Managing Probabilistic Data with MystiQ: The Can-Do, the Could-Do, and the Can't-Do*

Christopher Re and Dan Suciu

University of Washington

1 Introduction

MystiQ is a system that allows users to define a probabilistic database, then to evaluate SQL queries over this database. MystiQ is a middleware: the data itself is stored in a standard relational database system, and MystiQ is providing the probabilistic semantics. The advantage of a middleware over a re-implementation from scratch is that it can leverage the infrastructure of an existing database engine, e.g. indexes, query evaluation, query optimization, etc. Furthermore, MystiQ attempts to perform most of the probabilistic inference inside the relational database engine. MystiQ is currently available from `mystiq.cs.washington.edu`.

The MystiQ system resulted from research on probabilistic databases at the University of Washington [8, 11, 10, 13, 23, 14]. Some of these research results have been fully incorporated in MystiQ, like the query evaluation techniques that allow it to evaluate SELECT-FROM-WHERE-GROUPBY queries over large probabilistic databases: this is what MystiQ *can do*. Other results are not implemented in the system, but they could either be implemented in some future version after only minor extensions, or can be used even today by a database administrator to perform certain data management tasks manually; an example are the techniques for representing materialized views over probabilistic data. This is what MystiQ *could do*. Finally, other research results require more work before they can be implemented in a system. For example, our evaluation techniques for queries with a HAVING clause applies only to safe queries; for another example, we currently don't know of a good approach to extend safe queries and safe plans to queries with self-joins. This is what MystiQ *can't do*.

In this paper we give a gentle introduction into the MystiQ system, and describe the associated research that is used, or could be used, or is not yet ready to be used in MystiQ.

Related Work Our research has focused primarily on SQL query evaluation and on views. Other groups have studied different aspects of probabilistic or incomplete databases. The Trio project [29, 7, 6, 15] focused on the study of lineage in incomplete databases. The MayBMS project has focused on representation problems, query language design, and query evaluation [3, 4, 2, 20]. Other

* Supported in part by NSF grants IIS-0415193, IIS-0627585, IIS-0513877, IIS-0428168, and a gift from Microsoft.

groups have studied correlations in probabilistic databases [27], continuous random variables [9], and complex probabilistic models with native Monte Carlo simulations [16]. Earlier work include [5] and the ProbView system [21].

2 The Data and Query Model

2.1 Tables and Events

To use MystiQ, one must start from a relational database, created on a relational database management system; in our work we used mostly postgres or SQL Server, but any RDBMS with a JDBC conection can be used instead. Upon starting MystiQ the user needs to provide the information to connect to the database, and the name of a *configuration file* that specifies how to interpret the tables in the database as probabilistic events.

The configuration file consists of three kinds of statements:

```
STATEMENT ::= TABLE table-name(attributes)
            | CREATE EVENT event-name(attributes1)
                    [CHOICE (attributes2)]
                    ON table-name(prob-expression)
            | CREATE VIEW event-name AS
                    SELECT ...
                    FROM ...
                    WHERE ...
```

The TABLE statements describe the schema of the relational database: while this is redundant, since MystiQ could obtain it directly from the database, in the current implementation MystiQ requires the schema to be given in the configuration file. Only the attribute names need to be listed, not their types.

A CREATE EVENT statement defines a probabilistic table, called *event*, from a table in the database. Here `table-name` is a relation in the database, while `event-name` is the probabilistic relation defined by this statement. The schema of the probabilistic table is the same as that of the deterministic table from which it is derived, and its semantics is a set of possible worlds: each subset of `table-name` is a possible world for `event-name`. In each possible world, `attributes1` are a key: they are called the *event key*. When present, `attributes2` are called the *choice* attributes. MystiQ requires that `attributes1`$\cup$`attributes2` contain the primary key of `table-name`. Finally, `prob-expression` is an expression, possibly involving attributes of `table-name`, defining the marginal probability of a tuple in `event-name`.

By definition, tuples with distinct values of the event keys are independent probabilistic events, and tuples with the same event keys but distinct values choice attributes are disjoint (or exclusive) probabilistic events. Under this assumption the marginal tuple probabilities uniquely define the probability of each possible world; we refer the reader to [14] for additional details.

In the CREATE EVENT statement `table-name` can also be a probabilistic table; in that case no choice attributes are allowed. We will illustrate this in Section 4.

