UPI: A Primary Index for Uncertain Databases

Hideaki Kimura
Brown University
hkimura@cs.brown.edu

Samuel Madden
MIT CSAIL
srmadden@mit.edu

Stanley B. Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

Uncertain data management has received growing attention from industry and academia. Many efforts have been made to optimize uncertain databases, including the development of special index data structures. However, none of these efforts have explored primary (clustered) indexes for uncertain databases, despite the fact that clustering has the potential to offer substantial speedups for non-selective analytic queries on large uncertain databases. In this paper, we propose a new index called a UPI (*Uncertain Primary Index*) that clusters heap files according to uncertain attributes with both discrete and continuous uncertainty distributions.

Because uncertain attributes may have several possible values, a UPI on an uncertain attribute duplicates tuple data once for each possible value. To prevent the size of the UPI from becoming unmanageable, its size is kept small by placing low-probability tuples in a special Cutoff Index that is consulted only when queries for low-probability values are run. We also propose several other optimizations, including techniques to improve secondary index performance and techniques to reduce maintenance costs and fragmentation by buffering changes to the table and writing updates in sequential batches. Finally, we develop cost models for UPIs to estimate query performance in various settings to help automatically select tuning parameters of a UPI.

We have implemented a prototype UPI and experimented on two real datasets. Our results show that UPIs can significantly (up to two orders of magnitude) improve the performance of uncertain queries both over clustered and unclustered attributes. We also show that our buffering techniques mitigate table fragmentation and keep the maintenance cost as low as or even lower than using an unclustered heap file.

1. INTRODUCTION

A wide range of applications need to handle uncertainty. Uncertainty comes from sources such as errors in measuring devices (e.g., sensors), probabilistic analysis, and data integration (e.g., integration of multiple semantic databases that are potentially inconsistent). As shown by the large body of recent research in this area [4–6, 14, 15], there is a high demand to process such uncertain data in an efficient and scalable manner.

The database community has made great progress in the area of uncertain databases by establishing new data models, query semantics and optimization techniques. Several models for uncertainty in databases have been proposed. In the most general model, both tuple existence and the value of attributes can be uncertain. For example, Table 1 shows 3 uncertain tuples in the *Author* table of a publications database modeled after the DBLP computer science bibliography (see Section 7.1 for how we derived the uncertainty). Each tuple has an existence probability that indicates the likelihood it is in the table and an uncertain attribute (denoted as ^{p}) *Institution* that the author works for. In the example, Alice exists with probability 90% and, if she exists, works for Brown with probability 80% and MIT with probability 20%.

Table 1: Running Example: Uncertain *Author* table

Name	*Institution*^{p}	Existence	...
Alice	Brown: 80%, MIT: 20%	90%	...
Bob	MIT: 95%, UCB: 5%	100%	...
Carol	Brown: 60%, U. Tokyo: 40%	80%	...

Query 1: Example Uncertain Query.
SELECT * FROM Author WHERE Institution=MIT
Threshold: confidence $\geq QT$ (QT is given at runtime)

Possible World Semantics [6] is a widely used model for uncertainty in databases. It conceptually defines an uncertain database as a probability distribution over a collection of possible database instances (*possible worlds*). Each possible world is a complete, consistent and deterministic database instance as in traditional DBMS. For example, there is a possible world where Alice exists and works for Brown, Bob works for MIT and Carol does not exist. The probability of such a world is $90\% \times 80\% \times 95\% \times 20\% \approx 13.7\%$. Based on possible world semantics, a probabilistic query over an uncertain database can output tuples along with a *confidence* indicating the probability that the tuple exists in some possible world where it satisfies the query predicates. For example, Query 1 would answer $\{$(Alice, confidence=$90\% \times 20\% = 18\%$), (Bob, 95%)$\}$. Thus, confidence represents how probable each answer is. Users can also specify thresholds on the minimum confidence they require from query results (the QT in Query 1.)

Though possible world semantics is a widely used data model, achieving an efficient implementation is difficult. In particular, it requires a new approach to data storage, access methods and query execution [6]. One active area of research has been in building index data structures to efficiently answer queries over such probabilistic tables [2, 5, 16]; the primary addition that these data structures provide over traditional B+Trees and R-Trees is the ability to find tuples with confidence above some specified threshold.

These proposed indexes, however, are *secondary* indexes. To the best of our knowledge, no work has been done to cluster a heap file containing uncertain attributes as a *primary* index. To address this limitation, the key contribution of this work is to propose techniques to build primary indexes over probabilistic databases. Just as in a conventional (non-probabilistic) database, a primary index can be orders of magnitude faster than a secondary index for queries that scan large portions of tables, for example in OLAP workloads. Because a secondary index stores only index keys with pointers to corresponding tuples in the heap file, the query executor has to access the heap file by following the pointers to retrieve non-indexed attributes. This can cause an enormous number of random disk seeks for an analytical query that accesses millions of tuples, even if the query executor sorts the pointers before accessing the heap file (e.g., *bitmap index scan*). Furthermore, recent work has shown that building primary indexes on appropriate attributes can also boost the performance of secondary indexes that are correlated with the primary index [9]. In this paper, we demonstrate that a new primary index structures on uncertain attributes can be up to two orders of magnitude faster than a secondary index and can boost the performance of secondary indexes by up to two orders of magnitude when an appropriate correlated primary index is available.

However, building a primary index on uncertain attributes poses several challenges. If we simply cluster a tuple on one of its possible values, a query that is looking for other possible values needs additional disk accesses. For example, if we store Carol in a *Brown* disk block, a query that inquires about *U. Tokyo* authors must access the Brown block in addition to the *U. Tokyo* block. One solution is to replicate tuples for every possible value, but this makes the heap file very large and increases maintenance especially for *long tail* distributions with many low probability values. Furthermore, building a primary index on attributes other than auto-numbered sequences imposes a significant maintenance cost (to keep the heap clustered) and leads to fragmentation of the heap file over time, which also slows down the query performance.

In this paper, we develop a novel data structure we call the UPI (*Uncertain Primary Index*), which is a primary index on uncertain attributes with either discrete or continuous distributions. UPI replicates tuples for all possible values but limits the penalty by storing tuples with a probability less than some threshold in a *Cutoff Index*. We propose a novel data structure for secondary indexes built over UPIs that stores multiple pointers for each entry to take advantage of the replicated tuples. We also describe the *Fractured UPI* which buffers data updates and occasionally flushes them to a new partition, or a *fracture* to reduce maintenance costs and fragmentation. Our experimental results on two real uncertain datasets show that UPI has substantial performance gains and similar maintenance costs to (unclustered) heap files.

