On APIs for Probabilistic Databases

Lyublena Antova and Christoph Koch
{lantova,koch@cs.cornell.edu}

Cornell University

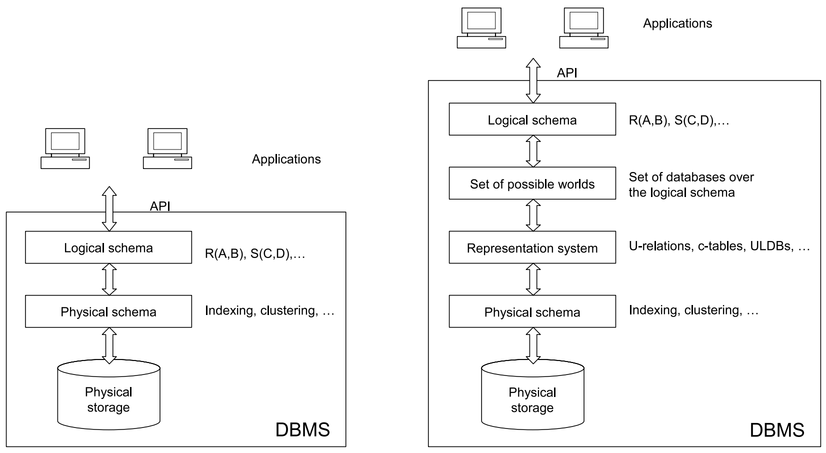
Abstract. We study database application programming interfaces for uncertain and probabilistic databases and present a programming model that is independent of representation details. Conceptually, we use the possible worlds semantics, and programs are independently evaluated in each world. We study a class of programs that appear to the user as if they are running in a single world rather than on a set of possible worlds. We present an algorithm for efficiently verifying this property. We discuss how updates can be implemented in uncertain database management systems, and propose techniques for optimizing database programs.

1 Introduction

In the last years there has been a profusion of research on managing uncertain and probabilistic data in different application scenarios such as Web information extraction, data cleaning, and tracking moving objects [1, 3, 4, 8–10]. Research on managing uncertainty has been focused on space-efficient models for representing uncertain information, query languages and efficient query processing, as well as confidence computation and ranking. Several systems for managing uncertain data are being developed (see e.g. [4, 3, 1]). However, no mature systems have evolved that support application development for uncertain databases.

A widely used approach to managing uncertain data is the possible worlds model. For example, in a moving object tracking scenario there can be several possible guesses for the identity of an object where the exact one is not known for sure. Each such option corresponds to one possible world. Under the possible worlds semantics, a query on an uncertain database is conceptually evaluated on each world independently, and the result is added to that world. In the special case when the possible world-set consists of a single world, this semantics coincides with the standard query evaluation semantics. In many scenarios however the set of possible worlds can be very large, or even infinite. Systems for managing uncertain data usually employ a compact representation for storing the possible worlds: e.g. MystiQ uses the tuple-independence model, Trio's representation is called ULDBs (databases with uncertainty and lineage), and MayBMS uses U-relations. All these systems rewrite queries on the set of possible worlds into queries on the representation.

An important component missing from current systems for managing uncertain data is the ability to write database application programs that access and update data through an application programming interface (API). Current



(a) Traditional DBMS (b) DBMS for uncertain data

Fig. 1. Levels of abstraction provided by a Database Management System.

systems either allow only batch execution of SQL-like queries on the uncertain database with no user interaction, or have a representation-dependent programming model that requires knowledge of system-specific implementation details.

Traditional database management systems abstract away the physical details of how the data is stored. External applications interact with the database by formulating queries and updates on the *logical schema*, which are then translated into operations on the physical storage structures on disk. Figure 1 shows an architecture of a traditional DBMS, and one of a DBMS for uncertain data. Compared to traditional DBMSs, systems for managing uncertain data need to deal with two additional levels of abstraction, the representation system and the possible worlds model.

To demonstrate some of the challenges of designing an API for uncertain databases, consider the following example (essentially from [3]). Figure 2 shows an example of a police database containing reports on stolen cars. Due to conflicting or missing information, several instances are possible, as shown in the figure. An application that manages such data should provide users with the ability to update or insert new evidence regarding observed objects, execute queries, or apply expert knowledge to resolve inconsistency.

