

Implementing NOT EXISTS Predicates over a Probabilistic Database*

Ting-You Wang**, Christopher Re, and Dan Suciu

University of Washington

Abstract. Systems for managing uncertain data need to support queries with negated subgoals, which are typically expressed in SQL through the NOT EXISTS predicate. For example, the user of an RFID tracking system may want to find all RFID tags (people or objects) that have traveled from a point A to a point C without going through a point D. Such queries are difficult to support in a probabilistic database management system, because offending tuples do not necessarily disqualify an answer, but only decrease its probability. In this paper, we present an approach for supporting queries with NOT EXISTS in a probabilistic database management system, by leveraging the existing query processing infrastructure. Our approach is to break up the query into multiple, monotone queries, which can be evaluated in the current system, then to combine their probabilities by addition and subtraction to compute that of the original query. We will also describe how this technique was integrated with MystiQ, and how we incorporated the top-k multi-simulation and safe-plans optimizations.

1 Introduction

Probabilistic databases have been used recently in a variety of applications of uncertain data: in acquisition systems such as sensor nets or RFID deployments [5, 9, 10, 15], to manage data extracted by information extraction systems [8], to query incompletely cleaned data [1], data obtained as a result of imprecise data mappings [7], and data that has been anonymized to protect privacy [12].

The research on query processing on probabilistic databases has led to techniques for processing select-project-join queries [4, 3], queries on data with explicit provenance [2], on Markov Network data [16], top-k queries [13], and queries with *having*-predicates [14].

An area that has not been explored so far are queries with negated subgoals, which, in SQL, can be expressed using the NOT EXISTS predicate. Such queries are an important piece of managing uncertain data. For example, in an RFID application a user may want to find all events when an RFID tag has traveled from point A to point C without going through a point D; in large scale information extraction systems s.a. DBLive [6] a user may want to find all database

* This work was partially supported by NSF grants IIS-0513877 and IIS-0713576.

** Corresponding author: tingyouuw@gmail.com.

researchers that have never served on a VLDB committee (e.g. when a PC chair forming a Program Committee is searching for junior researchers that have never served on the PC before), or a PC chair may want to find all authors that have never published before in a database conference (a much needed query when selecting the *best newcomer award*).

Yet queries with negated subgoals are difficult to compute on a probabilistic database. In a standard (deterministic) database, if at least one witness tuple satisfies the negated subgoal then the answer is immediately disqualified. But in a probabilistic database such a witness does not immediately disqualify the answer, only reduces its probability. In fact, if there are many witnesses then their probabilities need to be aggregated in order to compute by how much to reduce the probability of the answer.

In this paper we describe a technique for computing SQL queries with NOT EXISTS predicates on a probabilistic database, which we have implemented on top of MystiQ, a probabilistic database system [11]. Our approach splits a query with a conjunction of NOT EXISTS as its predicate into multiple parts: a query to represent the positive results, and many queries to capture the not exists subgoals. Importantly, these queries are all simple select-from-where queries, which can be evaluated using the existing MystiQ infrastructure. Then, we combine their results and compute the probability of each output tuple by taking into account the semantics of the NOT EXISTS predicate. We also describe how to incorporate in this approach two optimizations present in MystiQ: top k-query evaluation and safe subplans.

Motivating application: Logging Activities The RFID Ecosystem project at the University of Washington [17] deploys about 132 antennas throughout the department’s building and tracks thousands of RFID tags attached to objects or carried by people. As these tagged people/objects roam through the hallways, a database table is populated with their movements. In a perfect world, the table would contain exactly every instance a tagged person passes by an antenna. However, this is not the case. It turns out that there are many times that readings are dropped, which are called false negatives, and other times, extra readings are registered, which are identified as false positives. As a consequence, every reading recorded in the database is probabilistic. (We refer the reader to [15] for a full description of the probabilistic model.)



Fig. 1. Person is walking down the hall starting at A

For example, consider the diagram in Fig. 1 that shows the position of three antennas A, B, C (out of a few dozens), and suppose that we are interested in retrieving all timestamps when a user has walked from A to C going through B .

