http://www.ourairports.com/data/

9http://www.rtreeportal.org/software/SpatialDataGenerator.zip

10http://www.cs.hku.hk/~pwzhang/pvc.zip

11Notice that fast solutions such as [11] can be used to implement PC. Then, the fraction of \(T_q\) time spent on OR is increased. Thus, enhancing the time for OR becomes more important.
increases with $d$ (Figure 9(f)). We also found that the rate of change of $T_{PC}$ is lower than that of $T_{OR}$. This explains why $T_q$ is the smallest at $d = 3$.

**Real datasets.** As shown in Figure 9(h), for 2D datasets (roads and rlines), UV and PV-indexes are about 40% faster than R-tree. In the 3D dataset (airport), PV-index is 45% better than R-tree. Hence, Voronoi-based techniques outperform R-tree in PNNQ evaluation.

**C. Analysis of the PV-Index**

In this section, we study the effect of different parameters on the performance of the PV-index. We also present results about construction and update of this index.

**a) Parameter Testing.** We first examine the effect of $\Delta$, $k$, and $k_{partition}$, on the query performance of the PV-index. We found that $T_q$ is quite stable, except when the parameter values are extremely high or low (e.g., $\Delta > 500$). It is thus not very hard to choose parameters to attain a high query performance. The indexes created by using FS and IS strategies also yield a similar performance. We also study the effect of these parameters on the index construction time $T_c$. Figure 10(a) shows that $T_c$ drops with the increase of $\Delta$. This is because $SE$ needs fewer iterations to compute the UBR. We also found that $T_c$ increases with $k$ and $k_{partition}$. The detailed results can be found in [38].

**b) Index Construction.** We study how FS and IS, used in chooseCSet, affect $T_c$. We compare these methods with a naive solution, called ALL, where chooseCSet return $S$ as the C-set. Figure 10(b) shows that ALL is extremely inefficient; when $|S| = 20k$, the construction time is 103 hours. However, FS and IS needs 10 minutes or less to complete. Figures 10(c) and 10(d) compare IS and FS over different values of $|S|$ and $|u(o)|$ respectively. Observe that IS is always better than FS. This is explained by Figure 10(e), which shows the two major time components of $SE$: (1) Run chooseCSet; and (2) Compute the UBR. (The time for inserting the UBR to the PV-index, which is relatively small (less than one second), is omitted here). Observe that most of the time is spent on computing the UBR. Although IS is slower than FS, it can select a smaller and better C-set. In particular, while FS returns 200 objects, IS returns 120 objects on average. Hence, IS can compute the UBR more efficiently than FS.

**Real datasets.** As shown in Figure 10(f), IS is faster than FS for all the datasets tested. Figure 10(g) compares the time for constructing the PV- and the UV-index on 2D datasets. We can see that the construction time of the PV-index is 15-25 times faster than the UV-index.

**c) Index Update.** Finally, we compare the incremental update algorithm (Inc) and the solution that rebuilds the index. For deletion, we randomly remove 1K objects from $S$. For insertion, we use the database where the 1K objects have been removed, and re-insert all those objects to it. We measure the average time $T_u$ to handle insertion/deletion per object. Figure 10(h) shows that Inc is more than two orders of magnitude faster than Rebuild. For example, at $|S| = 20K$, $T_u = 350s$ for Rebuild, but $T_u = 2s$ for Inc. For deletion, Figure 10(i) shows that Inc is much faster than Rebuild. For the real datasets, the average speedup of Inc over Rebuild is 1,092 (for insertion) and 1,258 (for deletion). Notice that the query performance of the indexes generated by Inc and Rebuild is highly similar: for object insertion (deletion), the average difference of $T_q$ between Inc and Rebuild is 1.44% (0.88%). Thus, Inc does not impact query performance significantly.

**VIII. Conclusion**

Evaluating PNNQs over a multi-dimensional uncertain database is an important and challenging problem. In this paper, we study a PNNQ algorithm based on PV-cells. We found that while a PV-cell is difficult to derive and store, finding its MBR can be much more efficient. We also propose the PV-index, which stores these MBRs in a systematic manner, in order to efficiently answer a PNNQ. The PV-index can be incrementally refreshed to reflect the changes occurring in the underlying database. In the future, we will study how to use the PV-index to support other queries, e.g., group NN [12] and reverse NN [13], [14]. We are also interested in
developing other precomputation techniques (e.g., bulkloading and compression) for facilitating the access of uncertain data.

ACKNOWLEDGMENTS

Reynold Cheng and Peiwu Zhang were supported by the Research Grants Council of Hong Kong (GRF Projects 711110 and 711309E). We would like to thank the anonymous reviewers for their insightful comments.

REFERENCES