The CREATE VIEW statement is identical to that in SQL, except that the FROM clause may use both events and deterministic relations. We describe views in Sec. 4.

For a simple illustration, suppose we have a database or products, where the colors are uncertain:

`Product:`

Name	Color	Prob
Gizmo	Green	0.3
Gizmo	Blue	0.5
Gizmo	Red	0.2
Gadget	Blue	0.1
Gadget	Black	0.9

This is a standard deterministic relation, there is no probabilistic semantics yet. The attribute `Prob` represents our confidence in that product's color. In order to turn this table into a probabilistic relation, we define the configuration file:

```
TABLE Product(Name, Color, Prob)
CREATE EVENT ProductUniqueColor(Name) CHOICE(Color) ON Product(Prob)
```

In the new relation `ProductUniqueColor` is probabilistic, and `Name` is its key. Its semantics are all the possible subsets of `Product`, and each such subset has a certain probability. For example the instance[1]:

`ProductUniqueColor:`

Name	Color
Gizmo	Blue
Gadget	Blue

has probability $0.5*0.1 = 0.05$. There are 6 possible worlds, because `Name` must be a key.

Alternatively, we can interpret the uncertainty about the `Color` in `Product` differently, namely as saying that any product may have multiple colors, but it is uncertain which colors are present. Then we define another event:

```
CREATE EVENT ProductMultipleColors(Name, Color) ON Product(Prob)
```

[1] Strictly speaking `Prob` should also be an attribute of `ProductUniqueColor`, but we omit it since we only need it to define the probabilities of the possible worlds.

Now every subset of `Product` is a possible world, and there are 32 possible worlds. The world above (with two tuples) is still a possible world, but now it has probability $(1-0.3)*0.5*(1-0.2)*0.1*(1-0.9) = 0.0028$.

As a last example, we can make each of the 32 possible worlds equally likely (with probability 1.0/32) by defining the event:

```
CREATE EVENT ProductUniform(Name, Color) ON Product(1.0/32)
```

In MystiQ we can define several probabilistic tables from the same deterministic table, and the system treats them as independent probabilistic tables.

2.2 Queries

MystiQ currently supports SELECT-FROM-WHERE-GROUPBY with aggregates in the SELECT clause. The FROM clause may mention both deterministic relations, and probabilistic relations. Nested queries are permitted, but these may only refer to deterministic relations (i.e. no event names). MystiQ returns a list of tuple answers together with their marginal probabilities and computes expected values for the aggregate functions (if any are present). The answers are always ranked by their marginal probabilities, and only the top k answers are returned to the user, where k is either specified in the query or is a system parameter.

We illustrate queries with three examples. The following returns all products that are not red:

```
Q1 = SELECT DISTINCT Name FROM ProductUniqueColor WHERE Color!='red'
```

The answer is:

Name	Probability
Gadget	1.0
Gizmo	0.8

Thus, the user learns that Gadget is certainly not red, since its probability is 1.0. Gizmo is not red with probability 0.8. The system returns only the marginal tuple probabilities in the answer, here 1.0 and 0.8, and does not retain any correlations between the tuples[2]: in order to retain the tuple correlations (which are needed, for example, in order to run another SQL query on the answer) one has to define a view, see Sec. 4.

The tuples are ranked in decreasing order of their probability. It is very important to rank answers when processing uncertain data, because the uncertainties will often cause many false positives in the answer. We want to return to the user the tuples with a higher probability first, and spare him the effort to examine tuples with small probabilities. MystiQ ranks the answers, and computes only the top k.

Note the importance of including DISTINCT in this query. If we run the same query without DISTINCT, then the answer is:

[2] The tuples in the answer of `Q1` are independent; the tuples in the answers of `Q2` and `Q3` are correlated in non-trivial ways.

Name	Probability
Gadget	0.9
Gizmo	0.5
Gizmo	0.3
Gadget	0.1

It is much harder for the user to see here that Gadget is guaranteed to be a correct answer, or to get a sense of which are the most likely products to not be red.