The antennas read at a frequency of four readings per second. False negatives are common (missed tags) and false positives are also frequent (readings from nearby antennas). Our system converts these uncertainties into probabilities, and as a consequence might record a sequence of readings like this:

$$A_1 A_2 B_3 A_4 A_5 B_6 B_7 B_8 C_9 C_{10} B_{11} C_{12} \dots A_{21} A_{22} B_{23} A_{24} D_{25} D_{26} D_{27} E_{28} E_{29} C_{30}$$

If the database were deterministic then (A_5, C_9) is the only answer: we don't consider (A_1, C_9) because there are intermediate readings at A , and we don't consider (A_{24}, C_{30}) an answer because obviously the user has taken a different route to get from A to C , other than through B . Our query is expressed by the following SQL statement which finds all pairs for readings (A, C) for which there exists no intermediate readings other than those at antenna B :

```
SELECT distinct r1.pid, r1.time, r2.time
FROM Data r1, Data r2
WHERE r1.time < r2.time AND r1.pid = <UserID> AND
      r2.pid = r1.pid AND
      r1.antenna = 'A' AND r2.antenna = 'C' AND
      NOT EXISTS (SELECT distinct *
                  FROM Data r3
                  WHERE r3.pid = r1.pid AND r3.time > r1.time
                  AND r3.time < r2.time
                  AND r3.antenna != 'B')
```

Evaluating this query on a probabilistic database poses important technical challenges. It is no longer the case that (A_5, C_9) is the single answer. For example (A_1, C_9) may also be an answer, namely when the offending readings A_2, A_4, A_5 are false positives. In fact the probability of the event (A_1, C_9) is $(1 - p_2)(1 - p_4)(1 - p_5)$ (we denote p_2 the probability of the event A_2 , etc: these values are stored in the database). Similarly, (A_{24}, C_{30}) is now also a possible answer, and its probability is that of all the offending witnesses being absent.

Paper Organization We start by describing the basic data model in Sec. 2. We describe the main idea in Sec. 3, then provide the general approach in Sec. 4. We describe some implementation details in Sec. 5 and some preliminary experiments in Sec. 6. We conclude in Sec. 7.

2 Data and Query Model

We briefly describe MystiQ's query and data model from [3]. The relations are independent/disjoint probabilistic relations, in which every two tuples are either independent, or disjoint events. Queries are expressed in SQL, and each answer is annotated by a probability representing the system's confidence in that answer: the answers are typically presented to the user in decreasing order of their output probability, see Fig. 2. MystiQ supports SELECT-DISTINCT-FROM-WHERE

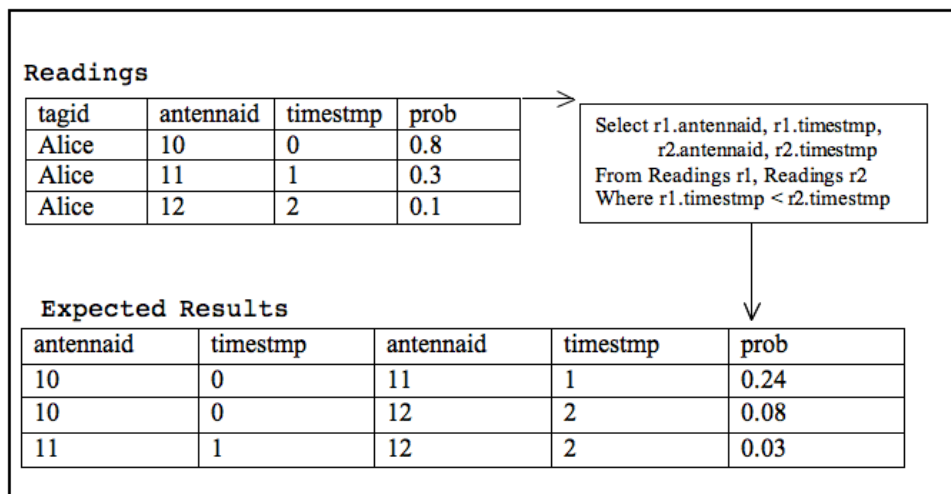


Fig. 2. An example of a query for MystiQ

queries and uses two algorithms to compute probabilities: *safe plans* [4] and *multisimulation* [13].

