Implementing NOT EXISTS Predicates over a Probabilistic Database

By: Ting-You Wang

December 3, 2007

Abstract: Many SQL queries over data collected from RFID devices use the NOT-EXISTS predicate, but such queries are not currently supported by the probabilistic database system, MystiQ. This work consists of an extension of MystiQ that supports SQL queries with the NOT-EXISTS predicates. This is a non-trivial extension, because NOT-EXISTS is non-monotone, while all previous queries supported by MystiQ were monotone, making the query probabilities more difficult to compute. The approach described is to break up the query into multiple, monotone queries, and combine their probabilities by addition and subtraction to compute that of the original query. We will also describe how this technique was integrated with the rest of the MystiQ system, including top-k multi-simulation and safe-plans optimizations.
When most people hear about querying databases, they think of discrete tuples in a table with no consideration of a tuple's validity. But, in reality, there are many situations where an imperfect table could exist. One such example is in the area of Radio Frequency Identification (RFID). Here, data is collected via antennas and identification tags. But, these pieces of equipment are not perfect. There are instances where the table of recorded readings contains errors. These errors lead to the need of putting probabilities into the table in an attempt to make them more complete, and, as one would suspect, there needs to be an application that would make sense of querying a table with probabilities. Such a program was developed, but only worked for monotone queries. And as queries for RFID data began to take shape, a clear need for using the NOT EXISTS clause became apparent. Extending the probabilistic querying program, called MystiQ, to accommodate non-monotone queries turned out to be a non-trivial problem. In this paper, we will discuss the RFID system in more detail, the MystiQ application, and the implementation of NOT EXISTS in this MystiQ.

Logging Activities

The RFID project takes place throughout the hallways of the UW Computer Science building. Multiple antennas are set up in these hallways and people will wear RFID tags that will register with these antennas. There are 132 antennas with about twenty-six of them on five floors. As these tagged people roam through the hallways, a database table is populated with the movements of each person. 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 a false negative, and other, less frequent times that extra readings are registered, which are identified as false positives. For example, consider the following diagram:

![Diagram of person walking down the hall](image)

**Figure 1: Person is walking down the hall starting at A**

A person is walking down the hall from point A to point C. We would suspect that there would be readings at point A, then at point B then at point C. However, the false negatives occur when we see a reading at A, then it is immediately followed by a reading at C. That is, antenna B simply did not read the id tag of the person. On the other hand, a false positive could have a reading at A, then B, then suddenly two readings, one at A and one at C. One occasion in which this occurs is when the antennas for A and C are too close to B so that when a person stands at B, we also get readings for both A and C. How do we know where the person is? He/she could have stopped at B and turned around, or he/she could have been continuing on to C and the A was an accidental reading.

To accommodate for these uncertainties, probabilities were attached to the tuples in the table. These probabilities are added manually using the following criteria: raw data in the table, minus the false positives, have probability one, and new tuples with probabilities less than one are inserted to fix false negatives. False positives also had probabilities less than one. These probabilities are usually learned from previous readings that have coherent sequences of movement (A to B to C). Based on the times that it took for someone to go from one point to another, such as from A to B then to C, we can approximate when to insert tuples for missing readings at intermediate points, such as B.

With this new probabilistic database, we can make interesting queries about the movement of objects or people. For example, one can ask who was in a particular meeting at a specified time. The
results would include those people whose readings show them in the room with associated probabilities of how confident those result are. But, to make such queries, we soon noticed a common pattern with these queries—they all include NOT EXISTS. A common query would look something like the following:

\[
\text{SELECT distinct } r1.pid, r1.time, r2.time \\
\text{FROM Data r1, Data r2} \\
\text{WHERE r1.time < r2.time AND r1.pid = <UserID> AND} \\
r2.pid = r1.pid \text{ AND NOT EXISTS (SELECT distinct *} \\
\text{FROM Data r3} \\
\text{WHERE r3.pid = r1.pid AND r3.time > r1.time} \\
\text{AND r3.time < r2.time)}
\]

This query simply attempts to find two readings that are adjacent, in time, to each other. This is done by making sure that there is no third reading that lies between two readings increasing in time. Without the NOT EXISTS clause we could end up with a reading from one day and another from a completely new day, which is not an accurate portrayal of a person's movement.

