HKU CS Tech Report TR-2012-10

Voronoi-based Nearest Neighbor Search for Multi-Dimensional Uncertain Databases

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

#*The University of Hong Kong, Pokfulam Road, Hong Kong*
{*pwzhang* 1, *ckcheng* 2, *nikos* 3,*ytang* 6}@*cs.hku.hk*
**Ludwig-Maximilians-Universität München, Munich, Germany*
*{*renz* 4, *zuefle* 5, *emrich* 7}@*dbs.ifi.lmu.de*

***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, such that for any point $p \in R$, 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 the 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 a Voronoi diagram is a data structure that 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.

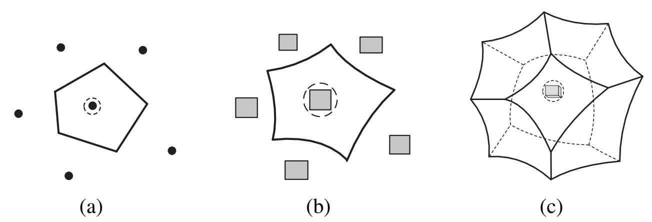


Fig. 1. Illustrating the PV-cell of (circled) object o, whose uncertainty region is: (a) point; (b) 2D rectangle; and (c) 3D rectangle (where the PV-cell is composed of curved surfaces).

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 $D \subseteq \Re^d$. 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 $u(o) \subseteq D$ which we call *uncertainty region* of o. Specifically, u(o) minimally bounds all possible values of o's attributes $o.a$. Let us 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: The **Possible Voronoi Cell** (PV-cell) of an uncertain object o, denoted by $\mathcal{V}(o)$, is a d-dimensional region R, such that for any point $p \in R$, 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, $\mathcal{V}(o)$ reduces to a Voronoi cell of o, as illustrated in Figure 1(a). This example shows the (2D) uncertainty regions of locations of five objects. Given an uncertain database S and an object $o \in S$, a PV-cell is a region R, where for any point $p \in R$, o has a non-zero chance to be the closest to p. In Figure 1(b), the PV-cell of o (in dotted circle) is a region bounded by solid curve segments. To see whether o is the closest to 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 o_1 and o_2. 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 need to compute the exact PV-cell. The main idea is to approximate the PV-cell iteratively in an analytical 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 a UBR.

Based on the UBRs computed by SE, we develop the *PV-index* to enable PNNQ evaluation. This space-partitioning structure 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

Our work is related to the Voronoi diagram and the PNNQ, as detailed below.

The **Voronoi Diagram** is a partitioning of a multi-dimensional space that contains point data. Each partition of the diagram, called the *Voronoi cell*, is associated with a point p, such that any point inside p's cell has p as its nearest neighbor (*NN* in short) [1]. Figure 1(b) illustrates a Voronoi cell.

The Voronoi diagram is primarily used to answer NN queries over points [1], [18], [19]; the NN of a query point q is the one whose corresponding Voronoi cell contains q. In this context, the Voronoi diagram has been used to support nearest neighbor queries in geo-spatial applications [19], [20], in spatial data streams [21] and, recently, in distributed spatial environments [22] as well as in spatial network environments [23], [24]. Furthermore, the Voronoi diagram has been used in wireless database services [25], [26], location-based services [27], [28], and virus spread analysis [29]. The PV-cell of object o defines the area where any point inside it may have o as its NN. Hence, the PV-cell is a generalized version of the Voronoi cell.

Constructing a Voronoi diagram for multi-dimensional points is often costly. Hence, researchers studied its approximate form. In [30], the Voronoi diagram was approximated as a disjoint set of convex polygonal objects. In [31], 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. We show that for a PV-cell, there does not exist any efficient solution for finding its MBR. We then study how to compute the UBR, which is only slightly larger than the MBR. These UBRs, as we will explain, support PNNQ evaluation.

Relatively few works studied the use of the Voronoi diagram for uncertain data. In [32], the Voronoi diagram is used to support uncertain data clustering. In [33], [34], the Voronoi diagram is employed to find out all uncertain objects that must be the nearest neighbor of a query point. 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 [35], 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 [35] 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 [35] 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 costs 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], [36], efficient methods were proposed to generate answer objects' probability bounds without performing expensive integration operations.

Symbol	Meaning
S	A database of $\|S\|$ uncertain objects
d	The no. of dimensions of S
D	The domain of S
o	An uncertain object of S
q	A query point in D
$dist_{max}(o,p)$	max distance of o from point p
$dist_{min}(o,p)$	min distance of o from point p
$u(o)$	The uncertain region of $o.a$
$\mathcal{V}(o)$	The PV-cell of o
$\mathcal{M}(o)$	The MBR of $\mathcal{V}(o)$
$\mathcal{B}(o)$	The UBR of $\mathcal{V}(o)$

TABLE I
SYMBOLS AND MEANINGS USED IN THIS PAPER.

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. THE POSSIBLE VORONOI CELL

As we have discussed, PV-cells can be used to evaluate Step 1 of PNNQ. Let us now examine them in more detail. In the following we first discuss several important properties of a PV-cell. Then, in Section III-B, we show how to approximate a PV-cell. Table I shows the symbols used in our paper.

A. PV-cell: Basic properties

1. Shape of $\mathcal{V}(o)$. Given a point $p \in \Re^d$, let $dist_{max}(o,p)$ ($dist_{min}(o,p)$) 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, $H_{o',o}$:

$$H_{o',o} = \{p \in \Re^d | dist_{max}(o',p) = dist_{min}(o,p)\} \quad (1)$$

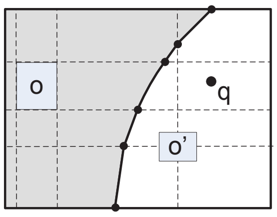


Fig. 2. Illustrating o, o', $H_{o',o}$ (in solid line), and the PV-cell of o (shaded).

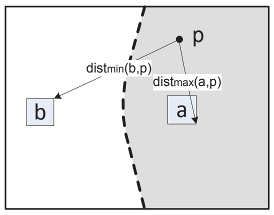


Fig. 3. $dom(a, b)$ (grey) and $\neg dom(a, b)$, separated by $H_{a,b}$ (dotted line).

Figure 2 illustrates o, o', and $H_{o',o}$, bounded within a 2D domain. Equation 1 cuts the domain space into two *half-spaces*. If a point $q \in \Re^d$ is located in the half-space containing o', then since $dist_{max}(o', p) < dist_{min}(o, p)$, o must not be the nearest neighbor of q. The corresponding PV-cell of o includes the boundaries of the domain, as well as a portion of $H(o', o)$. The PV-cell of o is shaded in Figure 2.

Computing $H_{o',o}$ 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 [37]. 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(a)). 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 UV-cell, and thus the time complexity of any algorithm using UV-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 1d-surfaces. Further information on the complexity of higher dimension Voronoi cells can be found in [38].

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., [39]). The authors in [31] 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 $\mathcal{M}(o)$ be the MBR of $\mathcal{V}(o)$. There does not exist any polynomial-time algorithm for finding $\mathcal{M}(o)$.

Proof: **(Sketch)** Since the surfaces of $\mathcal{V}(o)$ are concave in shape, $\mathcal{M}(o)$ 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 $\mathcal{M}(o)$ as a *convex optimization problem*. In particular, any dimension of $\mathcal{M}(o)$ 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 [40], this kind of problems does not have any polynomial-time solution. Correspondingly, no efficient solution exists for $\mathcal{M}(o)$. ■

The detailed proof of the above lemma can be found in Appendix A. We conclude that it is impractical to find $\mathcal{M}(o)$. 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 $\mathcal{B}(o)$, is a d-dimensional rectangle that completely contains $\mathcal{V}(o)$.