The first algorithm translates a SQL query Q into another query Q' that manipulates probabilities explicitly: it multiplies or adds them, or computes their complement $(1 - p)$. When such a rewriting is possible then Q is called a *safe query* and Q' is called a *safe plan*. The relational database server executes Q' and returns tuples ordered decreasingly by their probabilities: MystiQ will display them without any further processing.

The second algorithm is used when Q is not safe. In this case MystiQ runs a much simpler query on the database engine, but needs to do considerably more work to compute the probabilities. The database engine simply returns all possible answer tuples t_1, \dots, t_n together with their *lineage* [2]: it does not compute any output probabilities. The lineage of a tuple is a positive DNF formula whose propositional variables are tuples from the input database: importantly, these input tuples carry with them the input probability. The probability of each tuple t_i is precisely the probability of its associated DNF formula, and MystiQ evaluates these probabilities p_1, p_2, \dots, p_n by running n Monte Carlo simulation algorithms. This is an expensive process, and here MystiQ uses an optimization called *multisimulation*, introduced in [13], whose goal is to run the smallest number of simulation steps required to identify and rank only the top k tuples, ordered decreasingly by their probabilities. We describe here how the multisimulation algorithm identifies the top k tuples, without necessarily sorting them, and refer the reader to [13] for details. Suppose that at some point we have a confidence interval for each tuple: $[a_1, b_1], [a_2, b_2], \dots, [a_n, b_n]$, s.t. it is guaranteed that $p_i \in [a_i, b_i]$, for $i = 1, \dots, n$. (Initially, $a_1 = a_2 = \dots = 0$ and $b_1 = b_2 = \dots = 1$.) The crux of the algorithm consists of choosing which DNF

formula to simulate next. Let c be the k 'th largest value in $\{a_1, \dots, a_n\}$ and let d be the $k + 1$ 'st largest value in $\{b_1, \dots, b_n\}$. The first observation is that if $d < c$ then the algorithm has succeeded in identifying the top k tuples: these are precisely the k tuples t_i s.t. $c \leq a_i$ since for any other tuple t_j not in this set it is the case that $b_j \leq d$, hence $p_j < p_i$. If $c \leq d$, then the interval $[c, d]$ is called *the critical region*: the second observation is that in this case one must improve the confidence interval $[a_i, b_i]$ of some tuple that contains the critical region: $a_i \leq c < d \leq b_i$. Thus, the algorithm chooses one of the tuples t_i whose current confidence interval $[a_i, b_i]$ "crosses" the critical region (hence t_i is called a *crosser*), then runs a few simulation steps on the DNF formula for t_i , which further shrinks the interval $[a_i, b_i]$ and then re-computes the critical region c, d . It stops when $d < c$.

In summary, MystiQ tries to evaluate a safe query, when possible, or runs the multisimulation algorithm otherwise.

3 Evaluating Queries with a Single NOT EXISTS

In this work we considered SQL queries that have a conjunction of NOT EXISTS in the **where** clause; the nested queries are in turn simple **select-from-where** queries, i.e. we do not allow nested NOT EXISTS. We describe in this section our approach, and we start with the simple case of a single nested NOT EXISTS query. We use the following as our running example:

```
Original query Q:
  SELECT distinct col_1, col_2, col_3
  FROM R1, R2, R3
  WHERE condition1 AND NOT EXISTS
    ( SELECT *
      FROM R'1, R'2, R'3
      WHERE condition2 )
```

From Q, we derive the following two queries. The *outer query* is the query Q with the NOT EXISTS predicate removed:

```
Outer query Q1:
  SELECT distinct col_1, col_2, col_3
  FROM R1, R2, R3
  WHERE condition1
```

The *inner query* is the query under the NOT EXISTS predicate:

```
Inner query Q2:
  SELECT distinct col_1, col_2, col_3
  FROM R1, R2, R3, R'1, R'2, R'3
  WHERE condition1 AND condition2
```

Let E_1 be the event that a tuple t is in the answer to Q_1 , and E_2 be the event that t is in the answer to Q_2 . Then the answers to Q are precisely the tuples satisfying the event $E_1\overline{E_2}$ and their probability is:

$$P(t \in Q) = P(E_1) - P(E_1E_2) = P(E_1) - P(E_2)$$

The last equality holds because every tuple that is an answer to Q_2 is also an answer to Q_1 .