For the second query, we assume to have another probabilistic table, called `LikesMultipleColor`, which lists for every customer, color pair the probability that the customer likes that color:

```
TABLE Customer(custID, Color, Prob)
CREATE EVENT LikesMultipleColors(custID, Color) on Customer(Prob)
```

In a possible world, a customer may like multiple colors. The query below finds all customers to which we could sell products whose colors they like:

```
Q2 = SELECT DISTINCT y.custID
     FROM  ProductUniqueColor x, LikesMultipleColors y
     WHERE x.Color = y.Color
```

Our third query illustrates aggregates, which are interpreted as expected values. The query computes for each color the number of products that have that color:

```
Q3 = SELECT Color, count(*) FROM ProductUniqueColor GROUP BY Color
```

and the answer is:

Color	Count	Probability
Black	0.9	0.9
Blue	0.6	0.55
Green	0.3	0.3
Red	0.2	0.2

3 Query Evaluation

3.1 Can Do: Safe and Unsafe Queries

MystiQ uses two algorithms for query evaluation: an efficient algorithm that works only for *safe* queries, and a more expensive algorithm that works for all queries.

First, if the query is *safe*, then MystiQ computes a safe plan and rewrites the plan into a new SQL query called the *extensive query*. It then evaluates

the extensive query on the database engine. The answers of the extensive query already contain the correct probabilities, and are sorted in decreasing order of these probabilities: MystiQ simply cuts this set after the top k answers. For example, the query `Q1` is safe, and the extensive query simply sums for each product the probabilities of all entries that are not red. The treatment of safe queries is fully described in [10] (this contains a system's perspective) and in [14] (this contains a complete, theoretical treatment).

Second, if the query is *unsafe* then MystiQ generates a much simpler SQL query that simply fetches and joins all relevant tuples from the database. Once these tuples are read, they are grouped by their output attributes, and MystiQ computes the probability of each output tuple using Luby and Karp's Monte Carlo simulation algorithm [23]. For example, the query `Q2` is unsafe (in fact, it's data complexity is #P hard because it is essentially h_2 in [14]).

Safe queries run very fast, mostly at the same speed as a normal SQL query on a relational database. Unsafe queries, by contrast, are about two orders of magnitude slower. Thus, it is very important for MystiQ to identify if a query is safe, and run the much faster algorithm. In a trivial case, if all relations in the FROM clause are deterministic, then the query is automatically safe: this query is simply passed through to the database system. Another trivial case is when the query does not have DISTINCT or GROUP BY, then it is also safe; however, as we saw above, queries over uncertain data are likely to have the DISTINCT clause.

3.2 Can Do: Optimize

MystiQ uses two optimizations for unsafe queries. First, it tries to identify subqueries that are both safe, and whose answers consists of independent or disjoint tuples; they are called *super-safe* subqueries. These subqueries can both be evaluated with a safe plan, and their results can be used as any other probabilistic table.

The second optimization is much more important and consists of an aggressive exploitation of the top-k query answering paradigm. MystiQ attempts to perform the Monte Carlo simulation only for the top k tuples, which are the only tuples that need to be returned to the user: there are many more tuples in the answer, and restricting the simulation to only the top k results in significant performance improvement. But this is a chicken and egg problem: MystiQ doesn't know which of the candidate answer tuples are the top k until it has run the Monte Carlo simulation on all answers and ranked them by their probability. The solution to this problem is described in [23] and is called *multisimulation*.

3.3 Could Do: More Aggregates, NOT Exists

MystiQ supports only `sum` and `count`. It also supports `avg`, but it interprets it as the ratio of `sum` and `count`. This is not the correct semantics of `avg`: a correct treatment of `avg` is more difficult, see [17]. MystiQ *could* support `min` and `max`:

they require only minor extensions to the current MystiQ system, but they are not implemented at the time of writing.

Some applications, such as management of RFID data, require SQL queries with NOT EXISTS predicates. We have investigated evaluation algorithms for queries with one level of NOT EXISTS predicates in [28], and described an evaluation algorithm that could be fully integrated in MystiQ. Our algorithm is exponential in the number of NOT EXISTS predicates; some optimizations are likely required in order to make this approach more practical.

3.4 Can't Do: Queries with a HAVING Clause, and queries with Selfjoins

Queries with a HAVING clause are very important in decision support. For example, a manager wants to retrieve all products having at least 50 customers interested in that product. In the setting of a probabilistic database this means that we need to compute, for a given product, the probability that at least 50 customers are interested in that product. This is different from, and harder than computing the expected number of customers interested in a given product. Motivated by the need to support HAVING queries, we have studied this problem in [24]. We have described an efficient algorithm to evaluate queries with a HAVING clause, but only if its skeleton (i.e. the query obtained by removing the HAVING clause) is safe. If the query is unsafe, then we currently do not know how to evaluate a query with a HAVING clause, in similar fashion to the Monte Carlo simulation approach[3].