The following program allows to enter new evidence about a stolen car. In case the car already exists in the database, and the information about it matches the user's input, the program increments the number of witnesses; otherwise a new entry is inserted into the database.

```
read("Enter license plate:", $x);
if (exists select * from cars where num=$x){ // modify existing entry
    for($t in select * from cars where num=$x){
      write("Current entry: $t");
      read("New location and color:", $loc,$color);
      if (exists select * from cars where num=$x and loc=$loc and color=$color)
```

Cars^1	num	color	loc	wit
1	S87	red	MN	1
2	M34	blue	PA	1

Cars^2	num	color	loc	wit
1	S87	red	TX	1
2	M34	blue	MD	1

Cars^3	num	color	loc	wit
1	B87	red	TX	1
2	M34	blue	PA	1

Cars^4	num	color	loc	wit
1	B87	red	TX	1
2	M34	blue	MD	1

(a)

```
Current entry: S87 red MN
New location and color: _
```

```
Current entry: S87 red TX
New location and color: _
```

```
No entry found for S87
Enter location and color: _
```

```
No entry found for S87
Enter location and color: _
```

(b)

Cars^1	num	color	loc	wit
1	S87	red	MN	2
2	M34	blue	PA	1

Cars^2	num	color	loc	wit
1	S87	red	TX	1
2	M34	blue	MD	1
3	S87	red	MN	1

Cars^3	num	color	loc	wit
1	B87	red	TX	1
2	M34	blue	PA	1
3	S87	red	MN	1

Cars^4	num	color	loc	wit
1	B87	red	TX	1
2	M34	blue	MD	1
3	S87	red	MN	1

(c)

Fig. 2. Executing programs on uncertain databases: (a) set of possible worlds reporting information on stolen cars; (b) output of the program in each of the worlds of (a); (c) result of running the program on the world-set of (a).

```
        update cars set wit=wit+1 where num=$x and loc=$loc and color=$color;
      else
        insert into cars values($x,$loc,$color,1);
    }
}
else { // entry does not exist
    write("No entry found for $x");
    read("Enter location and color:", $loc, $color);
    insert into cars values($x,$loc,$color,1);
}
```

The program makes sense in the case of certain databases and is straightforward to execute. What is different in the presence of uncertainty? Consider the world-set in Figure 2 (a). The four worlds differ in the license plate number specified for car 1, and the reported location for cars 1 and 2. If the user enters S87 for the license plate number, in worlds 1 and 2 the car already exists in the database, so its entry will be shown to the user. In worlds 3 and 4 the car is not found and the user will receive a message reporting that. As shown in Figure 2 (b), there are three different outputs for the four worlds of Figure 2 (a), and the meaning of the expected input is also different. Figure 2 (c) shows the outcome of the program on each of the worlds if the user specifies S57, MN, red as input in each of the worlds. In world 1 the witness count for S57 is increased to 2, and in the remaining worlds a new tuple is added to the database.

Clearly, we cannot expect users to supply input for each world individually, or to deal with different output produced in each of the worlds. Such programs are not only unintuitive to the user: they are also infeasible to implement as in practice the number of possible worlds can be prohibitively large. One solution to this problem is to introduce an intermediate level between the database and the user that verifies that the messages (output and input) that are passed between the database and the user are the same *in all worlds*, and collapses those into one. This has the advantage of hiding the uncertain nature of the data from the users, allowing them to work with the database as if it were complete. We say that in this case the program is observationally deterministic. A second approach is to only allow programs that are *guaranteed* to have the same behavior on all worlds, and to verify observational determinism in a *static manner*. For example, the above program can be modified to first request all evidence information, and then carry out the update in each of the worlds by either updating existing tuples or inserting new ones, without disclosing this different behavior to the user.

The second major challenge in API development for uncertain databases is to map programs that are conceptually executed on each world individually into programs on the representation. There have been studies on how to do this efficiently for queries. Programs are more complex as they provide richer structure such as updates, looping and branching constructs and the flow of execution can be different in each world. We take the approach of first pushing as much of the program code as possible into (set-at-a-time) queries and updates, which we then map to queries and updates on representations.

In this paper we present novel techniques for database programming. The contributions of this paper are:

– **Updates.** Updating uncertain databases presents a challenge as often a compressed representation is used that stores a single copy of a tuple appearing in multiple worlds. Updates can require decompressing the representation to allow changing a tuple in some worlds only. We discuss techniques for implementing updates on several recent representation systems, such as the U-relations of MayBMS [1]. Our techniques preserve compactness of the representation despite the need for decompression.
– **Programming model.** We describe a programming model for developing applications for uncertain databases where users can interactively execute queries and updates on the database and process the results in a high-level language. Our model is independent of the underlying representation. For that we adopt the possible worlds semantics. Conceptually, programs run on all worlds in parallel.
– **Observational determinism.** We study a class of programs whose behavior on uncertain databases is indistinguishable to the user from that on complete/certain databases, without restricting the way programs behave in the background where no user interaction occurs. We call such programs observationally deterministic and argue that this property is crucial if we want to design efficient programs for uncertain databases with intuitive user-friendly interfaces. We also devise an efficient algorithm for deciding *statically* whether a program satisfies observational determinism. The underlying idea consists of examining the output sent to the user in terms of the query that produces it and checking whether the query can return uncertain results. As a side effect we obtain an efficient heuristic for deciding tuple q-certainty,

i.e. whether a tuple is certain in the answer to a query. The heuristic involves positive relational algebra operators only, which are often efficiently implementable on succinct representations.