In summary, to evaluate Q on a probabilistic database we proceed as follows. First we evaluate Q_1 and obtain a set of tuples t_1, \dots, t_n with probabilities p_1, \dots, p_n . Next, we evaluate Q_2 : we assume its answer is the same list of tuples, with probabilities p'_1, \dots, p'_n : if Q_2 returns some tuple that is not among t_1, \dots, t_n then we can ignore it, and conversely, if it does not return some tuple t_i we simply add it and set its probability $p'_i = 0$. The answer to the query Q is the list of tuples t_1, \dots, t_n , and their probabilities are $p_1 - p'_1, \dots, p_n - p'_n$.

4 Evaluating Queries with Multiple NOT EXISTS

We now show how to generalize to multiple NOT EXISTS predicates. Our running example is:

```
Q:
SELECT DISTINCT col_1, col_2, col_3
FROM A_1, B_1, C_1
WHERE condition_1 AND NOT EXISTS
  (SELECT *
   FROM A_2, B_2, C_2
   WHERE condition_2)
AND
  ...
AND NOT EXISTS
  (SELECT *
   FROM A_m, B_m, C_m
   WHERE condition_m)
```

As before we define an *outer query* Q_1 , and several *inner queries* Q_2, Q_3, \dots, Q_m ; their definitions are by a straightforward generalization of those in the previous section.

Let E_i be the event that a tuple t is in the answer set to the query Q_i . Then, the event “ t is in the answer to Q ” is equivalent to:

$$(t \in Q) \equiv E_1\overline{E_2}\dots\overline{E_k}\dots\overline{E_m}$$

Therefore, by using the inclusion-exclusion formula we derive:

$$\begin{aligned}
P(t \in Q) &= P(\overline{E_1} \overline{E_2} \dots \overline{E_k} \dots \overline{E_m}) \\
&= P(E_1) - P(E_1(E_2 \vee E_3 \vee \dots \vee E_m)) \\
&= P(E_1) - \sum_{2 \leq i_2 \leq m} P(E_1 E_{i_2}) + \sum_{2 \leq i_2 < i_3 \leq m} P(E_1 E_{i_2} E_{i_3}) - \dots - (-1)^m P(E_1 E_2 \dots E_m)
\end{aligned}$$

Thus, in order to evaluate the query Q we proceed as follows. We start by evaluating the query Q_1 : suppose it returns n tuples, t_1, \dots, t_n , with probabilities p_1, \dots, p_n . Next we evaluate the following $2^{n-1} - 1$ queries: $Q_{i_2} \dots Q_{i_k}$ for $k \geq 1$ and $1 \leq i_2 < \dots < i_k \leq n$. (Note that $Q_1 Q_{i_2} \dots Q_{i_k}$ is equivalent to $Q_{i_2} \dots Q_{i_k}$ when $k > 0$.) Assume that each of these queries returns the same set of tuples t_1, \dots, t_n , with some probabilities. (Any additional tuples can be removed, and any missing tuples can be added with probability 0.) Then the answer to Q consists of the tuples t_1, \dots, t_n , and the probability of t_i is obtained by summing the 2^{n-1} probabilities of t_i in all queries $Q_1 Q_{i_2} \dots Q_{i_k}$, with signs $(-1)^{k+1}$.

We end this section with a note on our algorithm for generating a list of 2^{n-1} queries that need to be evaluated. This process is briefly demonstrated in Fig. 3. We maintain a list of already generated queries and then appending onto the list by merging new nested queries with those already in the list. However, the order of the appending is very important; it must be from the back of the list to the front in order to create the alternating property that we desire. With this we are ready to begin the implementation process for NOT EXISTS.