In summary, our contributions include:

- The UPI data structure and corresponding secondary indexes
- Algorithms to answer queries using UPIs
- Methods to reduce update cost and fragmentation of UPIs
- Cost models to help select cutoff values and guide the formation of cutoff indexes
- Experimental results on real datasets that verify our approach and demonstrate order-of-magnitude performance gains over existing secondary indexses

In the next section, we describe a naive implementation of UPI and discuss its limitations. Sections 3 through 6 extend UPIs to address these limitations. Section 7 validates our approach with intensive experiments on two real datasets. Finally, Section 8 summarizes related work and Section 9 concludes this paper.

2. A SIMPLE UPI

We begin by describing a naive implementation of UPIs, followed by a discussion of their shortcomings that are addressed in later sections.

To answer Query 1, an uncertain secondary index on *Institution* would be inefficient because there are thousands of researchers who work for MIT, and each would require a random disk seek to fetch. Instead, if we build a UPI on *Institution*, it will duplicate each tuple once for each possible value of *Institution*, as shown in Table 2.

Also, we do not need tuples that have less than QT probability to satisfy the query. Therefore, we order the tuples by decreasing probability of institution, which allows the query executor to terminate scanning as soon as it finds a tuple that has a lower probability than the query threshold. Physically, the heap file is organized as a B+Tree indexed by $\{$*Institution* (ASC) and *probability* (DESC)$\}$. This is similar to the inverted index in [13] except that we duplicate the entire tuple, rather than just a pointer to the heap file.

Table 2: A Naive UPI. Sorted by institution and probability.

Institution^p$\downarrow$ (*Probability*$\uparrow$)	*TupleID*	Tuple Data
Brown (80%*90%=72%)	Alice	...
Brown (60%*80%=48%)	Carol	...
MIT (95%)	Bob	...
MIT (18%)	Alice	...
UCB (5%)	Bob	...
U. Tokyo (32%)	Carol	...

This scheme achieves significantly faster performance than a secondary index for Query 1 because it requires only one index seek followed by a sequential scan of matching records. However, this naive UPI has several limitations.

First, since it duplicates the whole tuple for every possible value of *Institution*, the size of the heap file can be significantly larger than a heap file without the primary index. This is especially true when the probabilistic distribution has a long tail (i.e., many possible values with low probabilities).

Second, now that a single tuple exists in multiple places on disk, it is not clear how we should organize secondary indexes. Specifically, if we could use the duplicated tuples, a query could use the secondary index to access fewer heap blocks (fewer seeks) and run substantially faster.

Third, maintaining UPIs causes two problems. As newly inserted or deleted tuples will have different values of *Institution*, we need to update the B+Tree nodes in a variety of locations leading to many disk seeks. Also, splits and merges of B+Tree nodes will fragment the disk layout of the UPI and degenerate query performance.

Lastly, the naive approach applies only to tuples with discrete probability distributions. For continuous distributions like Gaussians, we need index schemes other than B+Trees.

We address these problems in turn. Section 3 describes the design *Cutoff Indexes* to address long-tail distributions and proposes a new index data structure for a secondary index that exploits duplicated tuples in the UPI. Section 4 explains the design of *Fractured UPIs* that minimize UPI maintenance cost and fragmentation. Section 5 extends UPIs for continuous distributions. Finally, Section 6 defines cost models which are useful to design and maintain UPIs.

3. IMPROVED UPI

In this section, we improve our basic UPI design by addressing issues with the database size and improving the performance of secondary indexes on the same table as the UPI.

3.1 Cutoff Index

One problem with our naive UPI is that the database size can grow significantly when a tuple has many possible values of the indexed attribute. This increased size will not only affect the storage cost but also increase maintenance costs.

We observe, however, that for long-tailed distributions, with many duplicated values, the user may not care about very low confidence tuples, since those are unlikely to be correct answers. For example, Query 1 includes the threshold QT that filters out low-confidence tuples. Such queries are called Probabilistic Threshold Queries, or PTQs, and are very common in the literature [2, 5, 16]. For PTQ's, low probability tuples can typically be ignored.

We anticipate that most queries over long-tailed distributions will be PTQs. To handle such queries, we attach a *Cutoff Index* to each UPI heap file. The idea is that the query executor does not need to read the low probability entries when a relatively high probability threshold is specified in a PTQ. Therefore, we can remove such entries from the UPI heap file and store them in another index, which we call the *cutoff index*. The cutoff index is organized in the same way as the UPI heap file, ordered by the primary attribute and then probability. It does not, however, store the entire tuple but only the

Input: t: Inserted tuple, C: Cutoff threshold.

```
Alternatives = sort_by_probability (t.primary_attribute);
foreach $a \in Alternatives$ do
    if $a = Alternatives.first$ OR $a.probability \geq C$ then
        Add (key: $a$, tuple: $t$) to Heap File;
    else
        Add (key: $a$, pointer: $Alternatives.first$, TupleID: $t.TupleID$) to Cutoff Index;
    end
end
```

Algorithm 1: Insertion into a UPI

Input: *key*: Queried value, *QT*: Probability threshold, C
Output: *S*: Set of tuples to return.

```
$S = \emptyset$;
Cur = UPI.seekTo (key);
while $Cur.key = key$ AND $Cur.probability \geq QT$ do
    $S = S \bigcup Cur.tuple$;
    Cur.advance();
end
if $QT < C$ then
    Cur = CutoffIndex.seekTo (key);
    while Cur.key = key AND Cur.probability $\geq QT$ do
        CurIn = UPI.seekTo (Cur.pointer);
        CurIn.moveTo (Cur.TupleID);
        $S = S \bigcup CurIn.tuple$;
        Cur.advance();
    end
end
```

Algorithm 2: Answering a PTQ using a UPI

Table 3: Cutoff Index to compress UPI (C=10%)

UPI Heap File

Brown (72%)	Alice	...
Brown (48%)	Carol	...
MIT (95%)	Bob	...
MIT (18%)	Alice	...
U. Tokyo (32%)	Carol	...