– **Optimizing database programs.** Avoiding iteration over the possible worlds is crucial for achieving efficient execution. For that we need to map programs with update operations into ones that execute in bulk on all worlds. However, it is not clear how to deal with branching and for-loops in the programs, as the control flow can be very different in each world. For that we provide rewrite rules that map nested update programs into sequences of simple update programs that execute on all worlds. These results, while important in the context of uncertain databases, are relevant also in the case of certain databases.

2 Database Programming

A database programming model enables the development of applications that access and manipulate data stored in a database from a high-level programming language. A database program connects to a DBMS and can execute update commands, or issue queries and obtain cursors to iterate over the result. APIs provide means of accessing the data without knowing how the data is stored on disk, and often allow for porting programs between different database management systems. We next propose and study the properties of a programming model for uncertain data.

2.1 Queries and updates on uncertain DBMSs

We consider queries and updates specified using the SQL select, insert, update and delete statements. We take the possible worlds semantics to define the meaning of queries and updates. A query, applied on a set of possible worlds extends each world with the result of the query in that world. Similarly, an update operation is executed on each world of the world-set. For querying we will also consider the operator $conf$ for computing the confidence of the possible tuples in the result of a query. The confidence of a tuple t is defined as the sum of the probabilities of all worlds containing t. We also consider two special cases of this operator: *possible*, which computes all possible tuples (tuples with confidence greater than 0), and *certain*, which returns the tuples appearing in all possible worlds (i.e., confidence = 1), see e.g. [2].

There have been a number of studies on how to evaluate queries on uncertain databases by translating them into queries on the representation, see e.g. [6, 3, 1]. None of the present works however has considered the problem of applying updates on uncertain databases. As with querying, we would like to execute updates on the representation rather than iterate over the possible worlds.

Most systems for managing uncertain data use a compact representation for storing large sets of worlds. This usually means that tuple or attribute values that appear in several worlds are stored only once, with additional constraints that describe to which worlds they belong. Correlations are represented by means of lineage in Trio, using world-set descriptors in MayBMS, and with graphical models in [9].

U_1	D_1	TID	A
	$(x_1 \mapsto 1, 0.2)$	t_1	1
	$(x_1 \mapsto 2, 0.8)$	t_1	2
	$(x_2 \mapsto 1, 0.6)$	t_2	3
	$(x_2 \mapsto 2, 0.4)$	t_2	4

U_2	D_1	TID	B
	$(y_1 \mapsto 1, 0.1)$	t_1	1
	$(y_1 \mapsto 2, 0.9)$	t_1	2
	$(y_2 \mapsto 1, 1)$	t_2	2

(a)

U_1	D_1	D_2	TID	A
	$(x_1 \mapsto 1, 0.2)$	$(y_1 \mapsto 1, 0.1)$	t_1	8
	$(x_1 \mapsto 1, 0.2)$	$(y_1 \mapsto 2, 0.9)$	t_1	1
	$(x_1 \mapsto 2, 0.8)$	$(y_1 \mapsto 1, 0.1)$	t_1	8
	$(x_1 \mapsto 2, 0.8)$	$(y_1 \mapsto 2, 0.9)$	t_1	2
	$(x_2 \mapsto 1, 0.6)$		t_2	3
	$(x_2 \mapsto 2, 0.4)$		t_2	4

(b)

Fig. 3. Updating uncertain databases: (a) U-relational database; (b) relation U_1 after applying the update of Example 1.

We will use as running example the U-relational database of Figure 3 (a) representing a relation $R(A, B)$ in an uncertain database. It consists of two vertical partitions $U_1(D_1, TID, A)$ and $U_2(D_1, TID, B)$ containing the possible values for the A and the B attribute of R, respectively. The column D_1 in the two relations is used to specify correlations of the possible values. For example $t_1.A$ has a value of 1 whenever the variable x_1 is mapped to 1 (which happens with probability 0.2), and with probability 0.8 has value 2 whenever $x_1 \mapsto 2$. In this way the U-relational database represents compactly eight possible worlds, one for each of the possible combinations of values for the variables x_1, x_2, y_1, y_2.

Example 1. Let T1 be an operation that updates R:

```
T1: update R set A = 8 where B = 1;
```

This operation will update tuple t_1 for the worlds where $B = 1$. These worlds are constructed by taking y_1 to be 1. However, the current representation does not capture the desired correlation; therefore we need to create two copies of each of the alternatives of tuple t_1 in U_1 for the cases where it has to be updated or not. To compute the result we have to undo part of the decomposition, as shown in Figure 3 (b). □

Intuitively, each update operation consists of two steps: in the first one we create copies of those tuples that will need to be updated in some worlds only, and in the second step we execute the update. The first step only changes the representation, but not the world-set itself. We only decompress when it is necessary – when in some world the attribute value needs to be updated, and we do not merge in tuples that will not be updated in any world.