A trivial $\mathcal{B}(o)$ is the domain space D, whereas a UBR that tightly contains $\mathcal{V}(o)$ is essentially $\mathcal{M}(o)$. Our goal is to develop an efficient algorithm for finding a $\mathcal{B}(o)$, which is only a bit looser compared to the corresponding $\mathcal{M}(o)$. 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 $\mathcal{B}(o)$. Section VI then explains how to use UBRs to evaluate a PNNQ.

IV. PV-cell and Dominated Regions

Our main idea of finding $\mathcal{B}(o)$ 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 Appendix B.

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 $dom(a, b)$, is a subset of D, such that:

$$dom(a, b) = \{p \in D | dist_{max}(a, p) < dist_{min}(b, p)\}.$$

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

$$\neg dom(a, b) = \{p \in D | dist_{max}(a, p) \geq dist_{min}(b, p)\}.$$

Figure 3 illustrates these two regions, which are separated by hyperplane $H_{a,b}$ (Equation 1). When point p is inside $dom(a, b)$, according to Definition 3, a is always closer to

p than b. If $p \in \neg dom(a,b)$, then b may be closer to p than a.

Lemma 2: $dom(a,b) = \emptyset$ if and only if $u(a)$ intersects $u(b)$.

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

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.,

$$I(A,o) = \bigcap_{\forall a \in A} \neg dom(a,o).$$

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.,

$$U(A,o) = \bigcup_{\forall a \in A} dom(a,o).$$

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. ■

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

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

We also define the **candidate V-set** of $\mathcal{V}(o)$, or simply **C-set**:

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

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

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

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

Lemma 6: $\mathcal{V}(o)$ is a connected region.

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

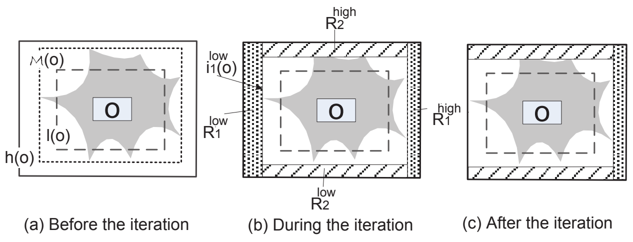


Fig. 4. Illustrating an iteration of SE (x-dimension, *low* direction).

V. Generating a UBR

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

The main idea of SE is to estimate $\mathcal{M}(o)$ with the aid of a pair of d-dimensional rectangles: the *lower bound* $l(o)$, which is enclosed by $\mathcal{M}(o)$; and the *upper bound* $h(o)$, which contains $\mathcal{M}(o)$. In other words, $\mathcal{M}(o)$ is sandwiched between $l(o)$ and $h(o)$. The algorithm iteratively adjusts the size of these rectangles until $\mathcal{M}(o)$ 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 $\mathcal{M}(o)$, and;
- **Expand:** Increase the size of $l(o)$, by including regions that are assured to be inside $\mathcal{M}(o)$.

When the distance between $l(o)$ and $h(o)$ is smaller than some threshold value Δ, SE outputs $h(o)$ as the UBR of o. Figure 4(a) illustrates $\mathcal{V}(o)$ (in grey), $\mathcal{M}(o)$, $l(o)$, 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 $\mathcal{V}(o)$. 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, $\mathcal{V}(o) = I(S,o)$. 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 $u(o) \subseteq \mathcal{V}(o)$ (Lemma 5), and $\mathcal{V}(o) \subseteq \mathcal{M}(o)$. 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 $\mathcal{V}(o)$.

We now discuss Steps 4-12 in more detail. Let $\rho = \{$*low,high*$\}$ be the "direction" of object o along the j-th dimension (where $j = 1, \ldots, d$). For example, in Figure 4(a), $\rho =$ *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 ρ of dimension j. Step 7 computes $i_j^{\rho}(o)$, which is a hyperplane in the middle of $h(o)$ and $l(o)$, in direction ρ along the j-th dimension. Figure 4(b) illustrates

Algorithm 1: The SE algorithm

```
input  : Database $S$, object $o$
output: UBR of $o$ (i.e., $\mathcal{B}(o)$)
1  begin
2      $\mathcal{C}_{set}(o) \leftarrow$ chooseCSet$(o, S)$
3      $h(o) \leftarrow D$, $l(o) \leftarrow u(o)$
4      while $|h(o) - l(o)|_d \geq \Delta$ do
5          for each dimension $j = 1, \ldots, d$ do
6              for $\rho \in \{low, high\}$ do
7                  Let $i_j^\rho(o)$ be the middle plane between
                   $h(o)$ and $l(o)$ in direction $\rho$ of $j$-th
                   dimension
8                  Let $R_j^\rho$ be the region between $i_j^\rho(o)$ and
                   the plane of $h(o)$
9                  if $R_j^\rho \cap I(\mathcal{C}_{set}(o), o) = \emptyset$ then
10                     Remove $R_j^\rho$ from $h(o)$
11                 else
12                     Expand $l(o)$ to position of $i_j^\rho(o)$
13     return $h(o)$
```

$i_1^{low}(o)$, where dimensions 1 and 2 denote the x- and the y-axes respectively. In Step 8, we consider the region between $i_j^\rho(o)$ and $h(o)$, denoted by R_j^ρ. Figure 4(b) demonstrates these regions. In Step 9, we test whether R_j^ρ overlaps $I(\mathcal{C}_{set}(o), o)$. According to Definition 8, $\mathcal{V}(o)$ is bounded by $I(\mathcal{C}_{set}(o), o)$. Hence, if R_j^ρ does not touch $I(\mathcal{C}_{set}(o), o)$, it must also not touch $\mathcal{V}(o)$. This R_j^ρ cannot be part of $\mathcal{M}(o)$, and can be safely removed from $h(o)$ (Step 10). Otherwise, we expand $l(o)$ in direction ρ, dimension j, up to the position of $i_j^\rho(o)$ (Step 11). Figure 4(c) shows that $h(o)$ is shrunk with the removal of R_1^{low}. The shrinking-and-expanding process is repeated until the maximum distance between the boundaries $h(o)$ and $l(o)$, denoted by $|h(o) - l(o)|_d$, is less than Δ, as indicated in Step 4. [1]

Discussions. Observe that in each iteration, the distance between $h(o)$ and $l(o)$ (in one direction) is halved. Let $|D|_{max}$ be the maximum of the lengths of domain D projected to all dimensions. Then, the number of iterations executed for each direction is at most $\log(|D|_{max}/\Delta)$, and the total number of iterations required by SE is $\log(|D|_{max}/\Delta) \cdot 2d$. Thus, $l(o)$ and $h(o)$ converge to $\mathcal{M}(o)$ quickly. When $\Delta = 0$, $h(o) = l(o)$, and $h(o)$ becomes $\mathcal{M}(o)$, the MBR of $\mathcal{V}(o)$. In our experiments, by using a small Δ, the UBR returned by SE is only a bit larger than its corresponding MBR.

However, SE is still not very efficient, because of Step 9:

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

[1]Specifically, $|h(o) - l(o)|_d$ is the maximum distance between $h(o)$ and $l(o)$, among all dimensions.

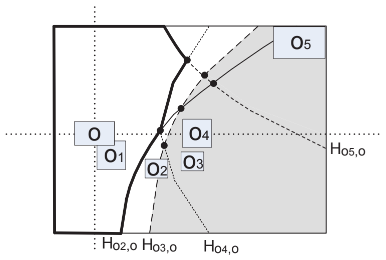


Fig. 5. Illustrating how different objects affect the PV-cell of o.

- **Problem 2:** Evaluating $R_j^\rho \cap I(\mathcal{C}_{set}(o), o)$ 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 Appendix C.

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

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, $I(\mathcal{C}_{set}(o), o)$ is simply $\neg dom(o', o)$, and Step 9 of SE can be efficiently executed. Unfortunately, the UBR of o returned by SE can be much larger than its corresponding PV-cell, as illustrated by he following example.