Cutoff Index

Key↓	TupleID	Pointer
UCB (5%)	Bob	MIT

Stores pointers for possible values with probability $< C$

uncertain attribute value, a (*pointer*) to the heap file to locate the corresponding tuple, and a tuple identifier (*TupleID*). For example, in Table 3, the Bob tuple with institution value *UCB*, which has only 5% probability, is moved to the cutoff index with a pointer to another possible value of Bob (MIT).

Top-k queries and nearest neighbor (*NN*) queries [14] benefit from the cutoff index as well. A top-k query can terminate scanning the index when the top-k results are identified. Thus, a cutoff index is particularly useful when a majority of the queries on the database are PTQs or Top-k.

Algorithm 1 shows how we build and maintain UPIs and cutoff indexes. Given a *Cutoff Threshold C* for the UPI, we duplicate a tuple in the UPI for every possible value that has probability equal to or greater than C. For every possible value with probability less than C, we insert a pointer to the first possible value of that tuple (a value that has highest probability) into the cutoff index. If a value has probability lower than C, but is the first possible value, we leave the tuple in the UPI instead of moving it, to not lose tuples that do not have any possible value with probability larger than C.Deletion from the UPI is handled similarly, deleting entries from the heap file or cutoff index depends on the probability. Updates are processed as a deletion followed by an insertion.

Algorithm 2 shows how we use the UPI to answer PTQs. When C is less than QT, we simply retrieve the answer from the UPI heap file, which requires only one index seek. When C is larger than QT, we additionally need to look in the cutoff index to retrieve cutoff pointers and perform an index seek for each pointer.

The value of C is an important parameter of a UPI that the database administrator needs to decide. Larger C values could reduce the size of the UPI by orders of magnitude when the probability distribution is long tailed. But, they substantially slow the performance of PTQs with query threshold less than C, since such queries require pointer-following (and many random I/Os.) Smaller values of C work well for a large mix of queries with varying QT, at the cost of a larger UPI. To help determine a good value of C taking into account both the workload and limits on storage consumption and maintenance cost, we developed an analytic model for cutoff index performance; we present this model in Section 6.

Input: *key*: Queried value, *QT*: Probability threshold.
Output: *P*: Set of pointers to heap file.

```
$P = \emptyset$;
$Entries = SecondaryIndex.select(key, QT)$;
foreach $e \in Entries$ do
    if $e.pointers.length = 1$ then
        $P = P \bigcup e.pointers[0]$;
    end
end
foreach e in Entries do
    if $\forall p \in e.pointers : p \notin P$ then
        $P = P \bigcup e.pointers[0]$;
    end
end
```

Algorithm 3: Tailored Secondary Index Access

Table 4: *Country^p* in Author table

Name	Institution^p	Country^p	Existence
Alice	Brown: 80%, MIT: 20%	US: 100%	90%
Bob	MIT: 95%, UCB: 5%	US: 100%	100%
Carol	Brown: 60%, U. Tokyo: 40%	US: 60%, Japan: 40%	80%

Table 5: Secondary Index on *Country^p*

Country^p↓	TupleID	Pointers	
Japan (32%)	Carol	Brown	U. Tokyo
US (100%)	Bob	MIT	<*cutoff*>
US (90%)	Alice	Brown	MIT
US (48%)	Carol	MIT	U. Tokyo

3.2 Secondary Indexes on UPIs

Another challenge is exploiting the structure of UPIs to improve secondary index performance. A secondary index in conventional databases points to a single tuple in the heap file by storing either a *RowID* consisting of physical page location and page offset (e.g., PostgreSQL) or the value of the primary index key (e.g., MySQL InnoDB). Unlike such traditional secondary indexes, in UPIs, we employ a different secondary index data structure that stores multiple pointers in one index entry, since there are multiple copies of a given tuple in the UPI heap.

For example, suppose *Country^p* is another uncertain attribute of the relation Author shown in Table 4 with a secondary index on it as shown in Table 5. Each row in the secondary index stores all possible values of the primary attribute (*Institution^p*), except cutoff values. Algorithm 3 shows our algorithm for answering PTQs using these multiple pointers. For example, suppose the following PTQ is issued on *Country^p* with $QT = 80\%$:

SELECT * FROM Author WHERE Country=US

We first retrieve matching entries from the secondary index (Bob and Alice) and then find entries that have only one pointer (Bob). We record the institution for these pointers (MIT) and then check other secondary index entries, preferentially choosing pointers to institutions we have already seen. In the above case, Alice contains a pointer to MIT, so we retrieve tuple data for Alice from the MIT record about her. The advantage of this is that because Bob's data is also stored in the MIT portion of the heap, we can retrieve data about both authors from a small, sequential region of the heap corresponding to MIT. If there is no pointer to an institution we have already seen, we simply pick the first (highest probability) pointer. Note that in this case we would have accessed two disk blocks (MIT and Brown) if the secondary index stored only the first pointers.

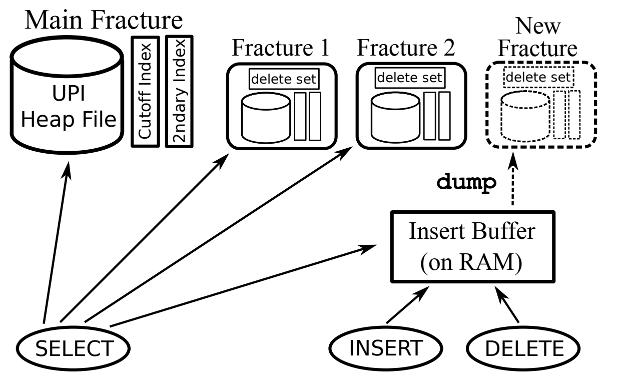

Figure 1: Fractured UPI Structure

We call this algorithm as *Tailored Secondary Index Access* and demonstrate in Section 7 that it can speed up secondary indexes substantially for analytical queries . One tuning option for this algorithm is to limit the number of pointers stored in each secondary index entry. Though the query performance gradually degenerates to the normal secondary index access with a tighter limit, such a limit can lower storage consumption.

4. FRACTURED UPI

In this section, we describe a second extension to UPIs called *Fractured UPIs*. The idea of fracturing is to reduce UPI maintenance cost and fragmentation. The approach we take is similar to that taken in log structured merge trees (*LSM-Trees*) [12] and partitioned exponential files [8] for deterministic databases, which try to eliminate random disk I/O by converting all updates into appends to a log similarly to *deferred updates* of transaction processing.