We already had a program called MystiQ that could perform monotone queries on probabilistic databases. A monotone query is a single SELECT-FROM-WHERE query without any nested queries within it. So the next step was clear, we would need to extend MystiQ to run non-monotone queries, more specifically, NOT EXISTS clauses.

**MystiQ**

MystiQ takes probabilistic databases and run queries over them, calculating probabilities for resulting tuples using basic laws of probability. For example,

```
<table>
<thead>
<tr>
<th>tagid</th>
<th>antennaid</th>
<th>timestamp</th>
<th>prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>10</td>
<td>0</td>
<td>0.8</td>
</tr>
<tr>
<td>Alice</td>
<td>11</td>
<td>1</td>
<td>0.3</td>
</tr>
<tr>
<td>Alice</td>
<td>12</td>
<td>2</td>
<td>0.1</td>
</tr>
</tbody>
</table>
```

```
Select r1.antennaid, r1.timestamp, r2.antennaid, r2.timestamp \\
From Readings r1, Readings r2 \\
Where r1.timestamp < r2.timestamp
```

**Figure 2: An example query for MystiQ**

However, not all queries can be computed as easily as this example. There are occasions where queries become complex enough that there is no way to rewrite the query to compute the probability directly.
These times require probabilistic simulations to take place.

On a high level, simulations run by first sending an input query to the database and gets the resulting table from the database. Then, for each one of the tuples in the return table, an interval is created to represent the range in which the true probability lies. As one would suspect, at the start, these intervals are $[0, 1]$. Once these intervals are set up, we proceed to select an interval through a predefined criterion. This criterion attempts to choose intervals in such a way, that the top K highest probabilities are isolated efficiently. Once the interval is chose, we perform a specified number of simulations on this interval. The details of how the simulation works is beyond the scope of this paper. However, it has been proven by Re, Dalvi, Suciu in the paper: Efficient Top-k Query Evaluation on Probabilistic Data[1], that when these simulations run, we are guaranteed that the range interval decreases in length (lower end point increases and/or upper end point decreases). Hence, by running enough simulations on the different resulting tuples, we can isolate the probabilities from each other and list out the top results using the ranges' midpoints as approximate probabilities.

In this paper we describe how we modified the existing code of MystiQ to support non-monotone queries with NOT EXISTS.

Calculating NOT EXISTS

The first step to developing an implementation of NOT EXISTS is to come up with the theoretical solution to the problem. So, we start with the simple case of a single nested NOT EXISTS query:

Original query Q:
\[
\text{Select distinct col}_1, \text{col}_2, \text{col}_3 \\
\text{From R1, R2, R3} \\
\text{Where condition1 AND NOT EXISTS} \\
( \\
\text{Select *} \\
\text{From R'1, R'2, R'3} \\
\text{Where condition2} \\
\)
\]

By definition, we the results of the query are those tuples that match the outer query (the query with the NOT EXISTS removed), but do not match the nested query (everything in the parenthesized expression). This is also the exact probabilistic predicate that we wish to solve. In order to compute such a probability, both parts of the query must be isolated. Hence, we break apart the query into the following two parts:

First query Q1:
\[
\text{Select distinct col}_1, \text{col}_2, \text{col}_3 \\
\text{From R1, R2, R3} \\
\text{Where condition1}
\]

Second query Q2:
\[
\text{Select distinct col}_1, \text{col}_2, \text{col}_3 \\
\text{From R1, R2, R3, R'1, R'2, R'3} \\
\text{Where condition1 AND condition2}
\]
We shall refer to queries like Q2 as 'the outer and inner query merged together'. More specifically, we maintain the same SELECT clause from the outer query, but we combine the FROM and WHERE clauses together. With these two queries, we can find the probability that a tuple x is a result of Q by the following equation:

\[
\text{Let } A = \{ \text{x is in } Q \} \text{, } B = \{ \text{x is in } Q_1 \} \text{, } C = \{ \text{x is in } Q_2 \} \\
\text{THEN: } \\
P(A) = P(B \text{ and } C') = P(B) - P(B \text{ and } C) = P(B) - P(C) \quad (1)
\]

Here, we shall note that the columns that are selected in the original query are subject to the constraint that if they are to be used in the NOT EXISTS clause, then they must be selected. This is needed because tuples inside the database are not necessarily mutually exclusive or independent.