Example. *In Figure 5, $S = \{o, o_1, \ldots, o_5\}$. The boundary of $\mathcal{V}(o)$ is drawn in bold lines, and one of the V-sets of $\mathcal{V}(o)$ is $\{o_2, o_4, o_5\}$. Notice that o_1 and o_3 are not in this $\mathcal{V}_{set}(o)$. Recall that $\mathcal{V}(o) = I(S, o)$ (Lemma 4), or the intersection of $\neg dom(o', o)$, for every $o' \in S$. Since $u(o)$ overlaps $u(o_1)$, by Lemma 2, $\neg dom(o_1, o)$ is D. Hence, o_1 does not affect the shape of $\mathcal{V}(o)$, and o_1 can be excluded from $\mathcal{V}_{set}(o)$. For o_3, notice that $dom(o_3, o)$ (the grey region) does not intersect $\mathcal{V}(o)$. Consequently, $\neg dom(o_3, o)$ does not affect $\mathcal{V}(o)$, and o_3 can also be pruned from $\mathcal{V}_{set}(o)$.*

Now, suppose that chooseCSet *returns o_1. Then, $I(\mathcal{C}_{set}(o), o)$ becomes $I(\{o_1\}, o)$, or just D. As a result, in Step 9 of* SE*, $R_j^\rho \cap D$ 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* SE *may not be desirable, since it may be much larger than $\mathcal{V}(o)$ or $\mathcal{M}(o)$. With a similar argument, when $\mathcal{C}_{set}(o) = \{o_3\}$, $h(o)$ cannot be shrunk to tightly bound $\mathcal{M}(o)$.*

To ensure that SE returns a small MBR, a thoughtful design of chooseCSet is important. Notice that if $\mathcal{C}_{set}(o) =$

$\mathcal{V}_{set}(o)$, SE can attain the highest effectiveness. This is because the condition tested in Step 9 becomes $R_j^\rho \cap \mathcal{V}(o)$ (Lemma 4). If $\Delta = 0$, the UBR returned is *exactly* $\mathcal{M}(o)$. Again, S is one of the V-sets of o. To solve Problem 1, however, it is desirable to obtain a $\mathcal{V}_{set}(o)$ with the minimal size. For example, $\{o_2, o_4, o_5\}$ is the *minimum V-set* of o, denoted by $\mathcal{V}_{set}^{min}(o)$, 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 $\mathcal{V}(o)$. This is similar to the computation of $\mathcal{V}(o)$, 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 $\mathcal{V}(o)$, 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 $\mathcal{C}_{set}(o) = \{o_1, o_2\}$. 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 $\mathcal{V}_{set}^{min}(o)$ than b. In Figure 5, o_2 is closer to o than o_3, and so $\mathcal{V}_{set}^{min}(o)$ 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 $\mathcal{V}_{set}^{min}(o)$. In the previous example, although o_5 is a member of $\mathcal{V}_{set}^{min}(o)$, 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 $\mathcal{V}_{set}^{min}(o)$.
- 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 $\mathcal{V}_{set}^{min}(o)$, 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, $o_5 \in \mathcal{V}_{set}^{min}(o)$.
- FS does not discard objects whose uncertainty regions overlap $u(o)$. As explained before, these objects should not be included in a C-set, since it does not affect $\mathcal{V}(o)$ 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, $\mathcal{C}_{set}(o)$ 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 $\mathcal{V}_{set}^{min}(o)$ 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 [41]. 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 $k_{partition}$, or when k_{global} nearest neighbors of o are examined. Suppose that $k_{partition} = 2$ and $k_{global} = 10$ in Figure 5. Then, o_4 and o_5 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 Appendix E.

One benefit of IS over FS is that it does not need to set the value of k anymore. Although IS needs to determine $k_{partition}$ and k_{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 $k_{partition}$.

Remarks. Although IS and FS may not return $\mathcal{V}_{set}^{min}(o)$, they are still correct, since according to Lemma 7, any subset of S can be $\mathcal{C}_{set}$. 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 $k = k_{global}$. 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^ρ and $I(\mathcal{C}_{set}(o), o)$ 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 $R_j^\rho \cap I(\mathcal{C}_{set}(o), o)$ 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 $dom(A, R)$. In Figure 6(a), this technique allows to decide that R is completely contained in the dominating region $dom(a, b)$.

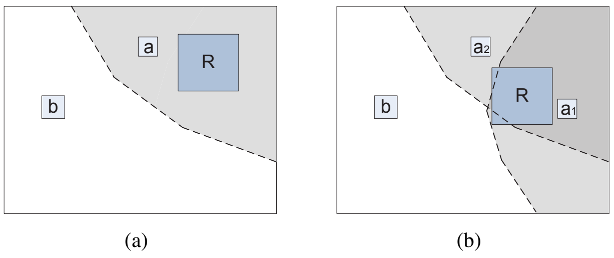


Fig. 6. Illustrating (a) spatial domination and (b) domination count estimation.

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 $dom(a_1, b)$ nor in $dom(a_2, b)$. 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^ρ does not intersect $I(\mathcal{C}_{set}(o), o)$ 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 $\mathcal{C}_{set}(o)$, and the spatial domination test is linear in the dimensionality of the dataset, the runtime of the intersection test is in O($|part(R)| \cdot |\mathcal{C}_{set}(o)| \cdot d$), where $|part(R)|$ denotes the number of partitions of R.

Efficiency of SE**.** As discussed in Section V-A, if chooseCSet is implemented by IS or FS, the cost of Step 2 is $O(|S|)$. For Steps 4-12, a maximum of $O(\log(|D|_{max}/\Delta) \cdot d)$ iterations is needed. In each iteration, the most costly task is the condition testing in Step 9. If the non-dominated region intersection test is used, the cost of SE is $O(\log(|D|_{max}/\Delta) \cdot |part(R)| \cdot |\mathcal{C}_{set}(o)| \cdot d^2)$. Notice that SE does not compute any intersection on $\mathcal{V}(o)$. Finally, if the uncertainty region of o (i.e., $u(o)$) is not rectangular, we can represent $u(o)$ by its minimum bounding rectangle. The size of the PV-cell for this rectangle will not be less that of $\mathcal{V}(o)$. Thus, the UBR returned by SE still contains $\mathcal{V}(o)$.

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.

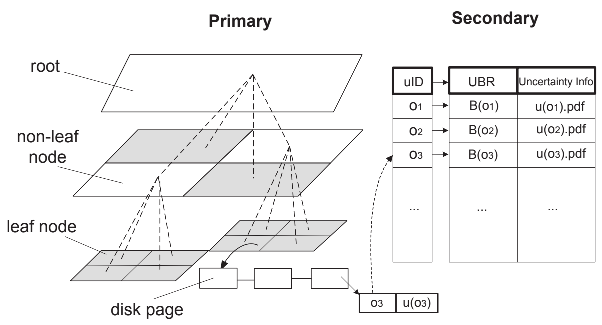


Fig. 7. Illustrating the PV-index (2D).

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 [42], while the secondary index is an extensible hash table [43]. 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. [2]

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

[2]Each 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.

on the method in [8]; in practice, any solution mentioned in Section II can be used. Appendix F discusses this in detail. [3]

Index Construction . The PV-index is created by inserting UBRs to 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 $\mathcal{B}(o)$ of every object $o \in S$ is then inserted to the index as follows:

1) Perform a range search on the PV-index, using $\mathcal{B}(o)$, and locate the leaf nodes whose regions overlap $\mathcal{B}(o)$.
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 $(\mathcal{B}(o), u(o).\mathit{pdf})$ to the secondary index.

Since both the region of a node and $\mathcal{B}(o)$ 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 $\lfloor M/2^{d+2} \rfloor \cdot (1 + 2^d)$ nodes. The construction cost of the index is $O((M + \mathit{cost}_{\texttt{SE}}) \cdot |S|)$, where $\mathit{cost}_{\texttt{SE}}$ denotes the time complexity of SE. Evaluating Step 1 of PNNQ requires a cost of $O(\log\lfloor M/2^{d+2} \rfloor + |S|/K)$. For details, please refer to Appendix G.