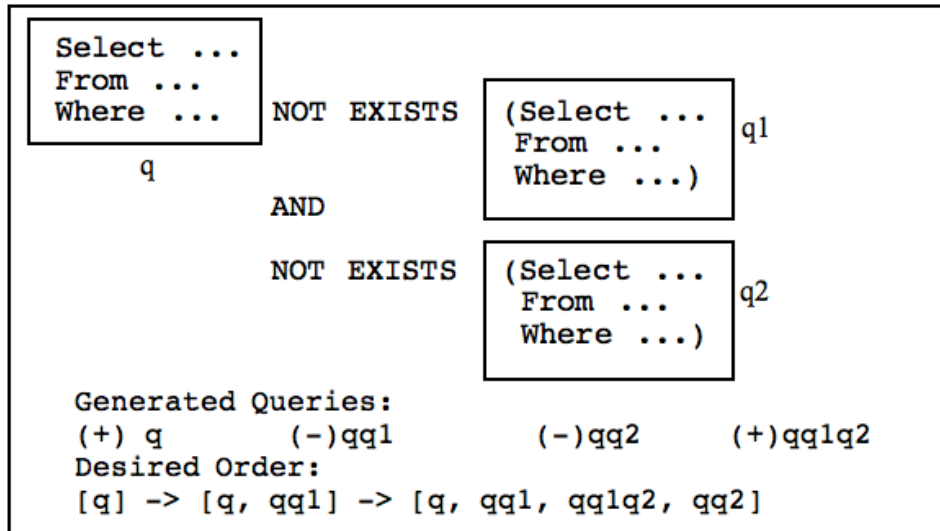


Fig. 3. Example of how a list of generated queries is populated

5 Implementing NOT EXISTS

It is clear that the first step of implementing NOT EXISTS is to parse a query and recognize the NOT EXISTS clauses. Finding the clauses is straightforward since it is a standard traversal of a tree generated to represent the input query. But once these locations are found, we cannot immediately begin the generation of queries because there are two main issues that must be resolved. First, the naming of table aliases could conflict between the outer and inner queries. Second, we need to generate the queries in the correct order to produce the alternating in signs that we desire for the probabilities.

The first problem requires a renaming that is done at the moment an input query is parsed. For every table alias that is given, we will append on a suffix to make the name unique from all others. The suffix chosen for MystiQ was `_X` where X is a non-negative integer. An example of this renaming is shown below:

```
SELECT R1.attr1, R1.attr2, R2.attr1
FROM Relation R1, Relation R2
WHERE R1.attr1 = R2.attr2 and R1.attr2 = 'in' AND NOT EXISTS
  (SELECT *
   FROM Relation R1
   WHERE R1.attr1 = R2.attr2 and R1.attr2 = 'out')
```

TRANSLATES INTO:

```
SELECT R1_1.attr1, R1_1.attr2, R2_2.attr1
FROM Relation R1_1, Relation R2_2
WHERE R1_1.attr1 = R2_2.attr2 and R1_1.attr2 = 'in' and NOT EXISTS
  (SELECT *
   FROM Relation R1_3
   WHERE R1_3.attr1 = R2_2.attr2 and R1_3.attr2 = 'out')
```

The second problem is much more simple to solve. In order to generate the new queries in the correct order, we first make a pass over the input query and extract/remove all the inner queries. At the end of this process, we have a list of all the nested queries and the outer query we need. For the first of the nested queries, we merge it with the outer query, then we take another nested query, if it exists, and merge it with the first two generated queries in reverse order (just as Fig 3 demonstrated). We proceed in this fashion for all nested queries. After this process is done, we are ready to begin calculating probabilities.

Calculation of probabilities has two pathways that it can take. The first is to compute the probabilities explicitly, taking advantage of the safe plans optimization. The second is to use the multi-simulation method. Clearly, explicitly calculating the probability is much more desirable, so unless a user specifies that simulations must be run, MystiQ will check if all of the generated queries, which are all monotone, can be computed exactly. If so, we can compute the NOT EXISTS query by substituting the probability values directly into the formulas discussed in Sections 3 and 4 to get an exact solution.

In a simulation, we no longer have exact probabilities to work with, but rather entire sets of confidence intervals, one set for each of the generated queries. Recall from Section 2, an *interval* represents a range in which the true probability lies for a particular result of the query. As mentioned in Section 4, the possible tuples of the original query are precisely those of the outer query. However, there is the additional complication of the dependency between the intervals for the original query and those of the generated queries. Since we never actually ran the original query we have no DNF formula to simulate, so we must use the other intervals.

The endpoints for a tuple of the original query is equivalent to just finding the extreme points of the probability formula. For example, if one was computing the lower endpoint of an interval, they would be computing the smallest possible probability value that could result from substituting the probabilities of the equivalent tuples of the generated queries. Similarly, the upper endpoint would be the largest possible probability value. Figure 4 provides a demonstration of the actual calculation and equations.