2.2 Programming interface

The programming language we will use is summarized in Table 1. As in the classical setting of certain databases, a database program is a sequence of statements

construct	**meaning**
update statements	Execution of SQL create table, insert, update and delete queries. The query statements can be constructed using constants, values read from the database or user-supplied values.
read(\$$x$,\$$x_{in}$);	Read user input into variable \$$x_{in}$. The user is displayed a prompt \$$x$.
write(\$x);	User output operation. The program can output messages to the user, including values stored in tuple variables.
+,-,*,/	Arithmetic operations on variables and constants.
for(\$t in Q){P}	Iterate over the result of a query Q and execute the nested program P for each binding of the tuple variable \$t. The query language we consider is an extension of SQL with the keywords **possible** and **certain**, that compute the tuples appearing in some, or all worlds, respectively, and the construct **conf** returning the confidence of a tuple in the result of a query.

Table 1. Language constructs for database programming.

that can execute select and update commands on the database, iterate over query results and provide user interaction through read and write commands.

How are programs executed on an uncertain database? We strive for a model for database programs for uncertain databases that satisfies the following desiderata. *First*, the execution model should be independent of the underlying uncertain database management system. This is important as it will allow porting of programs between different uncertain DBMSs. *Second*, programs should allow for efficient execution. *Third*, despite the fact that data is uncertain, programs should have intuitive user interfaces and should not expect users of the program to be aware of the uncertainty.

To satisfy the first requirement, we naturally adopt the possible worlds semantics that has been the standard semantics used to define the meaning of queries on uncertain databases. According to this, the program is executed in all worlds in parallel; within a world it behaves in the same way as on a complete database; all updates the program makes are applied to the current world. Of course, a direct implementation of this semantics is unrealistic as there are far too many worlds that can be represented by an uncertain database. In the context of querying several works have studied [6, 1, 3] how to avoid iterating over the possible worlds and evaluate a query directly on the representation. While for queries specified in a relational query language it is often possible to find an efficient translation of the query into one on the representation, a database programming language provides richer capabilities, such as executing updates, branching and user interaction that complicate the situation. In the next sections we study the implications of the requirements specified above. We will study criteria for programs to have an observationally deterministic behavior and will provide algorithms for optimizing database programs for set-based execution.

3 Optimizing database programs

The programming model of Section 2 is very powerful and allows for the writing of interesting but also potentially infeasible programs that access the database. We defined the semantics of a database program on uncertain databases to be the one where the program is executed on all worlds in parallel. A direct implementation of this semantics is not possible as uncertain DMBSs often represent compactly a large number of possible worlds. We would therefore like to execute programs in bulk on all worlds at the same time.

Previous work has studied how to translate queries on world-sets into ones on the representation, and in Section 2 we have seen how to do this for updates as well. A database program has richer constructs such as branching and loops and it is not clear how to map those to operations on the representation, as the flow of execution can be different on each world of the world-set. For that we will study techniques for unnesting programs, i.e. mapping programs to a sequence of read and write operations with no branching and loops. Another major issue is that a database program allows users to interact with the database via the read() and write() commands. This can cause problems whenever the user is returned output that is not the same in all worlds or when user is asked to supply different input in each world. Given that uncertain DBMSs often store an exponential number of worlds, such behavior is clearly unacceptable. We will therefore require that the uncertain database be observationally indistinguishable from a complete database (i.e. single-world database). We word this property *observational determinism*: a program is called *observationally deterministic* if the user interaction in terms of input and output of the program in one world is identical to the user interaction in all other worlds of the world-set.

We will first treat the problems of checking for observational determinism and of unnesting updates in isolation. We will start by discussing a method for *statically* checking whether a given program is observationally deterministic using a technique called *c-indicators*. We will then present rules that map update programs to linear sequences of update statements. The techniques from Sections 3.2 and 3.1 will be then used as building blocks of an algorithm that optimizes a database program containing both user interaction and updates.

3.1 Checking observational determinism

A program P is not observationally deterministic whenever it involves different user interaction in each world of the world-set. Let $x()$ be a user interaction operation that appears in P and let Q be the query that binds the values for $x()$. Then we can reason about whether $x()$ is the same in all worlds by checking whether the query Q produces only certain results. We illustrate the idea with the following examples.

Example 2. Consider for example an uncertain database that stores data as the *or-set relation* of Figure 4. Some fields of the table contain sets, the semantics being that one of the values in the set is the correct one for the respective

R	A	B	C
t_1	{1,2,3}	{4,5}	6
t_2	7	8	9
t_3	10	{11,12,13}	14

Fig. 4. Or-set relation

field. In our example the table represents 3 * 2 * 3 = 18 possible worlds over schema $R(A, B, C)$, one for each combination of values in the or-sets. Consider the following three programs that run on R[1]:

```
P1: for($t in "select * from R") write($t);
P2: for($t in "select possible * from R") write($t);
P3: for($t in select certain * from R where A=1
        union select * from R where A <> 1")
      if($t.A=1) write($t);
```

When executing $P1$ we run into the problem that the output is not the same in all worlds. On the other hand $P2$ is observationally deterministic, as in each world it outputs the possible tuples, i.e. the tuples that appear in at least one world. For our example the program will print 10 tuples in total: the different versions of t_1, t_2 and t_3. Deciding whether a program is observationally deterministic is not always simple. Consider $P3$: If we trace the origin of the tuples that are output by this program, we will see that these are exactly the certain tuples with A-value 1 in R. Thus the program satisfies observational determinism. $\square$

We next present our algorithm for deciding whether a program is observationally deterministic. We first construct a query corresponding to each user interaction (UI) operation in the program, and we couple this with a procedure for deciding whether a query produces only certain results.