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 $\mathcal{V}(X, o)$, $\mathcal{M}(X, o)$ and $\mathcal{B}(X, o)$ 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 $o \in S \wedge o \neq o'$) is *affected*, if $\mathcal{V}(S, o)$ and $\mathcal{V}(S', o)$ are different upon a database update. Lemma 8 lists the conditions for o to be not affected:

Lemma 8: Object o is not affected if:

1) $\mathcal{V}(S, o) \cap \mathcal{V}(S, o') = \phi$ (for **deletion of** o'); or
2) $\mathcal{V}(S, o) \cap \mathcal{V}(S', o') = \phi$ (for **insertion of** o'); or
3) $u(o) \cap u(o') \neq \phi$

[3]Alternatively, 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.

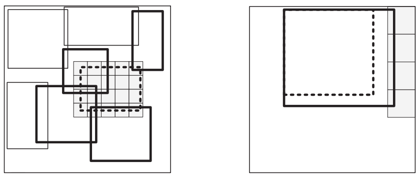

(a) (b)

Fig. 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).

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 $\mathcal{V}(S, o)$ does not overlap $\mathcal{V}(S', o')$, the PV-cell of o does not change. As shown in Condition (3), if the uncertainty regions of o and o' do not intersect, than again o is unaffected. These conditions can be proved by using the domination results in Section IV, as detailed in Appendix D.

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 Appendix D. Lemma 9 allows us to use the old UBR of o, i.e., $\mathcal{B}(S, o)$ to derive the new one, i.e., $\mathcal{B}(S', o)$. 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) Retrieve $\mathcal{B}(S, o')$ from the secondary index, using the ID of o'.
2) Identify the set $A \subseteq S$ of objects that may be affected.
3) Compute the new UBRs of objects in A.
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 $\mathcal{B}(S, o')$, and obtain leaf nodes whose space in D overlap $\mathcal{B}(S, o')$. 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 $\mathcal{B}(S, o')$. 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 $u(o) \cap u(o') \neq \phi$. Next, from the secondary index, we retrieve their UBRs, and discard any object o where $\mathcal{B}(S, o) \cap \mathcal{B}(S, o') = \phi$, or correspondingly, $\mathcal{V}(S, o) \cap \mathcal{V}(S, o') = \phi$. 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 $o \in A$, $\mathcal{V}(S,o)$, as well as $\mathcal{B}(S,o)$, may be changed after the update. To obtain $\mathcal{B}(S',o)$, 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 $\mathcal{B}(S,o)$, which was stored in the secondary index. As discussed in Lemma 9, $\mathcal{V}(S',o)$ cannot be smaller than $\mathcal{V}(S,o)$; hence, $\mathcal{B}(S',o)$ may also be bigger than $\mathcal{B}(S,o)$. We can thus use $\mathcal{B}(S,o)$ as $l(o)$. [4] Since $\mathcal{B}(S,o)$ 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 $\mathcal{B}(S,o')$ from the primary index. This is done by locating the leaf nodes whose regions overlap $\mathcal{B}(S,o')$ 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 $\mathcal{B}(S,o)$ and $\mathcal{B}(S',o)$ 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 $\mathcal{B}(S,o)$ is covered by $\mathcal{B}(S',o)$, 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 $\mathcal{B}(S',o')$ by running SE on S'.

[Step 2] Identify the set A of affected objects, by issuing a range query $\mathcal{B}(S',o')$ on the primary index. For any $o \in A$, remove o from A if $u(o) \cap u(o') \neq \phi$ or $\mathcal{B}(S,o) \cap \mathcal{B}(S',o') = \phi$; 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 $\mathcal{B}(S,o)$, instead of D. This is correct, because Lemma 9 states that $\mathcal{V}(S',o)$ cannot be larger than $\mathcal{V}(S,o)$; also $\mathcal{V}(S,o)$ is completely inside $\mathcal{B}(S,o)$. Since now $\mathcal{B}(S,o)$ 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 $\mathcal{B}(S,o)$ and $\mathcal{B}(S',o)$ 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 $O((M + cost_{\text{SE}}) \cdot |S|)$. Appendix G 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 $\mathcal{B}(S,o)$ is larger than $\mathcal{M}(S',o)$, SE is still correct, since $h(o) = D$ is always an upper bound of $\mathcal{M}(S',o)$.

[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.

parameter	values (synthetic)	values (real)
$\lvert S\rvert$	20k, **40k**, 60k, 80k, 100k	30k, 36k, 20k
d	2, **3**, 4, 5	2, 3
$\lvert u(o)\rvert$	20, 40, **60**, 80, 100	N/A
Δ	0.1, 0.5, **1**, 10-1000	1
m_{max}	2-5, **10**, 20, 40	10
k	20, 40, 100, **200**, 400	200
$k_{partition}$	2, 5, **10**, 20, 50	10
k_{global}	**200**	200

TABLE II
PARAMETERS AND THEIR DEFAULT VALUES (IN BOLD).

VII. Experimental Results

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 [13], [14], 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 II list the values of parameters used in our experiments.

For the three real datasets used, two of them, called *roads* (30k) and *rrlines* (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 [15]. 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 [44] 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.

[6] http://www.rtreeportal.org/software/SpatialDataGenerator.zip

[7] http://www.rtreeportal.org

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

For *PV-index*, the default values of k, $k_{partition}$, and k_{global} are 200, 10, and 200 respectively. The IS strategy is used to implement `chooseCSet` 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 4kb 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 Appendix F.

In the following results, each data point reported is an average of 50 runs. Unless stated otherwise, the discussion below 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.

[9]In our experiments, *R-tree* needs more main memory to store the non-leaf nodes than both *UV-index* and *PV-index*.

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

[11]Notice that fast solutions such as [11], [36] 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.

Figure 9(e) also reveals that for all the indexes tested, T_q does not increase with d. When d increases, the volume of D also increases, and objects in D tend to be more separated. Since fewer objects qualify for the PNNQ, the amount of time spent on the *PC*, i.e., T_{PC}, drops with d. However, the amount of time spent on *OR*, i.e., T_{OR}, increases with d (Figure 9(f)). As the drop in T_{PC} is higher than the rise in T_{OR}, the performance at $d = 2$ can be worse than that of other higher dimensions.

Real datasets. As shown in Figure 9(h), for 2D datasets (*roads* and *rrlines*, 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.

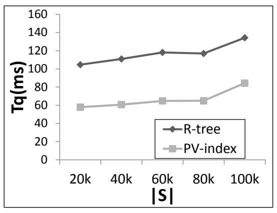
(a) Parameter Testing. Figures 10(a)-(c) shows the effect of Δ, k, and $k_{partition}$, on the query performance of the PV-index. Observe that T_q is quite stable, except when the parameter values are extremely high or low (e.g., $\Delta > 500$). It is thus not very hard to choose parameters to attain a high query performance. The indexes created by using FS and IS strategies also yield a similar performance.

We then study the effect of these parameters on the index construction time T_c. Figure 11(a) shows that T_c drops with the increase of Δ. This is because `SE` needs fewer iterations to compute the UBR. We further found that T_c increases with k and $k_{partition}$; the results are skipped due to space limitation.

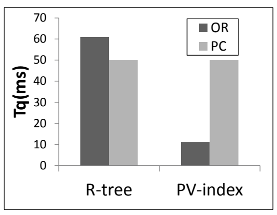
(b) Index Construction. We next study how FS and IS, used in `chooseCSet`, affect T_c. We compare these methods with a naive solution, called `ALL`, which instructs `chooseCSet` to return S as the C-set. Figure 11(c) shows that *ALL* is extremely inefficient; when $|S| = 20k$, the construction time is 103 hours. However, FS and IS needs 10 minutes or less to complete. Figures 11(d) and 11(e) 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 11(f), 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 11(g), IS is faster than FS for all the datasets tested. Figure 11(h) 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.