A Further Extension to Multiple NOT EXISTS:

We now want to see how we can generalize our previous approach to accept a conjunction of NOT EXISTS clauses. For example, consider the query:

\[
Q: \\
\text{Select distinct col}_1, \text{ col}_2, \text{ col}_3 \ldots \text{ From R1, R2, R3} \\
\text{Where condition1 AND NOT EXISTS} \\
( \\
\text{Select * } \\
\text{From R'1, R'2, R'3} \\
\text{Where condition2} \\
) \text{ AND NOT EXISTS} \\
( \\
\text{Select * } \\
\text{From R''1, R''2, R''3} \\
\text{Where condition3} \\
)
\]

Here, we are still under the same constraint that whenever a column is used in any of the NOT EXISTS queries, then they must be selected. To understand this query, let us break the query up in a similar fashion as we did above:

First query Q1: 
\[
\text{Select distinct col}_1, \text{ col}_2, \text{ col}_3 \text{ From R1, R2, R3} \\
\text{Where condition1}
\]

Second query Q2: 
\[
\text{Select distinct col}_1, \text{ col}_2, \text{ col}_3 \text{ From R1, R2, R3, R'1, R'2, R'3} \\
\text{Where condition1 AND condition2}
\]

Third query Q3: 
\[
\text{Select distinct col}_1, \text{ col}_2, \text{ col}_3 \text{ From R1, R2, R3, R''1, R''2, R''3} \\
\text{Where condition1 AND condition3}
\]
Now, observing the probability equation:

\[
\begin{align*}
P(A) &= P(B \text{ and } C' \text{ and } D') = P(B \text{ and } C') - P(B \text{ and } C' \text{ and } D) \\
&= P(B) - P(B \text{ and } C) - (P(B \text{ and } D) - P(B \text{ and } C \text{ and } D)) \\
&= P(B) - P(C) - P(D) + P(C \text{ and } D) \quad (2)
\end{align*}
\]

So, we see that we will actually need a fourth query:

Fourth query Q4:
Select distinct col_1, col_2, col_3 ... 
From R1, R2, R3, R'1, R'2, R'3, R''1, R''2, R''3
Where condition1 AND condition2 AND condition3

We thus arrive at a new equation:

\[
P(B) - P(C) - P(D) + P(E) \text{ where } E = \{x \text{ is in Q4} \} \quad (3)
\]

To generalize to an arbitrary number, n, NOT EXISTS we see now that we need:

\[
1 + n \binom{n}{1} + n \binom{n}{2} + \ldots + n \binom{n}{n} = 2^n \text{ queries}
\]

where a query will have the form of taking the Q and merging with all possible subgroups from the set of nested NOT EXISTS queries. The generalized probability equation follows the inclusion-exclusion principle of probability with query merging equivalent to conjunctions.

Now that we have an equation that will compute the probability of a query, how do we actually sum the values efficiently? The number of consecutive positive and negative values in a row change as the number of nested queries increases. So, we make the observation that the probabilities that we add to the sum are specifically the ones that have merged together an odd number of queries; whereas for the ones we subtract from the total are exactly those that have merged together an even number of queries. Using this fact, we can merge the queries in a specific order where the signs of the probabilities will alternate.

The developed process, demonstrated in figure 3, generates queries for the probability calculation by first maintaining a list of already generated queries and then appending onto the list by merging new nested queries with those 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.
From ... 
Where ...) 

Generated Queries:
(+) q       (-)qq1        (-)qq2       (+)qq1q2 
Desired Order:
[q] -> [q, qq1] -> [q, qq1, qq1q2, qq2] 