Let us suppose that we have annotated the input database and we know which tuples are certain and which are not. More precisely, for each input relation R, we think of R as consisting of two disjoint partitions $R = R^c \cup R^u$, where R^c contains the certain tuples of R, and R^u: the tuples that appear only in some worlds. R^c and R^u can be computed with the queries $R^c = \text{certain}(R)$ and $R^u = R - \text{certain}(R)$. According to our semantics both queries R^c and R^u produce one result for each world, where the result of R^c is the same in all worlds, and the one of R^u is potentially different.

It is interesting to see how the annotations propagate from the input into the results of querying. For example a selection applied on a relation R can only discard tuples, but cannot make any uncertain tuples certain, and vice versa: no certain tuple will become uncertain. A 'possible' query will make all tuples from the input certain in the result since every world will contain those tuples.

Figure 5 defines an operator $[\![\cdot]\!]$ that takes a query expressed in positive relational algebra extended with the confidence computation predicate and its two special cases: possible and certain, and returns a pair of queries (Q^c, Q^u),

[1] While the programs are somewhat artificial, they are simple enough and help demonstrate the challenges when deciding whether a program is observationally deterministic.

$$\begin{aligned}
&\text{Let } R - \text{relation name}, \phi - \text{boolean condition}\\
Q, Q_1, Q_2 &- \text{queries in } RA^+ \cup \{\text{conf,possible,certain}\}\\
[\![R]\!] &:= (R^c, R^u)\\
[\![\pi_U(Q)]\!] &:= (\pi_U([\![Q]\!]^c), \pi_U([\![Q]\!]^u))\\
[\![\sigma_\phi(Q)]\!] &:= (\sigma_\phi([\![Q]\!]^c), \sigma_\phi([\![Q]\!]^u))\\
[\![Q_1 \bowtie_\phi Q_2]\!] &:= ([\![Q_1]\!]^c \bowtie_\phi [\![Q_2]\!]^c, [\![Q_1]\!]^u \bowtie_\phi [\![Q_2]\!]^u \cup [\![Q_1]\!]^c \bowtie_\phi [\![Q_2]\!]^u \cup [\![Q_1]\!]^c \bowtie_\phi [\![Q_2]\!]^u)\\
[\![Q_1 \cup Q_2]\!] &:= ([\![Q_1]\!]^c \cup [\![Q_2]\!]^c, [\![Q_1]\!]^u \cup [\![Q_2]\!]^u)\\
[\![\text{conf}(Q)]\!] &:= ([\![Q]\!]^c \cup \text{possible}([\![Q]\!]^u), \emptyset)\\
[\![\text{possible}(Q)]\!] &:= ([\![Q]\!]^c \cup \text{possible}([\![Q]\!]^u), \emptyset)\\
[\![\text{certain}(Q)]\!] &:= ([\![Q]\!]^c, \emptyset)
\end{aligned}$$

Fig. 5. Propagation of certainty during querying and update operations.

whose components compute the certain and the uncertain tuples in the result, respectively. This construction is conservative in the sense that it produces correct results but can omit some on particular inputs. This is the case for queries containing either a projection or a union operation. For example if we apply the projection $\pi_C(R)$ on the world-set represented as the or-set relation of Figure 4, all tuples in the result are certain although the only certain tuple in the input was t_2. Nevertheless, we shall see that for performing static checks, no other construction will produce better results and will be correct on all inputs.

Example 3. The query $Q = \sigma_{A=1}(\text{certain}(\sigma_{A=1}(R)) \cup \sigma_{A\neq1}(R))$ corresponds to the write statement of $P3$ of Example 2. Applying the construction of Figure 5 yields the pair of queries $(\sigma_{A=1}(R^c), \sigma_{A=1\wedge A\neq1}(R^u))$. Independent of the actual instance of the database, we can conclude that $P3$ is observationally deterministic, as the query defining the uncertain partition of the result has an unsatisfiable selection condition and will always return the empty set as result. $\square$

Using these ideas we can construct a procedure, called *c-indicator*, that "certifies" tuples that are certain in the result of a query.

Ideally we would like to design c-indicators that do not require evaluation of the whole query and then checking which tuples are certain in the output, but instead try to predict this information based on the query and possibly on some constraints that hold on the data. We can measure the quality of a c-indicator based on two criteria. The first one asks for soundness of the c-indicator, that is, that no false positives are produced. The second condition requires that the c-indicator is as close as possible in predicting which tuples are certain in the output. We formalize these requirements below.