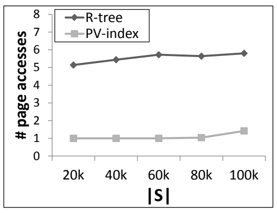
(c) Index Update. We compare the incremental update algorithm (*Inc*) and the solution that *rebuilds* the index. For dele-



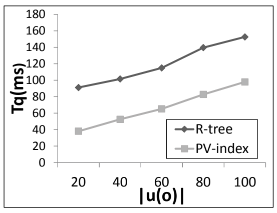
(a) $T_q(ms)$ vs. $|S|$.



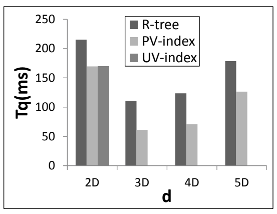
(b) *OR* and *PC*.



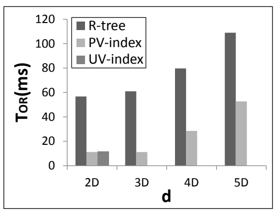
(c) $T_q(I/O)$ vs. $|S|$.



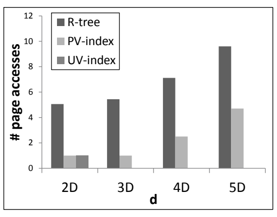
(d) $T_q(ms)$ vs. $|u(o)|$.



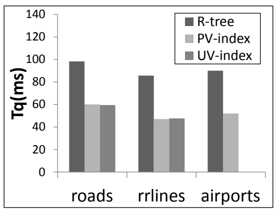
(e) T_q vs. d.



(f) T_{OR} vs. d.

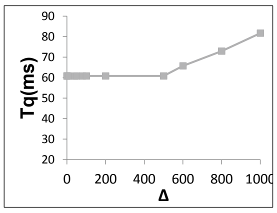


(g) $T_q(I/O)$ vs. d.

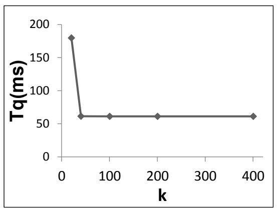


(h) Real DBs.

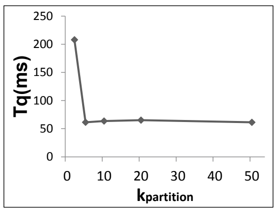
Fig. 9. PNNQ Performance.



(a) Effect of Δ.

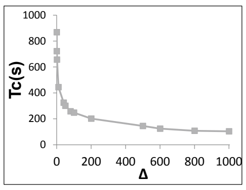


(b) Effect of k (FS).

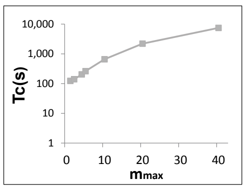


(c) Effect of $k_{partition}$ (IS).

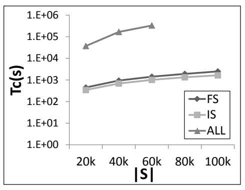
Fig. 10. Sensitivity of T_q to PV-index parameters.



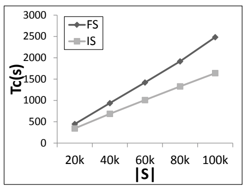
(a) Effect of Δ.



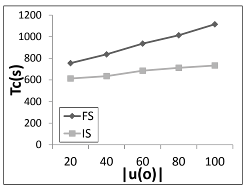
(b) Effect of m_{max}.



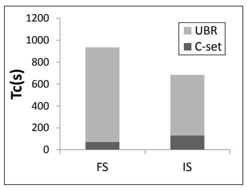
(c) $T_c(s)$ vs. $|S|$.



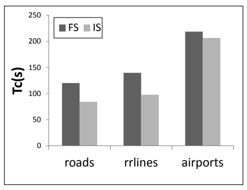
(d) $T_c(s)$ vs. $|S|$.



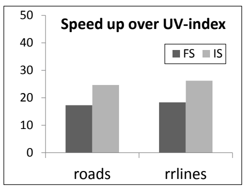
(e) $T_c(s)$ vs. $|u(o)|$.



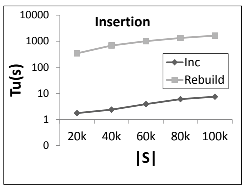
(f) Analysis of SE.



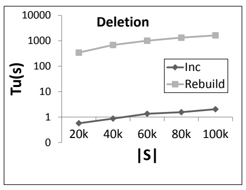
(g) Real DBs.



(h) Speedup on 2D Real DBs.



(i) Insertion



(j) Deletion

Fig. 11. Construction and Update Performance of the PV-Index.

tion, we randomly remove $1K$ objects from S. For insertion, we use the database where the $1K$ objects have been removed, and re-insert all these objects to it. We measure the average time T_u to handle insertion/deletion per object. Figure 11(i) 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 11(j) also shows that *Inc* is much faster than *Rebuild*. We remark 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.

References

[1] F. Aurenhammer, "Voronoi diagrams: a survey of a fundamental geometric data structure," *ACM Computing Surveys (CSUR)*, 1991.

[2] C. Aggarwal, "On unifying privacy and uncertain data models," in *ICDE*, 2008.

[3] A. Deshpande et al., "Model-based approximate querying in sensor networks," *VLDBJ*, 2005.

[4] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage," *Technical Report*, 2004.

[5] S. Singh et al., "Orion 2.0: native support for uncertain data," in *SIGMOD*, 2008.

[6] J. Boulos et al., "Mystiq: a system for finding more answers by using probabilities," in *SIGMOD*, 2005.

[7] J. Huang, L. Antova, C. Koch, and D. Olteanu, "Maybms: a probabilistic database management system," in *SIGMOD*, 2009.

[8] R. Cheng, D. Kalashnikov, and S. Prabhakar, "Querying imprecise data in moving object environments," *TKDE*, 2004.

[9] R. Cheng et al., "UV-diagram: A Voronoi diagram for uncertain data," in *ICDE*, 2010.

[10] G. Beskales, M. Soliman, and I. IIyas, "Efficient search for the top-k probable nearest neighbors in uncertain databases," *VLDB*, 2008.

[11] R. Cheng et al., "Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data," in *ICDE*, 2008.

[12] X. Lian and L. Chen, "Probabilistic group nearest neighbor queries in uncertain databases," *TKDE*, 2008.

[13] M.A. Cheema et al., "Probabilistic reverse nearest neighbor queries on uncertain data," *TKDE*, 2010.

[14] T. Bernecker et al., "Efficient probabilistic reverse nearest neighbor query processing on uncertain data," *VLDB*, 2011.

[15] B. Parkinson, "GPS error analysis," *Global Positioning System: Theory and applications.*, vol. 1, pp. 469–483, 1996.

[16] T. Emrich et al., "Incremental reverse nearest neighbor ranking in vector spaces," *Advances in Spatial and Temporal Databases*, 2009.

[17] ——, "Boosting spatial pruning: on optimal pruning of mbrs," in *SIGMOD*, 2010.

[18] C. Shahabi and M. Sharifzadeh, "Voronoi diagrams for query processing," in *Encyclopedia of GIS*, 2008, pp. 1235–1240.

[19] M. Sharifzadeh et al., "Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries," *PVLDB*, 2010.

[20] L. Hu, W.-S. Ku, S. Bakiras, and S. C., "Verifying spatial queries using voronoi neighbors," in *SIGSPATIAL*, 2010.

[21] M. Sharifzadeh and C. Shahabi, "Approximate voronoi cell computation on spatial data streams," *VLDB J.*, vol. 18, no. 1, pp. 57–75, 2009.