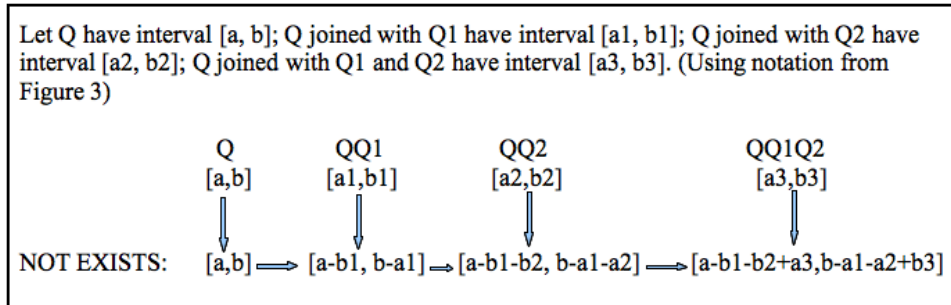


Fig. 4. Example of how intervals are calculated

With this key issue resolved, the simulation process can proceed in much the same way as it did in the original implementation of *MystiQ*, with only a slight modification. The process begins with initializing a set of intervals (to represent the original query), S , to the intervals of the outer query. Then, one interval, $I \in S$, is chosen (exactly in the same way an interval would have been chosen before). However, this interval itself is not simulated. Rather, all intervals representing the same tuple from the generated queries are simulated. These associated intervals are then aggregated together using the method just described before to come up with the new interval for our original query. This process is repeated until a certain number of intervals are isolated from each other giving us the top-k tuples.

6 Experiments

Experiments were run in order to judge how well the implementation performed when adjusting various parameters. For starters, we adjusted database table sizes and toggled using safe plans. Next, we varied the number of Monte Carlo simulations run every step in the multisimulation algorithm. Finally, we experimented with how the query system handles more complex queries.

First, experiments were run to test how the implementation performed over different data sizes. For this, we used a synthetic database (rows are generated with 'random' probabilities) consisting of `Products` and `Orders`, and ran the following query:

```
SELECT DISTINCT p.id, p.name
FROM ProductEvent p
WHERE NOT EXISTS(
    SELECT DISTINCT *
    FROM OrderEvent o
    WHERE o.productid = p.id and o.price > 10)
```

We varied the data set size (split almost equally between `Products` and `Orders`): ~4000, ~9500, ~14000, ~19000, ~28500 rows. We also ran this query, both, with the safe plan optimizations and without. Fig. 5 shows the results of those runs.

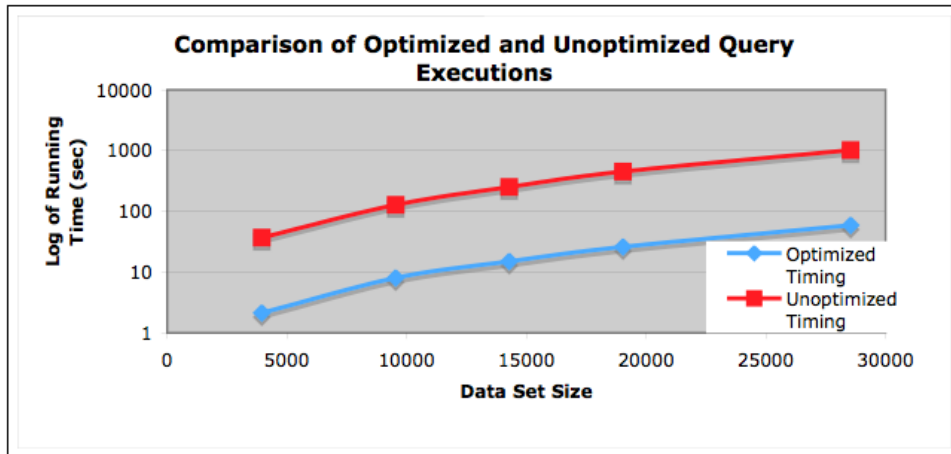


Fig. 5. Comparison of optimized and unoptimized queries

From this graph, we see that the running times of NOT EXISTS queries exponentially rise as the data set size increases. However, as the size increases, the slopes do seem to grow shallower. We also see in the graph that safe plan optimizations make a significant improvement in performance. From the tests,

there was generally a 94% improvement in time when optimizations could be found.