We say that a query has a self join if the same event name occurs twice in the FROM clause. MystiQ treats such queries automatically as unsafe queries. However, not all queries with self-joins are hard, and in fact some quite simple and natural queries with self-joins are in PTIME. Motivated by the need to extend MystiQ to handle efficiently queries with self joins, we have conducted a theoretical study in [12], and we have described a PTIME algorithm for a large class of queries with self-joins. Unfortunately, this algorithm has two limitation. First, it was developed only for probabilistic tables with independent tuples [4]: there are currently no efficient algorithms known for evaluating queries with self-joins over disjoint-independent databases. Second, the PTIME algorithm seems to be quite different from the safe plans used by MystiQ: more research is needed in order to adapt those PTIME algorithms to a query processor.

4 Views

In addition to views created on the relational database (which MystiQ treats as any regular deterministic table), MystiQ allows users to define views over events.

[3] A naive Monte Carlo algorithm can be used, but it will run in exponential time to achieve a given precision, unlike Luby and Karp's algorithms that runs in polynomial time to achieve a fixed precision.

[4] These are CREATE EVENT statements without the CHOICE clause.

These are an important tool in managing probabilistic data, and we discuss them here to some extend.

4.1 Can Do: Virtual Views

MystiQ allows users to define views over events; the result is a new event. For example, the view below defines an event called `Recommendations`, which contains pairs customer, product where the customer likes the product's color.

```
CREATE VIEW Recommendations
     SELECT DISTINCT x.custID, y.Name
     FROM LikesMultipleColors x, ProductUniqueColor y
     WHERE x.Color = y.Color
```

Views in MystiQ have a compositional semantics when used in another SQL query. In particular this means that the correlations between the tuples in the view are accounted for. For that, MystiQ supports only virtual views, and expands the view definition when it is used in another SQL query. For example if a SQL query uses `Recommendation` in the FROM clause, then MystiQ simply expands the view definition in the query, and this is by definition compositional semantics.

Note that materializing the view makes it much harder to ensure compositional semantics. It is not sufficient to materialize `Recommendations` in a table that stores the marginal probability of each tuple, because we need to also represent somehow the correlations between tuples. By contrast, when we run a query in MystiQ we only retrieve the marginal probabilities. Thus, in MystiQ views and queries are different: views are used for their compositional semantics, while queries are used to retrieve the marginal probabilities.

Views are important in probabilistic databases for two reasons.

First, by adding views to disjoint-independent probabilistic databases one obtains a complete representation system. In notation:

$$\textit{Disjoint-independent-Probabilistic-DBs} + \textit{Views} = \textit{Complete-Representation}$$

We proved this in [22, 14] using a simple, constructive proof that is quite useful in practice. Our result is similar to other results in the literature. For example Benjelloun et al. [7] show that a lineage system consisting essentially of DNF formulas also forms a complete representation system, and a similar result was shown by Antova et al. [3]. Our particular formulation of the result emphasizes the role of views in achieving completeness. This, we feel, is important for practical applications, since views are already used widely in data management.

We illustrate now the completeness result with a simple example. Suppose that we have a probabilistic table `ProductColorComplex` with exactly three possible worlds:

Name	Color
Gizmo	Green
Gizmo	Blue
Gadget	Black

Name	Color
Gizmo	Blue
Gadget	Black

Name	Color
Gizmo	Red
Gadget	Blue

and their probabilities are 0.2, 0.3, 0.5. Note that the tuples are no longer disjoint or independent, hence these possible worlds cannot be specified by a CREATE EVENT statement. Instead, in MystiQ the user would create the following two deterministic tables:

`ProductWorld`:

Name	Color	WID
Gizmo	Green	W1
Gizmo	Blue	W1
Gadget	Black	W1
Gizmo	Blue	W2
Gadget	Black	W2
Name	Color	W3
Gizmo	Red	W3
Gadget	Blue	W3

`World`:

WID	Prob
W1	0.2
W2	0.3
W3	0.5

then define the following events:

```
CREATE EVENT PossWorld( ) CHOICE(WID) on World(Prob)
CREATE View ProductColorComplex AS
    SELECT DISTINCT Name, Color
    FROM ProductWorld x, PossWorld y
    WHERE x.WID = y.WID
```

The possible worlds of `ProductColorComplex` are exactly the three worlds above (because the possible worlds of `PossWorld` are {`W1`}, {`W2`}, and {`W3`}), and their probabilities are also identical.