[22] A. Akdogan et al., "Voronoi-based geospatial query processing with mapreduce," in *The 2nd International Conference on CloudCom*, 2010.

[23] M. Kolahdouzan and C. Shahabi, "Voronoi-based k nearest neighbor search for spatial network databases," in *VLDB*, 2004.

[24] U. Demiryurek and C. Shahabi, "Indexing network voronoi diagrams," in *DASFAA 2012*, 2012.

[25] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. Lee, "Location-based spatial queries," in *SIGMOD*, 2003.

[26] B. Zheng et al., "Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services," *VLDBJ*, 2006.

[27] J. Xu et al., "Energy efficient index for querying location-dependent data in mobile broadcast environments," in *ICDE*, 2003.

[28] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik, "The v*-diagram: a query-dependent approach to moving knn queries," *VLDB*, 2008.

[29] P. Wang et al, "Understanding the spreading patterns of mobile phone viruses," *Science*, vol. 324, no. 5930, p. 1071, 2009.

[30] J. Vleugels and M. Overmars, "Approximating voronoi diagrams of convex sites in any dimension," *International Journal of Computational Geometry and Applications*, 1998.

[31] S. Berchtold, B. Ertl, D. Keim, H. Kriegel, and T. Seidl, "Fast nearest neighbor search in high-dimensional space," in *ICDE*, 1998.

[32] B. Kao, S. Lee, D. Cheung, W. Ho, and K. Chan, "Clustering uncertain data using voronoi diagrams," in *ICDM*, 2008.

[33] W. Evans et al, "Guaranteed voronoi diagrams of uncertain sites," in *20th Canadian Conference on Computational Geometry*, 2008.

[34] M. Jooyandeh, A. Mohades, and M. Mirzakhah, "Uncertain voronoi diagram," *Information processing letters*, 2009.

[35] M. Ali et al., "Probabilistic voronoi diagrams for probabilistic moving nearest neighbor queries," *Data and Knowledge Engineering*, 2012.

[36] T. Bernecker et al., "A novel probabilistic pruning approach to speed up similarity queries in uncertain databases," in *ICDE*, 2011.

[37] T. Emrich et al., "Constrained reverse nearest neighbor search on mobile objects," in *SIGSPATIAL*, 2009.

[38] R. Seidel, "The complexity of voronoi diagrams in higher dimensions," in *the 20th Annual Allerton Conference on CCC*. IEEE, 1982.

[39] T. Brinkhoff, H. Kriegel, and R. Schneider, "Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems," in *ICDE*, 1993.

[40] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge University Press, 2004.

[41] G. Hjaltason and H. Samet, "Distance browsing in spatial databases," *TODS*, 1999.

[42] H. Samet, *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[43] A. Rathi et al., "Performance comparison of extensible hashing and linear hashing techniques," in *Proc. ACM SIGSmall/PC Symposium on Small Systems*, 1990.

[44] N. Beckmann et al., "The r*-tree: an efficient and robust access method for points and rectangles," *SIGMOD*, 1990.

Appendix

A. Section III

[Lemma 1] Since the surfaces of $\mathcal{V}(o)$ are concave in shape, $\mathcal{M}(o)$ is determined by the vertices of $\mathcal{V}(o)$. Specifically, in each dimension, the lower (upper) bounds of $\mathcal{M}(o)$ is the smallest (largest) coordinate value among the vertices of $\mathcal{V}(o)$. Unless we can find the exact form of $\mathcal{V}(o)$ (which is extremely complex), the vertices of $\mathcal{V}(o)$ are not readily known.

We next show that the problem of finding $\mathcal{M}(o)$ can be classified as a *convex optimization problem*. Let low_j and up_j be the lower and upper bounds of $\mathcal{M}(o)$ in dimension j ($1 \leq j \leq d$) respectively. For each j, obtaining $\{low_j, up_j\}$ involves solving two equations:

$$low_j = \min\{p_j | p \in \mathcal{V}(o)\} \tag{2}$$

$$up_j = \max\{p_j | p \in \mathcal{V}(o)\} \tag{3}$$

where p_j is the coordinate of a point p in dimension j.

The problem of solving Equation 2 can be viewed as an optimization problem, whose feasible region (i.e., solution space) must be $\mathcal{V}(o)$, since $\mathcal{M}(o)$ is determined by the vertices of $\mathcal{V}(o)$. However, since $\mathcal{V}(o)$ is not a convex polygon, the feasible region of this problem is neither convex. According to [40], this kind of optimization problem has no efficient solution.

We can similarly conclude that there is no efficient solution for Equation 3. Since finding $\mathcal{M}(o)$ involves solving Equations 2 and 3 over d dimensions, it is not possible to derive $\mathcal{M}(o)$ efficiently.

B. Section IV

[Lemma 2] consists of the *if* and *only if* parts:

- **(If)** Since $u(a)$ overlaps $u(b)$, there exists a point p' such that $p' \in u(a)$ and $p' \in u(b)$. For any point $p \in D$, $dist_{min}(b,p) \leq dist(p',p)$, and $dist(p',p) \leq dist_{max}(a,p)$. Therefore, $dist_{max}(a,p) \geq dist_{min}(b,p)$. According to Definition 3, $dom(a,b) = \emptyset$.
- **(Only if)** Since $dom(a,b) = \emptyset$, using Definition 4, $\neg dom(a,b) = D$. This means for every $p \in D$, $dist_{max}(a,p) \geq dist_{min}(b,p)$. Suppose by contrary that $u(a)$ does not overlap $u(b)$. Then, we can find a point $p \in u(a)$ such that $dist_{max}(a,p) \geq dist_{min}(b,p)$ does not hold, resulting in a contradiction.

[Lemma 3] Notice that $D - I(A,o) = D - \bigcap_{\forall a \in A} \neg dom(A,o)$. By using set algebra, this becomes $\bigcup_{\forall a \in A} (D - \neg dom(A,o))$. This expression is equal to $\bigcup_{\forall a \in A} dom(A,o)$, or $U(A,o)$. The lemma is thus proved.

[Lemma 4] We want to show that 1) for any $p \in I(S,o)$, o has non-zero probability to be the closest of 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 Lemma 4 holds.

- For statement (1), based on Definition 4 we know that $\forall s \in S$, $dist_{max}(s,p) \geq dist_{min}(o,p)$. Hence, if every object s (except o) is at the position d_s such that $dist_{max}(s,p) = dist(d_s,p)$, and that o is at the position d_o where $dist_{min}(o,p) = dist(d_o,p)$, every object $s \in S - \{o\}$ is farther away to p than o. Hence, o has a non-zero probability to be the closest to p.
- For statement (2), we know that there exists an object $s \in S$ where p is located in $dom(s,o)$. Note that $s \neq o$, because $dom(o,o) = \emptyset$ (cf. lemma 2). According to Definition 3, $dist_{max}(s,p) < dist_{min}(o,p)$, meaning that s is always closer to p than o. Hence, o must not be the closest to p.

[Lemma 5] $\forall p \in u(o)$, $dist_{min}(o,p) = 0$. Since $dist_{max}(s,p) \geq 0$, there does not exist any $s \in S$ such that $p \in dom(s,o)$. Thus, $p \in I(S,o)$, or equivalently, $\mathcal{V}(o)$ (by Lemma 4). Thus, $u(o) \subseteq \mathcal{V}(o)$.