Definition 1. We say that a c-indicator $\mathcal{C}$ is *sound* if for all queries Q, databases $\mathbf{A}$ and tuples t, if $\mathcal{C}(Q)(t, \mathbf{A}) = \mathit{true}$, then t is certain in $Q(\mathbf{A})$. A c-indicator $\mathcal{C}$ *dominates* another c-indicator $\mathcal{C}'$ ($\mathcal{C} \supset \mathcal{C}'$) iff for all tuples t, queries Q and

Algorithm 1: Checking observational determinism

```
Input: P: program
Output: true or false
foreach UI operation $x()$ in $P$ do
    $Q_x$ be the query corresponding to $x()$;
    if $[\![Q_x]\!]^u$ is satisfiable then
        return false;
    end
end
return true;
```

databases $\mathbf{A}$, if $\mathcal{C}'(Q)(t, \mathbf{A}) = true$, then $\mathcal{C}(Q)(t, \mathbf{A}) = true$, and there is a tuple t such that $\mathcal{C}(Q)(t, \mathbf{A}) = true$ and $\mathcal{C}'(Q)(t, \mathbf{A}) = false$. This means that $\mathcal{C}$ identifies strictly more tuples as being certain than $\mathcal{C}'$. A c-indicator $\mathcal{C}$ is *maximal* iff there is no other c-indicator $\mathcal{C}'$ for the same query Q such that $\mathcal{C}' \supset \mathcal{C}$.

There are two obvious solutions to the problem of constructing a c-indicator which is sound. The first one is a procedure that rejects all tuples and is therefore trivially guaranteed to produce no false positives. The second way is to enclose the input query Q in a 'certain' construct and check whether the condition $t \in \text{certain}(Q)$ is satisfied. This c-indicator will not only be sound but maximal as well. However, deciding tuple Q-certainty, that is, whether a tuple is certain in the result of a query is coNP-hard on succinct representation systems [6]. Succinct representation systems are such that can represent an exponentially large, or even infinite set of worlds. For example, the tuple-independent model of [4] can represent 2^n possible worlds using n tuples only. Ideally we would like to have c-indicators that use relational algebra only. We will relax this condition to allow querying certain tuples in the input relations. Intuitively, we can annotate the certain tuples once at the beginning, and then incrementally maintain the annotations when the database is updated.

Theorem 1. *For each query Q expressed in positive relational algebra with possible and certain there exists a c-indicator using relational algebra operators only which is sound and maximal when we assume no knowledge about the data.*

Proof (sketch). Using the construction $[\![\cdot]\!]$ of Figure 5, we define a c-indicator $\mathcal{C}^{pr}$ (*pr* stands for propagating uncertainty, the idea used in defining $[\![\cdot]\!]$) as the following test: $\mathcal{C}^{pr}(Q)(t, \mathbf{A}) := \{t \in [\![Q]\!]^c(\mathbf{A})\}$.

$\mathcal{C}^{pr}$ is sound and maximal. Let $\mathbf{A}$ be an uncertain database, Q be a query expressed in positive relational algebra with possible and certain over the schema of $\mathbf{A}$ and t be a tuple in the schema of Q. By induction on the structure of the query Q we show that if $t \in [\![Q]\!]^c(\mathbf{A})$, then t is certain in $Q(\mathbf{A})$. On the other hand, if $t \notin [\![Q]\!]^c(\mathbf{A})$, then there is a witness world-set $\mathbf{B}$ such that t is not possible in $Q(\mathbf{B})$. If Q is a positive relational algebra query, we can take as witness the world-set $\mathbf{B}_0$ containing one world where all relations are empty.

Finally, using the idea of c-indicators we construct a procedure that automatically decides whether a given program is observationally deterministic or not on all inputs. For each UI operation $x()$ of the given program P we compute the query Q_x that corresponds to $x()$ by composing the for-loop statements that are ancestors of $x()$ in the parse tree of P. Algorithm 1 rewrites Q_x using the c-indicator rules of Figure 5 and checks whether the uncertain partition of that query is satisfiable.

Theorem 2. *Algorithm 1 rejects all programs that are not observationally deterministic.*

3.2 Unnesting updates

We next study how an update program can be turned into a sequence of update statements without for-loops or if-conditions.