The second reason why views are important is that they can be used to express rules with confidences. For example, consider the following rule, derived from the Calo project [1]:

```
IsAbout(e, p) :- EmailFrom(e, u), WorksOn(u, p)  CONFIDENCE = 0.8
```

The purpose of the rule is to predict if an email is about a project. The rule says that if the email is from a user `u`, and `u` works on a project, then that email is about that project with confidence 0.8. This can be expressed in MystiQ as:

```
CREATE VIEW V AS
    SELECT DISTINCT x.Email, y.Project
    FROM EmailFrom x, WorksOn y
    WHERE x.Sender = y.Person

CREATE EVENT IsAbout(Email, Project) ON V(0.8)
```

If `EmailFrom` and `WorksOn` are deterministic tables, then `V` is also deterministic and `IsAbout` is a tuple-independent relation where each tuple has probability 0.8. If `EmailFrom` or `WorksOn` are probabilistic tables, then `V` is also a probabilistic table. The new event `IsAbout` decreases the marginal probability of each tuple by 0.8, while maintaining any correlations between the tuples in `V`.

4.2 Could Do: Representable Materialized Views

Views are as important for managing uncertain data as they are for managing traditional data. Virtual views are easy to implement, but result in poor query performance. After view expansion the new query is larger than the original query; if a query was safe, it may no longer be safe after view expansion. To improve the query performance on probabilistic views we need to materialize them. By a *materialized probabilistic view* we mean a table in a relational database that stores for each tuple its marginal probability, and represents in some way the correlations between the tuples in that relation. Lineage [7] has been developed precisely with this goal: to represent how tuples were derived and, thus, to capture all their possible correlations. However, materializing views with lineage only postpones and exacerbates the query evaluation problem. We need a technique that allows us to materialize the view and benefit from having performed the computations offline.

Motivated by this need we have studied the problem of materializing probabilistic views in [25, 26], using two approaches. The first approach was to derive automatically a full, or a partial representation of the view.

A *partial representation* of a view consists of two sets of attributes called *key attributes*, and the *choice attributes*: the set of attributes that are neither key nor choice are called *unknown attributes*. The partial representation is correct if any two tuples that have distinct values of the key attributes are independent, and any two tuples that have the same values for the key and unknown attributes and have distinct values for the choice attributes are disjoint (i.e. exclusive). In addition, when all attributes are known (i.e. are either key or choice attributes) then we say that the representation is *full*. Essentially, a fully representable view is like a probabilistic database defined by a CREATE EVENT statement, and therefore can be used during query processing without any further extensions.

We have shown that any view defined by a SQL query has a canonical partial representation, which is maximal in the sense that the sets of key and choice attributes are as large as possible. Note that any view admits a trivial representation, where all attributes are unknown, but this is useless in practice because it makes no claims about tuples being disjoint or independent. The canonical representation is the best one can get for this view definition. If a full representation exists then it is also canonical, since we cannot further increase the set of key or choice attributes.

MystiQ currently has no support for materialized views. If the view has a full representation, then the database administrator can materialize it on the database server (using MystiQ to compute the probabilities) and then define in

the configuration file as a CREATE EVENT; MystiQ can then use it as a regular probabilistic table.

If the view has only a partial representation, then MystiQ could still use the materialized view to answer some queries, while for others it would have to fall back on view expansion: however this part is not implemented at the time of writing. For example the query may join two tuples in the view that agree on the key attributes, and disagree on the unknown attributes: we cannot compute the probability of the joined tuple because we don't know the correlations between these two tuples. To support such views, Mystiq would have to keep extra correlation information.

4.3 Could Do: Sufficient Lineage

Any probabilistic view can be stored faithfully by storing the complete *lineage* for each tuple in the output of the view, that is every derivation for that tuple in the database. This approach reduces the cost of processing the joins in the view, but does not reduce the complexity of the probabilistic portion of query processing, which is often the dominant cost. However, many applications can tolerate approximate query results. For example, in information extraction applications the probabilities are usually set *heuristically* and so there is very little real difference between a a tuple with probability of 0.99 and one that returns 0.995. Intuitively, to compute probabilities approximately, it is unnecessary to keep all possible derivations, just the most important ones. Figuring out which tuples are most important for a tuple, is the technical development underlying *approximate lineage* [26].