4.1 The Maintenance Problem

The problem of maintaining a UPI is that insertion or deletion may perform random I/O to the UPI to retrieve pages. This makes the maintenance cost of UPIs much higher than for an append-only table without primary indexes.

Another problem is that insertions cause splits of B+Tree nodes when nodes become full, and deletions cause merges of nodes. Thus, over time, these operations result in fragmentation of the primary index, leading to random disk seeks even when a query requests a contiguous range of the primary index attribute.

For these two reasons, primary B+Tree indexes sometimes have adverse effects on performance over time [8], canceling out the initial benefits obtained by clustering a table on some key.

4.2 Fractured UPI Structure

To overcome these problems, we store UPIs as *Fractured* indexes [7]. Figure 1 shows the structure of a Fractured UPI. The *insert buffer* maintains changes to the UPI in main memory. When the buffer becomes full, we sequentially output the changes (insertions and deletions) to a set of files, called a *Fracture*. A fracture contains the same UPI, cutoff index and secondary indexes as the main UPI except that it contains only the data inserted or deleted since the previous flush. Deletion is handled like insertion by storing a *delete set* which holds IDs of deleted tuples. We keep adding such fractures as more changes are made on the UPI, and do not immediately update the main UPI files.

To answer a SELECT query, the query executor scans the insert buffer and each fracture in addition to the main UPI, returning the union of results from each file and ignores tuples that were contained in any delete set. In this scheme, all files are read-only and are written out sequentially by the clustering key as a part of a single write. Therefore, the maintenance cost is significantly lower and there is essentially no fragmentation.

One difference from prior work (e.g., [12]) is that a fracture contains a *set* of indexes that constitute an independent UPI. A secondary index or a cutoff index in a fracture always points to the heap file in the same fracture. This architecture makes query execution in each fracture simpler and easier to parallelize. The only exception is the *delete set*, which is collected from all fractures and checked at the end of a lookup.

Another benefit of independent fractures is that each fracture can have different tuning parameters as long as the UPI files in the fracture share the same parameters. For example, the cutoff threshold C, the maximum number of pointers to store in a secondary index entry and even the size of one fracture can vary. We propose to dynamically tune these parameters by analyzing recent query workloads based on our cost models whenever the insert buffer is flushed to disk. This kind of adaptive database design is especially useful when the database application is just deployed and we have little idea about the query workload and data growth.

4.3 Merging Fractured UPI

Although fracturing UPIs avoids slowdown due to fragmentation, query performance still deteriorates over time as more and more fractures accumulate. The additional overhead to access the in-memory insert buffer is negligible, but accessing each fracture causes additional disk seeks. This overhead linearly increases for the number of fractures and can become significant over time.

Thus, we need to occasionally reorganize the Fractured UPI to remove fractures and merge them into the main UPI (this is similar to the way in which conventional indexes need reorganization or defragmentation to maintain their performance.) The merging process is essentially a parallel sort-merge operation. Each file is already sorted internally, so we open cursors on all fractures in parallel and keep picking the smallest key from amongst all cursors.

The cost of merging is about the same as the cost of sequentially reading all files and sequentially writing them back out, as we show in Section 7. As the size of the database grows, this merging process could take quite a long time, since it involves rewriting the entire database. One option is to only merge a few fractures at a time. Still, the DBA has to carefully decide how often to merge, trading off the merging cost with the expected query speedup. In Section 6, we show how our cost model can help estimate the overhead of fractures guide the decision as to when to merge.

5. CONTINUOUS UPI

In this section, we extend UPIs to handle attributes with continuous distributions (e.g., spatial attributes). For example, we might have imprecise GPS data for a position that is within a circle of 100m radius centered at (42°, 72°) with a uniform distribution. As the number of possible values in such distributions is infinite, we cannot apply the basic UPI presented above to such attributes.

Our solution is to build a primary index on top of R-Tree variants like PTIs [5] and U-Trees [16]. These indexes themselves are secondary indexes, and as such require additional seeks to retrieve tuples. We cannot make them primary indexes by simply storing tuples in the leaf nodes. As tuples are orders of magnitude larger than pointers, it would significantly reduce the maximum number of entries in a node, resulting in a deep and badly clustered R-Tree with high maintenance costs. Instead, we build a separate heap file structure that is synchronized with the underlying R-Tree nodes to minimize disk access. We cluster this separate heap file by the hierarchical location of corresponding nodes in the R-Tree.

Figure 2 shows a continuous UPI on top of an R-Tree. It consists of R-Tree nodes with small page sizes (e.g., 4KB) and heap pages with larger page size (e.g., 64KB). Each leaf node of the R-Tree is mapped to one heap page (or more than one when tuples for the leaf node do not fit into one heap page). Consider the 3rd entry

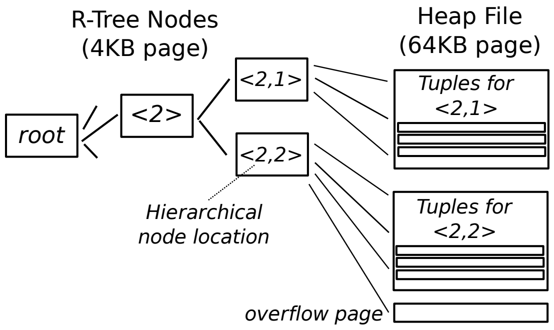

Figure 2: A Continuous UPI on top of R-Tree

in the R-Tree leaf node that is the 1st child of the 2nd child of the root node. We give this tuple the key <2, 1, 3> store it in the third position of heap page <2, 1>. When R-Tree nodes are merged or split, we merge and split heap pages accordingly. In this scheme, tuples in the same R-Tree leaf node reside in a single heap page and also neighboring R-Tree leaf nodes are mapped to neighboring heap pages, which achieves sequential access similar to a primary index as long as the R-Tree nodes are clustered well.

One interesting difference from prior work is that UPIs can exploit duplicated entries in the underlying R-Tree to speed up secondary index accesses as described in Section 3.2. Duplicating entries in an R-Tree (R+Tree) is also useful to reduce overlap of minimum bounding rectangles (MBRs) and improve clustering, which will lead to better query performance. PTIs and U-Trees are based on the R*Tree which does not duplicate entries although it tries to improve clustering by re-inserting entries. Developing an R+Tree analogue might further improve the performance of UPIs especially when wider and less skewed (e.g., Uniform) distributions cause too much MBR overlap. We leave this as future work.