Example 4. Consider a modification of the program from Section 1 that satisfies observational determinism:

```
read("Enter license plate, location and color:", $x,$loc,$col);
if (select * from cars where num=$x != NULL) {
  if (select * from cars where num=$x and loc=$loc and color=$col != NULL)
    update cars set wit=wit+1 where num=$x and loc=$loc and color=$col;
  else insert into cars values($x,$loc,$col,1);
}
else insert into cars values ($x,$loc,$col,1);
```

We can linearize the program by mapping the if-else block to the following three update statements:

```
update cars set wit=wit+1
where num=$x and loc=$loc and color=$col;
insert into cars select $x,$loc,$col,1
where not exists (select * from cars where num=$x and loc=$loc and
      color=$col) and exists (select * from cars where num=$x);
insert into cars select $x,$loc,$col,1
where not exists (select * from cars where num=$x);
```

This program is equivalent to the first one. Using the techniques of Section 2 we can translate it into a program on the representation, which then executes on all worlds at the same time. □

As seen in the above example we can often push if-conditions into the where-clause of an update operation. For this to work however we need to restrict the update such that it does not interfere with subsequent operations necessary to evaluate a query. For example if we exchange the first and the second update statement of the second program we will obtain a different result, although exchanging the if and the else block of the first program does not change its semantics.

We next formalize rules for unnesting update programs. We consider a somewhat simplified version of the API from Section 2, where the control structures are only for-loops and updates are of the following kind. Let R be a relation, Let Q be a query, $U_1, \ldots, U_n$ be update operations, and Q_ϕ be a semi-join query on R

(1)for($\$t$ in Q)$\langle \overline{A} \mapsto \overline{c} \mid Q_\phi \rangle \vdash$
$\langle \overline{A} \mapsto \overline{c} \mid Q_{\phi'} \rangle$, where $Q_{\phi'} = \{r \mid r \in R \wedge t \in Q \wedge \phi'\}$ and ϕ' is obtained from ϕ by replacing all occurrences of $\$t$ by t

(2) for($\$t$ in Q)$\{U_1; U_2; \ldots; U_n\} \vdash$
for($\$t$ in Q)$\{U_1;\}$; for($\$t$ in Q)$\{U_2; \ldots; U_n;\}$

Fig. 6. Rules for unnesting update programs.

$\{A_1, \ldots, A_m\} \subseteq \mathbf{sch}(R)$, $c_1, \ldots, c_n$ be constants and ϕ be a condition involving constants and attributes of R. We will restrict ourselves to updates that add tuples of constant values, or change tuples fields to constant values. We consider updates of the form:

$$\boxed{\text{update R set } \overline{A} = \overline{c} \text{ where } \phi;}$$

where $\overline{A} = \overline{c}$ is a shortcut for $A_1 = c_1, \ldots, A_m = c_m$.

Let Q_ϕ denote the semi-join query returning the tuples of R that need to be updated. We will use the notation $U = \langle \overline{A} \mapsto \overline{c} \mid Q_\phi \rangle$ for update operations.

To define rules for optimizing programs we rely on the independence of queries from the updates:

Definition 2. Let Q be a query and U be an update operation. For a database $\mathcal{A}$ let $U(\mathcal{A})$ denote the database obtained as a result of executing U on $\mathcal{A}$. We say that Q is *independent* of U iff for any input database $\mathcal{A}$ $Q(\mathcal{A}) = Q(U(\mathcal{A}))$. Similarly, we say that update U_1 is *independent* of another update U_2 if for any input database $\mathcal{A} : U_1(\mathcal{A}) = U_1(U_2(\mathcal{A}))$.

Figure 6 shows two rewrite rules that can be applied iteratively to optimize a database program. Recall that SQL updates have transactional semantics: changes made by update U are not visible to the update condition before the end of the update operation. However, if the update is nested within a for-loop, U will in general be executed multiple times - once for each result tuple returned by Q. Thus changes made by U will be visible in subsequent loop iterations. We next discuss when these rewrite rules produce equivalent programs.

Lemma 1. *(a) If Q_ϕ is independent of the update $U = \langle \overline{A} \mapsto \overline{c} \mid Q_\phi \rangle$, then rule (1) preserves equivalence.*

(b) If Q_ϕ is independent of U_1, U_1 is independent of $U_i, 2 \leq i \leq n$ and U_i is independent of $U_1, 2 \leq i \leq n$, then rule (2) preserves equivalence.

Proof. (a) Let P and P' denote the programs on the lhs and rhs of rule (1), respectively. Let $r \in R$ be a tuple that is updated in P and let this occur when iterating over tuple t from Q. Then $r \in Q_{\phi(t)}(\mathcal{A}')$ where $\phi(t)$ is the condition obtained by substituting the variable $\$t$ in ϕ with the values from t and $\mathcal{A}'$ is the state of the database at the beginning of that iteration of the loop. Since Q_ϕ is independent of U, $Q_\phi(\mathcal{A}') = Q_\phi(U(\mathcal{A}')) = Q_\phi(\mathcal{A})$, where $\mathcal{A}$ is the initial database before the start of P. But then this is equivalent to $r \in Q_{\phi'}(\mathcal{A})$, where $Q_{\phi'} = \{r \in R \wedge t \in Q \wedge \phi'\}$ is the query on the rhs of Rule (1). Thus r is also updated in P', and P and P' are equivalent.