One form of approximate lineage is *sufficient lineage*, where instead of storing the complete lineage, we store only a subset of the lineage. Of course, this introduces some error in the database and so our goal is to construct such lineage to guarantee small error. An interesting property of sufficient lineage is that any conjunctive query (SFW query) returns a probability value that is a lower bound of the true probability (without approximation). Further, sufficient lineage takes up space that is orders of magnitude smaller than a complete approach, even for very low error tolerance. The reduced size enables query processing and other data exploration tasks to proceed much more efficiently. Most importantly for Mystiq, sufficient lineage is syntactically identical to standard lineage and so can be used directly.

4.4 Can't Do: Polynomial Lineage

Although sufficient lineage may provide large amounts of compression, it is often possible to get better compression ratios using *polynomial lineage*. The essential idea of polynomial lineage is to transform the lineage functions into a polynomial and then use techniques from analysis, such as Fourier transforms and Taylor series, to approximate the resulting polynomial. While we show how to use this representation to process queries, the representation of these functions is quite different than the lineage functions used in Mystiq.

5 Constraints

Database constraints are a promising technique for managing uncertain data. The hope is that by specifying application specific constraints the database administrator can reduce the set of worlds to only those that "make sense" in that particular application. A common example of constraints are key constraints: in fact the CREATE EVENT statement already incorporates one key constraint for every probabilistic table. However, multiple, overlapping constraints require significant extensions.

5.1 Could Do: Hard Constraints

A hard constraint is an assertion must hold in all possible worlds: in other words, the worlds that fail the constraint are removed from the set of possible worlds. Hard constraints are dealt with by conditioning: that is, if B is the statement that the constraint holds (on a possible world), the probability of a boolean query Q in the presence of the constraint is $P(Q|B) = P(QB)/P(B)$. This observation was used by Koch and Olteanu in [20] to incorporate hard constraints in MayBMS. The challenge in this approach is computing the constraint probabilities efficiently, i.e. $P(B)$ and $P(QB)$. In the case of functional dependencies, B is the negation of a conjunctive query; in more complicated settings it may be a more general formula. MystiQ currently does not support hard constraints.

5.2 Can't Do: Soft Constraints

A soft constraint is an assertion is increases the probability of the worlds where it holds, and decreases the probability of worlds where it doesn't hold. This semantics is especially appropriate is the constraint is learned automatically from training data. For example the machine learning module may identify a set of attributes that are a *soft key*: some violations exists, but otherwise the attributes form a key.

Motivated by the need to incorporate in MystiQ constraints learned automatically from the data, we have studied *soft key constraints* in [18]. The natural semantics for soft keys is given by a Markov Network of a special kind, where the potential of a world depends on the number of violations to the key constraint. We were especially interested in the case when multiple soft keys exists for the same table, and have defined a class of *safe* queries for a given set of soft key constraints. However, we currently don't know how to evaluate unsafe queries, nor how to extend the query evaluation algorithms to other kinds of constraints. MystiQ does not currently support soft constraints of any kind.

6 Discussions

The goal of MystiQ is to serve as proof of concept: we proved that it is possible to evaluate SQL queries on large probabilistic databases, with high performance (for safe queries), or with tolerable performance (for unsafe queries).

For safe queries, the main performance bottleneck is the relational database engine, which needs to support complex, nested SQL queries: we found here that commercial database systems (like SQL Server) perform much better than free systems (like postgres). For unsafe queries, the main performance bottleneck is the inner loop of MystiQ's Monte Carlo simulation. Here we learned two lessons. The first is a project management lesson. Our initial prototype (done by the first author), which was used to report the experiments in [22], was written in C++ and deployed a number of C++ programming hacks to boost performance of the most critical Monte Carlo simulation steps: as a result the system had a very high performance. Subsequently, we used the funds from a small grant to hire a programmer and re-implement MystiQ in Java (for portability). We instructed him to emphasize features, the user interface, and completeness over performance. The initial performance of the Java code for the multisimulation was so poor that we had to replace the inner loop with our initial C++ code. This increased the performance to reasonable level, without matching that of our earlier prototype. The second lesson we learned is that the Monte Carlo simulation problem is far from being well understood. MystiQ has a friendly graphical interface that allows us to view the progress of the multisimulation and see the progress of the confidence bounds for all candidate tuples. By examining this progress it becomes quickly obvious that the top k tuples and their rankings converge much faster than the bounds given by Luby and Karp's formula [19]. While the formula is theoretically tight, observing the progress of MystiQ's multisimulation algorithm suggest that further improvements or optimizations are possible, either by stopping the current simulation earlier, or by using some other simulation algorithm.