Figure 3: Example of how list of generated queries is populated

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 the 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')
```

With this renaming, the combining of the outer query and inner query will not have the problem of two relations both called R1. But, as one can clearly see in the example, we cannot only rename the table aliases in the from clause, but the where clause must also be renamed to follow the scoping rules of SQL. To accomplish this, we implement a symbol table that will store the names that have been replaced and what they were replaced with. This symbol table acts like a stack where when we encounter an alias, we use the last stored renaming stored in the table as the new alias. This solves the problem of multiple relations with the same name.

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 all the inner queries and remove
them from the original query. At the end of this process, we have a list of all the nested queries and a
welcome outer query. Using the outer query as the starting point, generation of the necessary queries is
quite simple. For the first of the nested queries, we merge it with the outer query to create another
desired query. Next, we take another nested query, if it exists, and merge it with each of the already
generated queries in reverse order (just as the figure 3 demonstrated). We proceed in this fashion for
the remaining queries and we finish with a list of generated queries in the appropriate order. 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. The second is to use of the multi-simulation method. Clearly, explicitly
calculating the probability is much more desirable, so unless a user specifies that a simulation must be
run, MystiQ will check if all the generated queries, which are all monotone, can be computed exactly.
If so, we can compute the NOT EXISTS query by substituting in the values directly into the formulas
discussed in the theoretical section and get an exact solution. The second method of simulation is not
as simple to execute.

In a simulation, we no longer have exact probabilities to work with, but rather entire sets of
intervals for each of the generated queries; and one must understand how to interpret the NOT EXISTS
query in terms of these intervals. First, recall from the introduction to MystiQ, an interval represents
the range in which the true probability lies for a particular result of the query. Using this fact we create
a whole new set of intervals that will have the same meaning for a NOT EXISTS query that depends on
the intervals of the generated queries. The key to this dependency lies in the formulas developed in the
theoretical discussion above.

Computation of the endpoints of an interval for a NOT EXISTS query result 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 essentially be computing the smallest possible probability
value that could result from substituting values from the associated intervals (intervals representing
tuples with the same values) of the other queries. It is clear that the answer to the problem is for
probabilities that are added to the sum, the lower end point is chosen, whereas for those subtracted
from the sum, the upper end point is chosen. Computing the upper end point would simply be the
opposite of the previous method. The following provides a demonstration of the process

Let Q have interval \([a, b]\); Q joined with Q1 have interval \([a1, b1]\); Q joined with Q2 have
interval \([a2, b2]\); Q joined with Q1 and Q2 have interval \([a3, b3]\). (Using notation from
Figure 3)

\[
\begin{align*}
Q & \quad [a, b] \\
QQ1 & \quad [a1, b1] \\
QQ2 & \quad [a2, b2] \\
QQ1Q2 & \quad [a3, b3] \\
\text{NOT EXISTS:} & \quad [a, b] \implies [a-b1, b-a1] \implies [a-b1-b2, b-a1-a2] \implies [a-b1-b2+a3, b-a1-a2+b3]
\end{align*}
\]
With this key issue resolved the simulation process can proceed in much the same way as it did in the original implementation, but with a slight modification. The process begins with setting the NOT EXISTS result intervals to the outer query intervals since the outer query results are exactly the possible results for the original query. Then one of the NOT EXISTS intervals 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 other queries are simulated. These associated intervals are then aggregated together using the method just described before the come up with the new NOT EXISTS interval. This process is repeated until a certain number of intervals are isolated from each other giving us the top \( k \) tuples and thus completes the process.

**Testing and Debugging**

In order to gauge our implementation of NOT EXISTS, multiple tests were performed, mostly to examine its accuracy. To do this, queries had to be invented and the probabilistic results needed to be known. However, in the majority of cases, probabilities for interesting NOT EXISTS queries are not easily calculated by hand. So, accurate tests were extremely limited and needed to progress slowly, building on one another. Our tests first began with single NOT EXISTS queries to test if probabilities were being summed together properly, then using these single NOT EXISTS queries, some more complicated queries were derived from them to judge if multiple nested queries could be dealt with.

The first approach taken was to use deterministic tables in the NOT EXISTS query. In a deterministic table, if a tuple is in the table it has probability one, else zero. Using these special tables, we can generate queries whose results have probabilities equivalent to those that do not have the NOT EXISTS section of the query or zero. An example of such a query is:

```sql
SELECT DISTINCT pt.attr
FROM ProbabilisticTable pt, ProbabilisticTable2 pt2
WHERE pt.attr = pt2.attr
AND NOT EXISTS (SELECT *
FROM DeterministicTable dt
WHERE dt.attr = pt.attr)
```

This query finds all the attributes that are found in both the probabilistic tables but are not in the deterministic table. Due to the fact that the NOT EXISTS query used the deterministic table, the probabilities should be the same as just running the query:

```sql
SELECT DISTINCT pt.attr1
FROM ProbabilisticTable pt, ProbabilisticTable2 pt2
WHERE pt.attr = pt2.attr
```

with the probability for tuples that are also found in