6. COST MODELS

In this section, we develop two cost models that capture the effects of the number of fractures and query thresholds on the query runtime respectively. When we need to account for both effects in one query, both estimates are added to estimate the total query runtime. The cost models are useful for the query optimizer to pick a query plan and for the database administrator to select tuning parameters such as the merging frequency and the cutoff threshold. We verify the accuracy of our cost models in Section 7 and observe that the cost models match the observed runtime quite well.

6.1 Parameters and Histograms

Table 6 shows the list of parameters used in our cost model as well as their values in our experimental environment. We get these parameters by running experiments (e.g., measure the elapsed time to open/close a table in Berkeley DB) and by collecting statistics (using, e.g., DB::stat()) for the particular configuration of interest.

Another input to our cost model is the *selectivity* of the query. Unlike deterministic databases, selectivity in our cost model means the fraction of a table that satisfies not only the given query predicates but also the probability threshold (QT). We estimate the selectivity by maintaining a probability histogram in addition to an attribute-value-based histogram. For example, a probability histogram might indicate that 5% of the possible values of attribute X have a probability of 20% or more. We estimate both the number of tuples satisfying the query that reside in the heap file and that reside in the cutoff index using the histograms. We also use the histogram to estimate the size of the table for a given cutoff threshold.

6.2 Cost Model for Fractured UPIs

We estimate the cost of a query on a Fractured UPI with the following equation. In addition to the sequential read cost, it counts

Table 6: Parameters for cost models

Parameter	Description	Typical Value
T_{seek}	Cost of one random disk seek	10 [ms]
T_{read}	Cost of sequential read	20 [ms/MB]
T_{write}	Cost of sequential write	50 [ms/MB]
H	Height of B+Tree	4
S_{table}	Size of *table*	10 [GB]
N_{leaf}	Count of leaf pages	S_{table} / 8KB
N_{frac}	Count of UPI fractures	10
$Cost_{init}$	Cost to open a DB file	100 [ms]
$Cost_{scan}$	Cost to full scan the table	$T_{read} \cdot S_{table}$

the cost of table initialization and an index lookup for each fracture.

$$Cost_{frac} = Cost_{scan} \cdot Selectivity + N_{frac}(Cost_{init} + HT_{seek})$$

Based on this estimate and the speed of database size growth, a database administrator can schedule merging of UPIs to keep the required query performance. To estimate how long the merging will take, she can simply refer the cost to fully read and write all fractures; $Cost_{merge} = S_{table}(T_{read} + T_{write})$.

6.3 Cost Model for Cutoff Indexes

For a query whose probability threshold QT is less than the cutoff threshold C, we need to access the cutoff index, causing random seeks that are much more expensive than the sequential reads required to access the UPI itself. To confirm this, we ran Query 1 with various values for QT and C.

Figure 3 compares the runtime of a *non-selective* query over the *Author* table that could return as many as 37,000 authors and a *selective* query which returns as many as 300 authors. In both cases, the query performs slower for lower QT especially when $QT < C$ because the query has to access the cutoff index as expected. When $QT \geq C$, the query is very fast because it is answered purely through sequential I/O.

However, the runtime of the non-selective query is the same for all QT when $C > 0.4$. This result is not intuitive because the number of pointers read from the cutoff index should be larger for smaller values QT. In fact, $QT = 0.05$ retrieves 22,000 pointers from the cutoff index while $QT = 0.25$ retrieves 3,000, but the query runtime is the same.

This happens because in both cases we access nearly every page in the table. We call this case *saturation*. As the query needs to retrieve thousands of pointers from the cutoff index, these pointers already cover almost all of the heap file, and the disk access pattern degenerates to a full table scan (assuming the database performs a heap file lookup by ordering the pointers relatively to the positions in the heap file). At this point, further increasing the number of pointers (smaller QT) does not make the query slower. Another interesting observation is that, as demonstrated in the $QT = 0.05$ curve, a query might perform faster with larger C when pointers are saturated because the full table scan cost is smaller.

These observations suggest that query runtime is not simply the number of retrieved pointers multiplied by the disk seek cost, especially when the number is large. Instead, the growth of the number of real disk seeks gradually decreases for more pointers because more and more pointers will land on the same blocks and eventually get saturated. Our main target is non-selective analytical queries, so ignoring this effect can cause a huge error in query cost estimation.

In order to model this saturation behavior, we use a generalized logistic function $f(x)$, which is a type of *sigmoid* function. A sigmoid function is often used to model phenomena like population growth where the rate of reproduction is proportional to the amount of available resource which decreases as population increases. This is consistent with our notion of saturation.

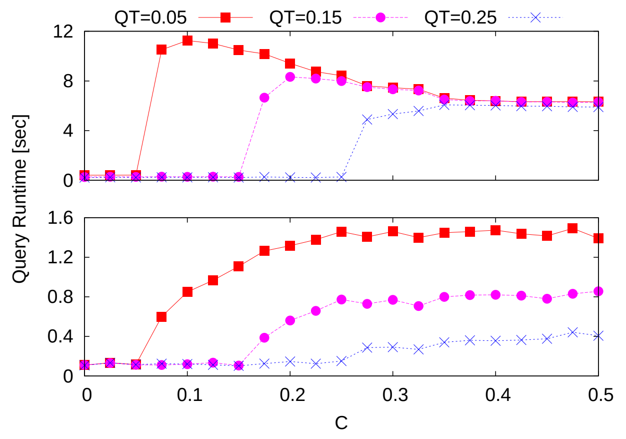

Figure 3: Cutoff Index Real Runtime. Non-selective (top) and Selective (bottom) queries.

$$Cost_{cut} = Cost_{scan} \cdot Selectivity + 2(Cost_{init} + HT_{seek}) + f(\#Pointers)$$

$$f(x) = Cost_{scan}\left(\frac{1-e^{-kx}}{1+e^{-kx}}\right)$$

The first line is basically the same as $Cost_{frac}$ except that we access two tables (the UPI Heap File and the Cutoff Index). $f(x)$ is the cost to retrieve tuples from the heap file which satisfies $f(0) = 0$ and $f(\infty) = Cost_{scan}$. k is a parameter that represents how quickly we reach saturation. We determine this value by applying a heuristic $f(0.05 \cdot N_{leaf}) = 0.99 \cdot Cost_{scan}$, which is based on experimental evidence gathered through our experience with UPIs.