References

1. J.L. Ambite, V.K. Chaudhri, R. Fikes, J. Jenkins, S. Mishra, M. Muslea, T.E. Uribe, and G. Yang. Design and implementation of the CALO query manager. In *AAAI*, 2006.
2. L. Antova, C. Koch, and D. Olteanu. 10^(10^6) worlds and beyond: Efficient representation and processing of incomplete information. In *ICDE*, 2007.
3. L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing incomplete information with probabilistic world-set decompositions (demonstration). In *ICDE*, 2007.
4. L. Antova, C. Koch, and D. Olteanu. World-set decompositions: Expressiveness and efficient algorithms. In *ICDT*, pages 194–208, 2007.
5. D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Trans. Knowl. Data Eng.*, 4(5):487–502, 1992.
6. O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.
7. O. Benjelloun, A. Das Sarma, C. Hayworth, and J. Widom. An introduction to ULDBs and the Trio system. *IEEE Data Eng. Bull*, 29(1):5–16, 2006.
8. J. Boulos, N .Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. Mystiq: A system for finding more answers by using probabilities. In *SIGMOD*, 2005. system demo.

9. R. Cheng and S. Prabhakar. Managing uncertainty in sensor databases. *SIGMOD Record*, 32(4):41–46, December 2003.
10. N. Dalvi, Chris Re, and D. Suciu. Query evaluation on probabilistic databases. *IEEE Data Engineering Bulletin*, 29(1):25–31, 2006.
11. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, Toronto, Canada, 2004.
12. N. Dalvi and D. Suciu. The dichotomy of conjunctive queries on probabilistic structures. In *PODS*, pages 293–302, 2007.
13. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDBJ*, 16(4):523–544, 2007.
14. N. Dalvi and D. Suciu. Management of probabilistic data: Foundations and challenges. In *PODS*, pages 1–12, Beijing, China, 2007. (invited talk).
15. A. Das Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.
16. R. Jampani, F. Xu, M. Wu, L.L. Perez, C.M. Jermaine, and P.J. Haas. MCDB: a Monte Carlo approach to managing uncertain data. In *SIGMOD*, pages 687–700, 2008.
17. T.S. Jayram, S. Kale, and E. Vee. Efficient aggregation algorithms for probabilistic data. In *SODA*, 2007.
18. A. Jha, V. Rastogi, and D. Suciu. Evaluating queries in the presence of soft key constraints. In *PODS*, 2008.
19. R. Karp and M. Luby. Monte-Carlo algorithms for enumeration and reliability problems. In *Proceedings of the annual ACM symposium on Theory of computing*, 1983.
20. C. Koch and D. Olteanu. Conditioning probabilistic databases. In *VLDB*, 2008.
21. L. Lakshmanan, N. Leone, R. Ross, and V.S. Subrahmanian. Probview: A flexible probabilistic database system. *ACM Trans. Database Syst.*, 22(3), 1997.
22. C. Re, N. Dalvi, and D. Suciu. Efficient Top-k query evaluation on probabilistic data (extended version). Technical Report 2006-06-05, University of Washington, 2006.
23. C. Re, N. Dalvi, and D. Suciu. Efficient Top-k query evaluation on probabilistic data. In *ICDE*, 2007.
24. C. Re and D.Suciu. Efficient evaluation of having queries on a probabilistic database. In *Proceedings of DBPL*, 2007.
25. C. Re and D.Suciu. Materialized views in probabilistic databases for information exchange and query optimization. In *Proceedings of VLDB*, 2007.
26. C. Re and D. Suciu. Approximate lineage for probabilistic databases. In *VLDB*, 2008.
27. Prithviraj Sen and Amol Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.
28. T.Y. Wang, C. Re, and D.Suciu. Implementing not exists predicates over a probabilistic database. In *Proceedings of MUD*, 2008.
29. Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.