Another experiment examined how changing the number of simulations run every step in the simulating process of MystiQ affected the running time. The graph in Fig. 6 charts the change in times as we varied the number of simulations from 10000 to 70000 on a fixed data set of about 10000 rows, about 5000 products and 5000 orders (we used the same query as before).

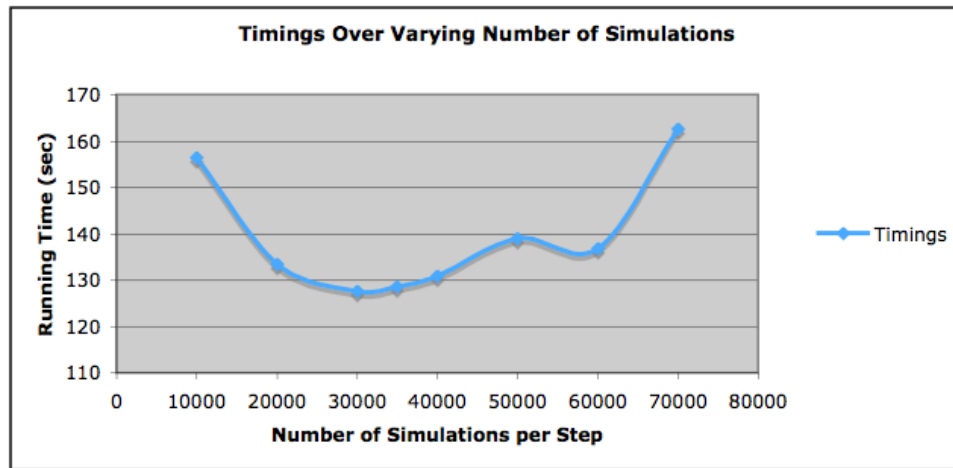


Fig. 6. Execution times against changing the number of simulations per step

We see that for both small and large values, the running times dramatically increase; however, the cause is very different. For the smaller values, the increase in running time is mostly due to the fact that many more steps have to run to reach enough simulations on the intervals to isolate the top k. For the larger values, the increase in time is due to the fact that, though we reduce the number of steps run, we are over simulating the intervals. The extra simulations are simply taking up time and not adding any new information. The middle numbers, particularly at around 30000, balance the two problems well. They run enough simulations to reduce the number of steps taken, but they do not over simulate the intervals.

Although Fig. 6 showed 30000 to be the optimal value for the **Number of Simulations per Step**, this is only for this particular case. Different queries will have different results. However, even in those cases, the graph of times ought to follow the same curve since these queries face the same problems, only it will be translated along the x-axis.

Lastly, experiments were performed on gathered data closely related to the RFID data (times when a person entered and left a room). A complex query was written that searched for consecutive events of entering and leaving a room

for a particular person, but the data set size was only 660 tuples (split between entering and leaving rooms). The query, at first, used two NOT EXISTS queries. Here is an example of the query:

```
SELECT DISTINCT er.tag_id, er.room_num, er.timestamp, lr.timestamp
FROM EnteredRoomEvent er, LeftRoomEvent lr
WHERE er.tag_id = lr.tag_id and er.room_num =
      lr.room_num and er.timestamp < lr.timestamp AND NOT EXISTS
      (SELECT DISTINCT *
       FROM EnteredRoomEvent e3
       WHERE e3.tag_id = er.tag_id and e3.timestamp >
            er.timestamp and e3.timestamp < lr.timestamp) AND NOT EXISTS
      (SELECT DISTINCT *
       FROM LeftRoomEvent e4
       WHERE e4.tag_id = er.tag_id and e4.timestamp >
            er.timestamp and e4.timestamp < lr.timestamp)
```

Then the query was translated into a single NOT EXISTS query to see how timings are affected. This new query is found below (where the AllSightings table is simply the union of EnteredRoom/LeftRoom events).

```
SELECT DISTINCT er.tag_id, er.room_num, er.timestamp, lr.timestamp
FROM EnteredRoomEvent er, LeftRoomEvent lr
WHERE er.tag_id = lr.tag_id and er.room_num =
      lr.room_num and er.timestamp < lr.timestamp AND NOT EXISTS
      (SELECT distinct *
       FROM AllSightings all
       WHERE all.tag_id = er.tag_id and all.timestamp >
            er.timestamp and all.timestamp < lr.timestamp)
```

The results of the test showed that it is about 25 times slower when adding one additional level of complexity. With only one NOT EXISTS sub query, results were found in 51.062 seconds where as by adding one more level of complexity, it took 1260.389 seconds. This increase comes from executing twice the number queries and running simulations on twice the number of sets of intervals.