[Lemma 6] By using Lemma 4, we can rewrite $\mathcal{V}(o)$ as:

$$\mathcal{V}(o) = \{p \in D | \forall s \in S, dist_{max}(s,p) \geq dist_{min}(o,p)\} \quad (4)$$

Now, let $\odot(c,r)$ denote a circle of centre c and radius r. Also, suppose by contrary that $\mathcal{V}(o)$ is not connected. Without loss of generality, suppose $\mathcal{V}(o)$ consists of two disjoint parts, namely R_1 and R_2, as shown in Figure 12. Since $u(o) \subseteq \mathcal{V}(o)$, suppose that $u(o) \subseteq R_1$. Now, given a point $p \in R_2$, based on Equation 4, we know that $\forall s \in S, dist_{max}(s,p) \geq dist_{min}(o,p)$. Let $dist_{min}(o,p)$ be $dist(v,p)$, where v is a point in $u(o)$. Let the uncertainty region of s be $u(s)$. Then there must not exist $s \in S$ such that $u(s)$ is enclosed by circle $\odot(p, dist(v,p))$; else, $dist_{max}(s,p) < dist_{min}(o,p)$.

Since R_1 and R_2 are not connected, there exists a point p' on line segment l_{vp} such that $p' \notin \mathcal{V}(o)$, which means that

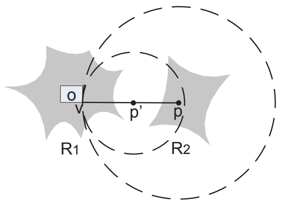


Fig. 12. Proof of Lemma 6.

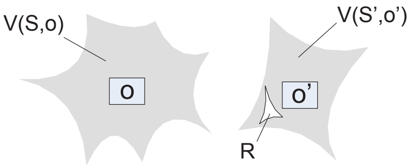


Fig. 13. Proof of Lemma 8.

$\exists s \in S$ such that $dist_{max}(s,p') < dist_{min}(o,p')$. Hence, $u(s)$ is enclosed by circle $\odot(p', dist_{min}(o,p'))$. However, $p' \in l_{vp}$, which means $dist_{min}(o,p') \leq dist(v,p')$. Hence, $\odot(p', dist_{min}(o,p'))$ is enclosed by $\odot(p, dist(v,p))$. This contradicts with the fact that $\nexists s \in S$ such that $u(s)$ is enclosed by $\odot(p, dist(v,p))$. Hence, $\mathcal{V}(o)$ must be a connected region.

C. Section V

[Lemma 7] Notice that

$$I(S,o) = (\bigcap_{\forall s \in T} \neg dom(s,o)) \cap (\bigcap_{\forall s \in S/T} \neg dom(s,o)) \quad (5)$$

Since $\mathcal{V}(o) = I(S,o)$ (by Lemma 4), we can see from Equation 5 that $\mathcal{V}(o) \subseteq \bigcap_{\forall s \in T} \neg dom(s,o)$, or equivalently, $I(T,o)$. By Definition 8, $T = \mathcal{C}_{set}(o)$.

D. Section VI

[Lemma 8] The proof is equivalent to showing that the following three claims are correct:

- **Claim 1:** If object o' is deleted from S, and $\mathcal{V}(S,o) \cap \mathcal{V}(S,o') = \phi$, then o is not affected.
- **Claim 2:** If object o' is inserted to S, and $\mathcal{V}(S,o) \cap \mathcal{V}(S',o') = \phi$, then o is not affected.
- **Claim 3:** If object o' is inserted to (or deleted from) S, and $u(o) \cap u(o') \neq \phi$, then o is not affected.

The following explains why these claims are correct.

[Claim 1] Let $S' = S - \{o'\}$ be the database after o' is removed. We want to show that if $\mathcal{V}(S,o)$ does not intersect $\mathcal{V}(S,o')$ (Figure 13), then $\mathcal{V}(S,o) = \mathcal{V}(S',o)$ after the deletion. Using Lemma 4, this is equivalent to showing the following:

$$\mathcal{V}(S,o) = \mathcal{V}(S',o) \quad (6)$$

From Lemma 4, we know that $\mathcal{V}(S',o) = I(S',o)$. Based on Lemma 7, since $S' \subseteq S$, we have $\mathcal{V}(S,o) \subseteq \mathcal{V}(S',o)$. Suppose that $\mathcal{V}(S',o) = \mathcal{V}(S,o) \cup R$ where R is some region in D such that $\mathcal{V}(S,o) \cap R = \emptyset$. Hence, if $R = \emptyset$, Equation (6) is true. The following explains why $R = \emptyset$.

First, we show that $R \subseteq \mathcal{V}(S, o')$ (i.e., Figure 13). Suppose by contrast that this is not true. That is, there exists a point $p \in R$ such that $p \notin \mathcal{V}(S, o')$. Since $p \in R$, $p \notin \mathcal{V}(S, o)$ and $p \in \mathcal{V}(S', o)$. Moreover, as $\mathcal{V}(S, o) = \mathcal{V}(S', o) \cap \neg dom(o', o)$, p cannot be inside $\neg dom(o', o)$; otherwise, $p \in \mathcal{V}(S, o)$. Thus, $p \in dom(o', o)$. On the other hand, because $p \notin \mathcal{V}(S, o')$, there exists $s \in S$ such that $p \in dom(s, o')$. Since $p \in \mathcal{V}(S', o)$, $s \notin S'$. Thus $s = o'$. Because $dom(o', o') = \emptyset$ (Lemma 2), p cannot exist. Hence, $R \subseteq \mathcal{V}(S, o')$.

Now, since $R \subseteq \mathcal{V}(S, o')$ and $\mathcal{V}(S, o)$ does not intersect $\mathcal{V}(S, o')$, if $R \neq \emptyset$, $\mathcal{V}(S', o) = \mathcal{V}(S, o) \cap R$ will be unconnected, which violates the result of Lemma 6. Thus, $R = \emptyset$, and the claim is correct.

[Claim 2] Let $S' = S \cup \{o'\}$, i.e., the database after inserting o'. From Lemma 9, we know that the new PV-cell of any object $o \neq o'$ must not be smaller than before, i.e., $\mathcal{V}(S', o) \subseteq \mathcal{V}(S, o)$. Hence, $\mathcal{V}(S', o) \cap \mathcal{V}(S', o') = \phi$. Next, we apply the result of Claim 1 in Lemma 8, by viewing S as result of removing o' from S', i.e., "If object o' is deleted from S', and $\mathcal{V}(S', o) \cap \mathcal{V}(S', o') = \phi$, then o is not affected." The lemma is thus proved.

[Claim 3] For deletion, since $u(o')$ overlaps $u(o)$, Lemma 2 tells us that $dom(o', o) = \emptyset$. Thus, $\neg dom(o'o) = D$, and

$$\mathcal{V}(S, o) = I(S, o) = I(S - \{o'\}, o) = \mathcal{V}(S', o)$$

For insertion, we can similarly argue that $\mathcal{V}(S, o) = I(S \cup \{o'\}, o) = \mathcal{V}(S', o)$. Hence, Claim 3 is correct.

[Lemma 9] For **deletion**, let S' be the database after deleting o'. Given an object $o \neq o'$, according to Lemma 4, $\mathcal{V}(S, o) = I(S, o)$ and $\mathcal{V}(S', o) = I(S', o)$. Since $S' \subset S$, S' is a C-set of $\mathcal{V}(S, o)$ (Lemma 7). Using Definition 8, we know that

$$\mathcal{V}(S, o) \subseteq I(S', o)$$

Hence, $V(S, o) \subseteq V(S', o)$. In other words, the PV-cell of o cannot be smaller than before.

For **insertion**, S' contains o'. We can use the proof in the above paragraph, by substituting S with S' and S' with S. We can then show that $V(S', o) \subseteq V(S, o)$. Thus, o's PV-cell cannot be larger than before.