We propose to use the cost models for selecting the cutoff threshold as follows: First, an administrator collects query workloads of the database to analyze the frequency of queries to have low QTs. Second, she figures out the acceptable size of her database given available disk capacity and expected maintenance time. Finally, she picks a value of C that yields acceptable database size and also achieves a tolerable average (or nth percentile) query runtime.

7. EXPERIMENTAL RESULTS

In this section, we evaluate the query and maintenance performance of UPIs as well as the accuracy of our cost models. We implemented a prototype UPI for both discrete and continuous distributions and compared the performance with prior uncertain indexes on two real datasets.

7.1 Setup

All of our UPI implementations are in C++ on top of BDB (BerkeleyDB) 4.7 except the continuous UPI because BDB does not support R-Tree indexes. Instead, we implemented a custom heap file layer (See Section 5) on top of the U-Tree provided by Tao et al [1] which pre-computes integrals of probability distributions as MBRs. For other experiments, we used BDB's B+Trees. We always sort pointers in heap order before accessing heap files similarly to PostgreSQL's bitmap index scan to reduce disk seek costs caused by secondary index accesses. Our machine for all experiments runs Fedora Core 11 and is equipped with a quad core CPU, 4GB RAM and 10k RPM hard drive. All results are the average of 3 runs, and were performed with a cold database and buffer cache.

DBLP Dataset and Query: Our first dataset is derived from DBLP [10], the free bibliographic database of computer science publications. DBLP records more than 1.3 million publications and 700k authors. This dataset exemplifies uncertainty as a result of data integration. DBLP itself has no uncertainty but by integrating DBLP with other data sources, we can produce uncertain data. For instance, the affiliated institution of each author is useful information for analysis, but is not included in DBLP. SwetoDblp [3] supplies it by integrating DBLP with ontology databases. Nagy et al [11] applied machine learning techniques to automatically derive affiliation information by querying a web search engine and then analyzing the homepages returned by that search engine.

Such analysis is useful but inherently imprecise, so the resulting affiliation information is uncertain. We generated such uncertain affiliations by querying all author names in DBLP via Google API and assigning probabilities to the returned institutions (determined by domain names) up to ten per author. We used a zipfian distribution to weigh the search ranking and sum the probabilities if an institution appears at more than one ranks for the author.

The resulting data is the *Author* table exemplified in Table 4 which has uncertain attributes like institution and country for all 700k authors. We also added the same uncertain attributes into the list of publications (assuming the last author represents the paper's affiliation) and stored it as the *Publication* table which contains information about 1.3M publications.

We loaded the uncertain data into BDB and built a UPI on the *Institution* attribute with various cutoff thresholds. For the Publication table, we also built a secondary index on Country, which is correlated with Institution. We then experimented with the following queries on the two tables.

Query 1: Author Extraction
SELECT * FROM Author WHERE Institution=MIT

Query 2: Publication Aggregate on Institution
SELECT Journal, COUNT(*) FROM Publication
WHERE Institution=MIT GROUP BY Journal

Query 3: Publication Aggregate on Country
SELECT Journal, COUNT(*) FROM Publication
WHERE Country=Japan GROUP BY Journal

Cartel Dataset and Query: Our second dataset is derived from Cartel (`http://cartel.csail.mit.edu`) data. Cartel is a mobile sensor network system which collects and analyzes GPS data sent from cars to visualize traffic. During the analysis, the raw GPS data is converted into car observations which contain the location, estimated speed, road segment and the direction of cars. Because of the imperfect accuracy of GPS and probabilistic analysis, the resulting car observations are uncertain.

We generated uncertain Cartel data based on one year of GPS data (15M readings) collected around Boston. We assigned a constrained Gaussian distribution to location with a boundary to limit the distribution as done in [16] and added an uncertain road segment attribute based on the location. We built our 2-D continuous UPI on the uncertain location attribute (i.e., longitude/latitude) and also built a secondary index on the road segment attribute. We then experimented with the following queries.

Query 4: Cartel Location
SELECT * FROM CarObservation
WHERE Distance(location, 41.2°, 70.1°) $\leq$ Radius

Query 5: Cartel Road Segment
SELECT * FROM CarObservation WHERE Segment=123

7.2 Results

UPI on Discrete Distributions: We now present our experimental results, starting with DBLP. The DBLP dataset has discrete distributions on several attributes, therefore, we compare our UPI with our implementation of PII [13] on an unclustered heap file. PII is an uncertain index based on an inverted index which orders inverted entries by their probability. We compared UPI with PII because PII has been shown to perform fast for discrete distributions [13].