```sql
SELECT DISTINCT pt.attr
FROM ProbabilisticTable pt, ProbabilisticTable2 pt2,
DeterministicTable dt
WHERE pt.attr = pt2.attr AND dt.attr = pt.attr
```

set to zero. With this in mind, all one needs to do is run the outer query alone to examine what the
probabilities ought to be and then run the second to find which tuples are to be 'eliminated' (probability goes to zero). After this knowledge is acquired, run the NOT EXISTS query to see if the results match up. Once this test passes, it is easy to extend this to multiple NOT EXISTS clauses. One just needs to ensure all tuples are eliminated correctly with the additional NOT EXISTS clauses. Testing with deterministic tables is good starting point, but more interesting queries often will only use probabilistic tables.

The next step in testing is to execute queries that only use probabilistic tables. In order to make sure values are correct, probabilities had to be calculated by hand. This meant that the query could not be overly complicated or else it would be very difficult to calculate the actual probability and there is a lot of room for human error. Hence, tests began with a single NOT EXISTS query that can easily be calculated. An example of such a query is:

```
SELECT DISTINCT *
FROM SightingsEvent s1, SightingsEvent s2
WHERE s1.tagid = s2.tagid and s1.timestmp < s2.timestmp
and NOT EXISTS
  (SELECT DISTINCT *
   FROM SightingsEvent s3
   WHERE s3.tagid = s2.tagid and s3.timestmp >
     s1.timestmp and s3.timestmp < s2.timestmp)
```

This query returns all pairs of sightings that occur consecutively. Using a small generated table we can calculate the probabilities exactly. An example is shown in Figure 5.

After testing such queries for a time, it is easy to expand to test multiple NOT EXISTS clauses by breaking the one NOT EXISTS clause into two separate ones and compare the probabilities.
Experiments

After running a sufficient number of correctness tests that all passed, some experiments were run to examine how well the implementation generalized to various inputs and how optimizations improved performance.

First, experiments were run to test how the implementation performed over different data sizes. To perform this task, tables representing ‘Products’ and ‘Orders’ were used. These tables were used because they provided a straightforward way to populate the tables and there are queries in which optimizations are guaranteed to be found. The optimized results were then used for comparisons. Figures 6, 7, and 8 graph the running times for a single NOT EXISTS query over varying data set sizes. The query used was simply to select all products such that there does not exist any order of that product with price greater than ten.
From these graphs, 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 graphs, that with optimizations, we get an entire factor of improvement. 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 following graph charts the change in times as we varied the number of simulations from 10000 to 70000 on a fixed data set of about 10000 (we used the same query and tables as before).
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 Figure 9 showed 30000 to be the optimal value for the number of Simulations per Step, this is only for this particular case. When there are more NOT EXISTS queries, the number of steps per simulation generally will increase due to the more complex nature of the query. However, even in these 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 data closely related to the RFID data (times when a person entered and left a room). A complex query was written that searched for concurrent events of entering and leaving a room for a particular person. This query, at first, used two NOT EXISTS queries and all results were fairly close to zero due to the way the table was populated. An example of a query is found in Figure 10. Then the query was translated into a single NOT EXISTS query to see how timings are affected. This new query is found in Figure 11 (where the AllSightings table is simply the union of EnteredRoom/LeftRoom events).

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. Clearly there is a time increase caused by the additional queries generated and executed by the additional level of complexity (about 25 times slower). However, there is an equally significant increase in the time to run the simulations (also about 25 times slower). This is because we no longer only simulate two sets of intervals, but rather four sets of them.
In addition to the immediate results of the experiment, two other observations were made. The first issue arises when there are many probabilities that lie very close together, particularly when they are all close to zero. While executions were run, there was a clear slow down when all the intervals narrowed and were centered about zero. Figure 12 provides an example of the intervals. 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.

**Figure 10: Query using two NOT EXISTS**

```
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)
```

**Figure 11: Query using only one NOT EXISTS**

```
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 second observation is that the timings seem to reflect the complexity of the original query itself. The queries made for RFID data use a more complex query in the NOT EXISTS clause than was used in the Products/Orders example. With only a data set size of 660 tuples, the time to compute the answer was almost twice as slow as the time to execute the Products/Orders query on a data set size of 3946 tuples.

Conclusion

In conclusion, through testing and experiments, the theory and implementation of evaluating NOT EXISTS queries appear to be accurate. Furthermore, the optimizations made do show improvement of performance by about 94%. However, the execution time still is fairly slow and can be improved upon. As the graphs above depicted, 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.
Related Work