In addition to the immediate results of the experiment, another observation was made. An issue arises when there are many probabilities that lie very close together, particularly when they are all close to zero (which was the case for this experiment). While executions were run, there was a clear slow down when all the intervals narrowed and were centered about zero. At this point, the simulator makes only small changes to an interval's endpoints leading to longer execution to distinguish a top k set of intervals. This is more prevalent in NOT EXISTS queries since using the NOT EXISTS clause, we can more easily send more probabilities closer to zero, but if certain monotone queries were chosen carefully over data sets with very small probabilities, it would also be possible for this case to arise.

7 Conclusion

Systems for managing uncertain data need to support queries with negated subgoals, such as SQL queries with NOT EXISTS predicates. We have described in this paper an approach for supporting such queries, while leveraging much of the infrastructure of an existing probabilistic database management system. We have described optimizations, which in our preliminary experiments show improvement of performance by about 94%. However, the execution time still is fairly slow and can be improved upon. As our experiments show, the running times are seemingly exponential with respect to the data size. Also, as the complexity of the query increases, there is also a significant increase in running time. Further improvements in any of these areas would be extremely beneficial to making the system more practical to use in many different environments.

Possible directions for future work would be to improve the simulation process or improving the interpretation of a NOT EXISTS query. Some specific areas that can be improved include implementing a faster way of running Monte Carlo steps, or understanding how to better choose intervals to simulate, or even figuring out how to choose the number of steps to take each time we simulate an interval. Other work could be to find better ways to break apart a NOT EXISTS query so that we no longer have an exponential number of queries. Unless we get away from the exponential increase in generated queries, we can always expect that there will probably be an exponential increase in time when more NOT EXISTS queries are added.

References

1. P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. In *ICDE*, 2006.
2. O. Benjelloun, A. D. Sarma, C. Hayworth, and J. Widom. An introduction to ULDBs and the Trio system. *IEEE Data Eng. Bull.*, 29(1):5–16, 2006.
3. N. Dalvi, C. Re, and D. Suciu. Query evaluation on probabilistic databases. *IEEE Data Engineering Bulletin*, 29(1):25–31, 2006.
4. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, Toronto, Canada, 2004.
5. A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Using probabilistic models for data management in acquisitional environments. In *CIDR*, pages 317–328, 2005.
6. A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen. Community information management. *IEEE Data Engineering Bulletin, Special Issue on Probabilistic Data Management*, 29(1):64–72, March 2006.
7. X. Dong, A. Halevy, and C. Yu. Data integration with uncertainty. In *VLDB*, 2007.
8. R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, pages 965–976, 2006.
9. T. Jayram, S. Kale, and E. Vee. Efficient aggregation algorithms for probabilistic data. In *SODA*, 2007.

10. S. Jeffery, M. Garofalakis, and M. Franklin. Adaptive cleaning for RFID data streams. In *VLDB*, pages 163–174, 2006.
11. MystiQ: a probabilistic database system, available at <http://mystiq.cs.washington.edu/>.
12. V. Rastogi, D. Suciu, and S. Hong. The boundary between privacy and utility in data publishing. In *VLDB*, 2007.
13. C. Re, N. Dalvi, and D. Suciu. Efficient Top-k query evaluation on probabilistic data. In *ICDE*, 2007.
14. C. Re and D. Suciu. Efficient evaluation of having queries on a probabilistic database. In *Proceedings of DBPL*, 2007.
15. C. Ré, J. Letchner, M. Balazinska, and D. Suciu. Extracting events from correlated streams. In *SIGMOD*, page to appear, 2008.
16. P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.
17. E. e. a. Welbourne. Challenges for pervasive RFID-based infrastructures. In *PERTEC Workshop*, March 2007.