E. The IS Algorithm (Section V-A)

The **Incremental Selection (IS)** is a heuristic used by `chooseCSet`. Its details are shown in Algorithm 2. Step 2 divides the space of D into 2^d disjoint partitions, based on the coordinates of o. Each partition P_i (where $i = 1, \ldots, 2^d$) is associated with a counter c_i, which records the number of NN's that intersect P_i (Step 3). The variable c_{global} records the total number of NN's scanned (Step 4). Steps 5-11 perform the incremental scanning of NN's. We first retrieve the next NN, o_n, of o (Step 6). If $u(o_n)$ overlaps $u(o)$, we skip it, increment c_{global}, and scan the next NN (Steps 7 and 11). Otherwise, for partition P_i that intersects $u(o_n)$, we increment c_i, and insert o_n to $\mathcal{C}_{set}(o)$ (Steps 8-11). These steps are repeated until either (1) the number of NN's scanned exceeds k_{global}; or (2) the counter value of every partition is larger than $k_{partition}$ (Step 5). Finally, Step 12 returns $C_{set}(o)$.

Algorithm 2: Incremental Selection (IS)

```
input : Object $o$, $k_{global}$, $k_{partition}$
output: $C_{set}(o)$
1  begin
2      Let $P_1, ..., P_{2^d}$ be the $2^d$ partitions of $D$ based on the
       $d$-dimensional coordinates of $o$
3      Let $c_i = 0$ be the counter for partition $P_i$
       $(i = 1, \ldots, 2^d)$
4      $c_{global} \leftarrow 0$; $C_{set}(o) \leftarrow \emptyset$
5      while $c_{global} < k_{global}$ or
       $\min\{c_1, \ldots, c_{2^d}\} < k_{partition}$ do
6          Let $o_n$ be the next nearest neighbor of $o$
7          if $u(o_n)$ does not intersect $u(o)$ then
8              for each partition $P_i$ do
9                  if $P_i$ intersects $u(o_n)$ then $c_i \leftarrow c_i + 1$
10             Insert $o_n$ to $C_{set}(o)$
11         $c_{global} = c_{global} + 1$
12     return $C_{set}(o)$
```

F. PNNQ Evaluation

This section summarizes how to evaluate a PNNQ.

1. Answer object retrieval. First, we retrieve the answer objects (i.e., objects with non-zero qualification probabilities). In particular, we use the query point q of the PNNQ to traverse the PV-index, and find the leaf node n_{leaf} whose region contains q. We then retrieve the objects from the disk pages that are associated with n_{leaf}. Let L be the objects retrieved from n_{leaf}. Since for every object $o \in L$, $\mathcal{B}(o)$ overlaps with n_{leaf}, q may be located in $\mathcal{V}(o)$. Let A be the answer object set of the PNNQ, then $A \subseteq L$. To determine A, we can simply perform a min/max refinement in [8]. Specifically, we find out the minimum of maximum distances of objects in L from q. We denote this distance by d_{minmax}. Any object within L that has a minimum distance from q larger than d_{minmax} is removed, since this object has no chance to be the closest to q. The answer objects are those that are not deleted from L.

2. Probability computation. For this step, we adopt the solution from [8]. We assume that the uncertainty pdf of attribute $o.a$ is discrete. In particular, $o.a$ is characterized by a set of d-dimensional points, w_1,...,w_t, which are regularly spaced in uncertainty region $u(o)$. Each instance w_j is associated with a probability $P(w_j)$, to denote the probability that $o.a = w_j$. Also, $\sum_{j=1}^{t} P(w_j) = 1$. A *possible world* $W = \{w_1, .., w_{|S|}\}$ is a set of instances, with exactly one instance chosen from each object. The probability of W is $P(W) = \prod_{i=1}^{|S|} P(w_i)$. The qualification probability of object o is then equal to:

$$\sum_{j=1}^{t} P(w_j) \cdot \mathcal{Q}(w_j). \quad (7)$$

where:

$$\mathcal{Q}(w_j) = \prod_{s \in S/\{o\}} (1 - \sum_{w_s \in s, dist(w_s, q) < dist(w_j, q)} P(w_s)). \quad (8)$$

G. Complexity Analysis of the PV-index

Let M be the number of bytes of main memory, and K be the disk page size. If each pointer to the child nodes occupies 4 bytes, the PV-index has at most $\lfloor M/2^{d+2} \rfloor$ non-leaf nodes. Since each non-leaf node has at most 2^d leaf nodes, the PV-index has at most $\lfloor M/2^{d+2} \rfloor \cdot (1+2^d)$, or $O(M)$ nodes.

To insert a UBR, we first have to compute it by using SE, with a cost $\mathit{cost}_{\texttt{SE}} = O(m_{max} \log(|D|_{max}/\Delta) \cdot |S| \cdot d^2)$. In the worst case, each UBR needs to traverse all the $O(M)$ nodes in the PV-index. Once a leaf node is located, the entry of an object can added to the head of the list in the leaf node in $O(1)$ times. Since $|S|$ UBRs are inserted, the complexity of constructing the PV-index is $O((M + \mathit{cost}_{\texttt{SE}}) \cdot |S|)$.

For Step 1 of PNNQ evaluation, $\log\lfloor M/2^{d+2} \rfloor$ non-leaf nodes need to be retrieved for a query point q, and once the leaf node is found, at most $O(|S|/K)$ pages need to be accessed. Thus, the total querying cost is $O(\log\lfloor M/2^{d+2} \rfloor + |S|/K)$.

Incremental update algorithm. Let us consider deletion and insertion.

(A) For **deletion**,

- **Step 1** needs a cost of $O(1)$.
- **Step 2:** in the worst case, the range search of $\mathcal{B}(S, o')$ touches all the nodes of the primary index, with a cost of $O(M)$. For each of the leaf nodes, we examine $O(|S|/K)$ pages to retrieve the objects and their UBR information, and this costs $O(M \cdot |S|/K)$. Retrieving the UBRs from the secondary index requires a cost of $O(|S|)$, for all objects in S in the worst case. Thus, the total cost of this step is $O(M \cdot |S|/K)$.
- **Step 3:** For every affected object in A, we run a slightly-changed version of SE which requires the same cost of SE, i.e., $\mathit{cost}_{\texttt{SE}}$. In the worst case, $A = S$, and so the cost of this step is $\mathit{cost}_{\texttt{SE}} \cdot |S|$.
- **Step 4** deletes the entries of o' from the leaf nodes in $O(M \cdot |S|/K)$ times. Finding N and N', as well as inserting the entries of every affected object to $N - N'$, needs a cost of $O(M)$. Updating the information of an affected object in the secondary index requires $O(1)$ times. Since we have $O(|S|)$ affected objects, the total cost is $O(M \cdot |S|/K + M \cdot |S|)$ times.

The total cost of the above steps is $O((M + \mathit{cost}_{\texttt{SE}}) \cdot |S|)$.

(B) For **insertion**,

- **Step 1** needs a cost of $\mathit{cost}_{\texttt{SE}}$;
- **Step 2**: in the worst case, the range search of $\mathcal{B}(S', o')$ touches all the nodes of the primary index, with a cost of $O(M)$. For each of the leaf nodes, we examine $O(|S|/K)$ pages to retrieve the objects and their UBR information, and this costs $O(M \cdot |S|/K)$. Retrieving the UBRs from the secondary index requires a cost of $O(|S|)$, for all objects in S in the worst case. Thus, the total cost of this step is $O(M \cdot |S|/K)$.
- **Step 3** runs a slightly-changed version of SE for every affected object in A. In the worst case, the cost of this step is $\mathit{cost}_{\texttt{SE}} \cdot |S|$.
- **Step 4**: Finding N and N', as well as inserting the entries of an affected object to $N' - N$, run at a cost of $O(M)$. Updating the information of an affected objects in the secondary index requires $O(1)$ times. Since we have $O(|S|)$ affected objects, the cost of handling affected objects is $O(M \cdot |S|)$. Inserting the entries of o' to the leaf nodes costs $O(M)$. Thus, the total cost of this step is $O(M \cdot |S|)$ times.

The total cost of the above steps is $O((M + \mathit{cost}_{\texttt{SE}}) \cdot |S|)$.

In the worst case, *incremental* has the same complexity as the cost of reconstructing the index. In our experiments, however, *incremental* runs much faster than rebuilding the index.