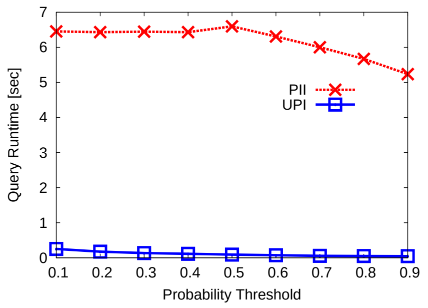

Figure 4: Query 1 Runtime

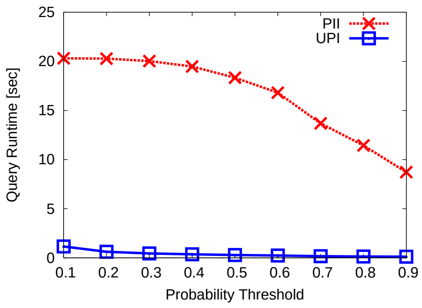

Figure 5: Query 2 Runtime

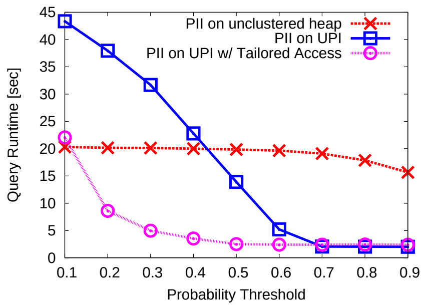

Figure 6: Query 3 Runtime

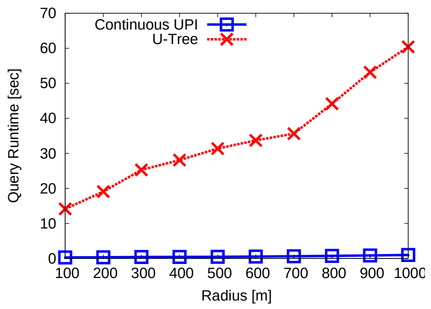

Figure 7: Query 4 Runtime

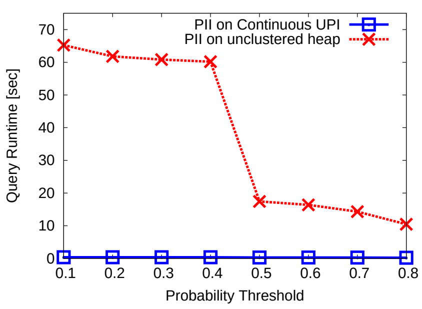

Figure 8: Query 5 Runtime

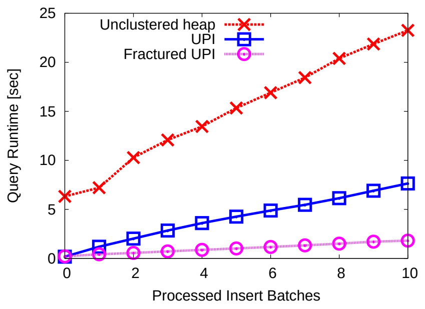

Figure 9: Q1 (C=QT=0.1) Deterioration

Figure 4 and Figure 5 show the runtimes of Query 1 and Query 2, comparing UPIs ($C = 10\%$) and PIIs on Institution. Both indexes perform faster with higher thresholds as they retrieve less data, but the UPI performs 20 to 100 times faster because the UPI sequentially retrieves tuples from the heap file while PII needs to do random disk seeks for each entry.

Figure 6 shows the runtime of Query 3 which uses a secondary index on Country. This time, we also test the UPI with and without tailored secondary index access as described in Section 3.2. Although both use secondary indexes in this case, our index performs faster because of correlation between the attributes of the primary and secondary indexes. However, the UPI without tailored index access is not very beneficial, and sometimes is even slower than the unclustered case because it cannot capture the possible overlap of pointers from the secondary index. Our tailored index access performs up to a factor of 7 faster than the UPI without tailored access, and up to a factor of 8 faster than PII.

UPI on Continuous Distributions: Next, we compare a continuous UPI with a secondary U-Tree on the Cartel dataset. Figure 7 shows the performance comparison between a 2-D continuous UPI and a U-Tree on Query 4. We fixed $QT = 50\%$ and varied the radius. The continuous UPI performs faster by a factor of 50 to 60 because the tuples in the UPI heap file are well clustered with the intermediate nodes. Figure 8 shows the runtime of Query 5, varying QT ($QT = 90\%$ returns no result). Both techniques use secondary indexes for this query. However, as in the discrete case, the secondary index performs much faster with a continuous UPI because of correlation between lat/long (primary index) and segment ID (secondary index) which reduces the number of disk seeks by orders of magnitude. The speed up is a factor of up to 180 when $QT < 50\%$. For queries $QT > 50\%$ (more selective queries) which have many fewer pointers to follow, heap access on both indexes are much faster so the performance gap is less because secondary index access cost is the same. However, the gap is still more than a factor of 50.

Fractured UPIs: We now evaluate maintenance of UPIs. To measure the maintenance cost, we randomly delete 1% of the tuples from the DBLP Author table and randomly insert new tuples equal to 10% of the existing tuples. We compare an unclustered table (clustered by an auto-increment sequence), a UPI and a Fractured UPI. For the Fractured UPI, we drop the insert buffer after all insertions and deletions.

Table 7: Maintenance Cost

	Insert	Delete
Unclustered	7.8 sec	75 sec
UPI	650 sec	212 sec
Fractured UPI	4.0 sec	0.03 sec

Table 8: Merging Cost

#	Time	DB size
1	150 sec	2.5 GB
2	247 sec	3.6 GB
3	275 sec	4.8 GB

As shown in Table 7, the non-fractured UPI performs quite poorly for both insertions and deletions because random locations in the B+Tree are read, written, split and merged. Unclustered and Fractured UPIs perform well because they sequentially write the inserted data to disk. Note that for deletions, even an unclustered table performs poorly because tuples are deleted from random places. The Fractured UPI performs much faster because it simply buffers TupleIDs of deleted tuples and sequentially writes them to disk as a batch.

We also tested the query performance deterioration after a number of insert batches, each of which consists of the 10% insertions and 1% deletions (as before). For the Fractured UPI, we made one fracture after each insert batch. Figure 9 shows the query runtime deterioration. After 10 insert batches, the table size is increased by only 90% (=10*(10%-1%)), but all three approaches show much more than 90% deterioration. The unclustered table becomes 4 times slower compared with the initial state, the non-fractured UPI is 40 times slower and the Fractured UPI is 9 times slower. For the unclustered table and the UPI, the slowdown is because of fragmentation caused by deletion and (for UPI) insertion.

This result illustrates that the Fractured UPI improves not only the maintenance cost but the query performance by eliminating fragmentation. Still, the Fractured UPI does gradually slow down because of the overhead of querying each fracture.

Cost Models: To restore the query performance of the Fractured UPI, we implemented merging of fractures and compared that with our cost model for fractures described in Section 6.2. Figure 10 shows the real and estimated query runtime during 30 insert batches. We merged fractures after every 10 insert batches. The query performance is restored after each merging, and the estimated runtime matches the real runtimes quite well. Table 8 shows the cost of three merges. As the result shows, the merge cost is almost the same as reading and writing the entire table in BDB