(b) Let P and P' be the programs on the lhs and rhs of Rule (2), respectively. Suppose U_i is updating relation R_i with $R_1, \ldots, R_n$ not necessarily disjoint. Since Q is independent of U_1, each loop in P' performs exactly the same iterations as the loop in P. In addition, since U_1 is independent of $U_2, \ldots, U_n$, a tuple $r \in R_1$ is updated by U_1 in P iff it is updated by U_1 in P'. Similarly, since each U_j, $2 \leq j \leq n$ is independent of U_1, a tuple $r \in R_j$ is updated by U_j in P iff it is updated by U_j in P'.

The correctness proof for Rule (2) does not use the fact that attribute values can be changed to constants only. Under the independence assumptions of Lemma 1 the rule remains correct when updates can change fields to arbitrary values, not only constants. We next discuss the implications for Rule (1) of allowing variables to appear in the set list. Consider for example the following update block, where x_i is either a constant or a reference of the form $\$t.A$:

$$\text{for}(\$t \text{ in } Q)\{\text{update R set } A_1 = x_1, \ldots, A_m = x_m \text{ where } \phi\}$$

In such an update block it is possible that the same field is set to two different values in two different loop iterations. Hence the final value of the field will depend on the order of reading the for-loop tuples, which is often undesirable. We require that this never happens, that is, a field is always set to the same value or is left unchanged. We can then generalize rule (1) for the case where variables can appear in the set-clause of an update statement.

Checking independence of queries from updates. The problem of statically deciding whether a query is independent from an insertion or deletion update has been studied in [5] and [7]. The proposed solution consists in checking equivalence of two programs: one that computes the query answer before the update, and one after the update. In our setting we are also considering updates specified with an SQL update statement. Since those can be simulated with a pair of insert/delete, one can reduce the problem of deciding independence from a general update statement to independence of insertion and deletions. Note however that for Rule 1 we need to check independence of the query corresponding to the where clause of an update statement from the update itself. Thus deciding independence at the level of inserts and deletes will be unnecessarily restrictive, as it will always return a negative answer. We can use a simpler but more precise condition to check update independence, namely: if no attribute appears both on the left side of the set-clause and in the where-clause of an update statement, then the update query is independent of the update.

3.3 Rewriting database programs for bulk execution

We will consider database programs where both updates and user interaction commands can be nested. Let P be the parse tree for a program, where P's

Algorithm 2: Optimize programs

```
Input: $P$: program
Output: $P'$: program equivalent to $P$ that can be executed on all worlds or
        FAIL.
foreach maximal subtree $P_0$ of $P$ with no UI operations do
    Let L:=unnest($P_0$);
    return 'FAIL' if $P_0$ cannot be unnested;
    Replace $P_0$ by L in $P$;
end
return 'FAIL' if $P$ is not observationally deterministic;
Otherwise return $P$;
```

nodes are sequences of for-loops, update statements and user interaction (UI) commands. Algorithm 2 shows an algorithm that optimizes a program for bulk execution on all worlds, or returns 'FAIL' if no optimization is found. On success the algorithm returns a program that satisfies the following two conditions:

1. All update operations are on the top-level or are nested within loops that operate on certain query results only.
2. The program is observationally deterministic.

The algorithm considers all maximal rooted subtrees of the parse tree for P that do not contain any UI operations. For those the algorithm applies the unnesting techniques (presented in the previous subsection) to turn them into flat sequences of update statements. Let P' be the program that results from this step. If we can verify that P' is observationally deterministic, then P' is the result of the optimization procedure. Finally, we can state our main result:

Theorem 3. *If Algorithm 2 returns program P', P' is equivalent to the input program P, satisfies observational determinism, and all update operations are either on the top level or are nested in for-loops that iterate over certain results.*

References

1. L. Antova, T. Jansen, C. Koch, and D. Olteanu. "Fast and Simple Relational Processing of Uncertain Data". In *Proc. ICDE*, 2008.
2. L. Antova, C. Koch, and D. Olteanu. "From Complete to Incomplete Information and Back". In *Proc. SIGMOD*, 2007.
3. O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. "ULDBs: Databases with Uncertainty and Lineage". In *Proc. VLDB*, 2006.
4. N. Dalvi and D. Suciu. "Efficient query evaluation on probabilistic databases". In *Proc. VLDB*, 2004.
5. C. Elkan. Independence of logic database queries and updates. In *Proc. PODS*, 1990.
6. T. Imielinski and W. Lipski. "Incomplete information in relational databases". *Journal of ACM*, **31**(4), 1984.
7. A. Y. Levy and Y. Sagiv. Queries independent of updates. In *Proc. VLDB*, 1993.

8. C. Re, N. Dalvi, and D. Suciu. "Efficient Top-k Query Evaluation on Probabilistic Data". In *Proc. ICDE*, 2007.
9. P. Sen and A. Deshpande. "Representing and Querying Correlated Tuples in Probabilistic Databases". In *Proc. ICDE*, 2007.
10. M. Soliman, I. Ilyas, and K. C. Chang. "Top-k Query Processing in Uncertain Databases". In *Proc. ICDE*, 2007.