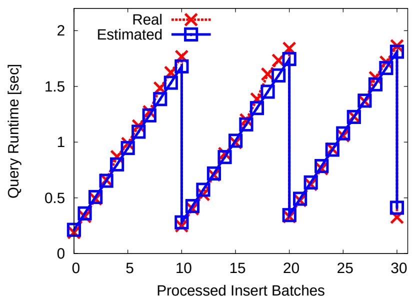

Figure 10: Fractured UPI Runtime

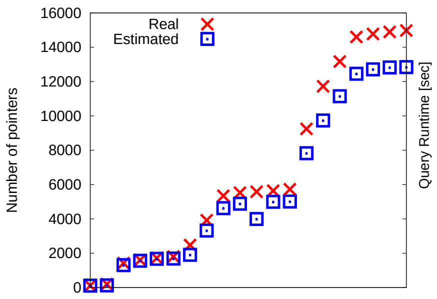

Figure 11: #Cutoff-Pointers Estimation

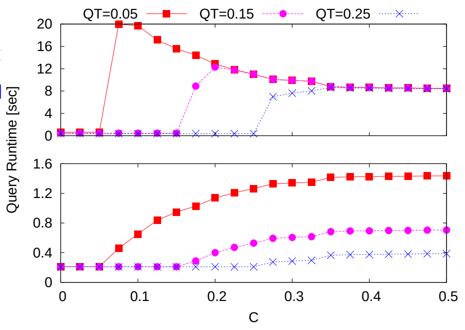

Figure 12: Cutoff Index Cost Model

(20+50 [ms/MB]) and conforms to our cost model.

Finally, we test the query runtime when a UPI has to access a cutoff index, and we verify that our cost model can predict the behavior. We again used Query 1 and varied both QT and C. First, we checked the accuracy of selectivity estimation described in Section 6.1 because our cost model relies on accurate estimates of the number of pointers returned from the cutoff index. Figure 11 compares the true number and the estimated number of cutoff pointers for various QT and C settings (except $QT > C$). The result shows that our selectivity estimation is accurate.

Figure 12 shows the runtimes estimated by our cost model with the exact same setting as Figure 3 in Section 6.3. As the two figures show, our cost model (which estimates disk seek costs and saturation of cutoff pointers using a sigmoid function) matches the real runtime very well for both selective and non-selective queries.

These results above confirm that our cost models can accurately estimate the query costs in various settings. These cost models will be useful for the query optimizer to choose execution plans and for a database administrator or auto tuning program to choose tuning parameters for UPIs.

8. RELATED WORK

The most closely related work to UPIs relates has to do with the use of indices for uncertain data. Some work [4] uses traditional B+Trees to index uncertain data. Other work has shown that a special index can substantially speed up queries over uncertain data. For example, Cheng et al [5] developed the PTI (*Probabilistic Threshold Indexing*) based on R-Trees to speed up PTQs on uncertain attributes with one dimensional continuous distributions. Other research has extended these ideas to higher dimensions (*U-Trees* [16]) and more variable queries (*UI-Trees* [17]). Similarly, Singh et al [13] proposed the PII (*Probabilistic Inverted Index*) for PTQs on uncertain attributes with discrete distributions based on inverted indexes as well as the PDR-tree (*Probabilistic Distribution R-tree*) based on R-Trees.

Although these indexes successfully speed up query execution in some cases, they are essentially secondary indexes and can lead to many random disk seeks when the query needs to retrieve other attributes from the heap file. This problem arises especially when the query is not selective as shown in Section 7. Hence, UPIs complement this prior work by adding support for primary indexes on uncertain attributes, which are particularly useful for analytical PTQs which process thousands or millions of tuples.

9. CONCLUSION AND FUTURE WORK

In this paper, we developed a new primary index for uncertain databases called a UPI. Our empirical results on both discrete and continuous uncertain datasets show that UPIs can perform orders of magnitude faster than prior (secondary) indexing techniques for analytic queries on large, uncertain databases. We proposed several techniques to improve the performance of UPIs, including cutoff indexes to reduce their size, and tailored indexes to improve the performance of secondary indexes built on top of UPIs. We also discussed Fractured UPIs that help handle data updates and eliminate fragmentation, further improving query performance. Finally, we provide accurate cost models to help the query optimizer to choose execution plans and the DBA to select tuning parameters.

As future work, we plan to apply UPIs for queries other than PTQs, especially *Top-k*. Ilyas et al suggested a query processing engine to determine probabilistic top-k answers with a minimal number of tuples extracted from a Tuple Access Layer (TAL) which provides tuples in probability order [14]. A UPI can work as an efficient TAL. One approach is to estimate the minimum probability of tuples required to answer the top-k query and use this probability as a threshold for the UPI. Another approach is to access UPI a few times with decreasing probability thresholds until the answer is produced. Both approaches are promising future work.

Acknowledgments

We thank Newton Ikhariale for his contribution on the fractured index structure. Samuel Madden was supported in part by NSF grant IIS-0448124. Hideaki Kimura and Stan Zdonik were supported in part by the NSF, under the grants IIS-0905553 and IIS-0916691.

10. REFERENCES

[1] www.cse.cuhk.edu.hk/~taoyf/paper/tods07-utree.html.
[2] P. Agarwal, S. Cheng, Y. Tao, and K. Yi. Indexing uncertain data. In *PODS*, 2009.
[3] B. Aleman-Meza, F. Hakimpour, I. B Arpinar, and A. Sheth. SwetoDblp ontology of Computer Science publications. *Web Semantics: Science, Services and Agents on the WWW*, 2007.
[4] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: databases with uncertainty and lineage. In *VLDB*, 2006.
[5] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *VLDB*, 2004.
[6] N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.
[7] N. Ikhariale. Fractured Indexes: Improved B-trees To Reduce Maintenance Cost And Fragmentation. Master's thesis, Brown University, 2010.
[8] C. Jermaine, E. Omiecinski, and W. Yee. The partitioned exponential file for database storage management. *VLDB J.*, 2007.
[9] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. Correlation Maps: a compressed access method for exploiting soft functional dependencies. *VLDB*, 2009.
[10] M. Ley. DBLP - Some Lessons Learned. *PVLDB*, 2009.
[11] I. Nagy, R. Farkas, and M. Jelasity. Researcher affiliation extraction from homepages. *ACL-IJCNLP 2009*, page 1, 2009.
[12] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
[13] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. Hambrusch. Indexing Uncertain Categorical Data. In *Proc. ICDE*, 2007.
[14] M. Soliman, I. Ilyas, and K. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.
[15] D. Suciu. Database theory column: Probabilistic databases. *SIGACT News*, 39(2):111–124, 2008.
[16] Y. Tao, R. Cheng, X. Xiao, W. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *VLDB*, 2005.
[17] Y. Zhang, X. Lin, W. Zhang, J. Wang, and Q. Lin. Effectively Indexing the Uncertain Space. *TKDE*, 2010.