An Introduction to ULDBs and the Trio System*

Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Jennifer Widom
Stanford University
{benjello,anish,hayworth,widom}@cs.stanford.edu

Abstract

We introduce ULDBs*: relational databases that add* uncertainty *and* lineage *of the data as first-class concepts. The ULDB model underlies the* Trio *system under development at Stanford. We describe the ULDB model, then present* TriQL*, our SQL-based query language for ULDBs. TriQL's semantics over ULDBs is defined both formally and operationally, and TriQL extends SQL with constructs for querying lineage and confidence values. We also briefly describe our initial prototype Trio implementation, which encodes ULDBs in conventional relations and automatically translates TriQL queries into SQL commands over the encoding. We conclude with research directions for ULDBs and the Trio system.*

1 Introduction

In the *Trio* project at Stanford, we are developing a new kind of database management system (DBMS): one in which *data*, *uncertainty* of the data, and data *lineage* are all first-class citizens in an extended relational model and SQL-based query language. In an earlier paper [Wid05], we motivated the need for these three aspects to coexist in one system and detailed numerous potential applications including scientific and sensor data management, data cleaning and integration, information extraction systems, and others. For examples in this paper we use a highly simplified "crime-solver" application with just two base tables: `Owns(owner,car)` and `Saw(witness,car)`, capturing (possibly uncertain) car ownership information and crime-vehicle sightings.

We call the type of relational database managed by Trio a *ULDB*, for *Uncertainty-Lineage Database*. To the best of our knowledge, ULDBs are the first database formalism to integrate uncertainty and lineage. In the rest of this section we briefly motivate the concepts and survey related work.

Uncertainty. Uncertainty is captured by tuples that may include several *alternative* possible values, with optional *confidence* values associated with each alternative. For example, if a witness saw a vehicle that was a Honda with confidence 0.5, a Toyota with confidence 0.3, or a Mazda with confidence 0.2, the sighting yields one tuple in table `Saw` with three alternative values. Furthermore, the presence of tuples may be uncertain, again with optionally specified confidence. For example, another witness may have 0.6 confidence that she saw a crime vehicle, but if she saw one it was definitely an Acura. Based on alternative tuple values and confidences, each ULDB represents multiple *possible instances* of a database.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work was supported by the National Science Foundation under grants IIS-0324431 and IIS-1098447, and by a grant from the Boeing Corporation.

Lineage. Lineage, sometimes called *provenance*, associates with a data item information about its derivation. Broadly, lineage may be *internal*, referring to data within the ULDB, or *external*, referring to data outside the ULDB, or to other data-producing entities such as programs or devices. As a simple example of internal lineage, we may generate a table `Suspects` by joining tables `Saw` and `Owns` on attribute `car`. Lineage associated with a value in `Suspects` identifies the `Saw` and `Owns` data from which it was derived. A useful feature of internal lineage is that the confidence of a value in `Suspects` can be computed from the confidence of the data in its lineage.

As an example of external lineage, `Owns` may be populated from various car registration databases, and lineage can be used to connect the data to its original source. Many varieties of lineage are discussed in [Wid05]. In this paper we focus on simple internal lineage.

Queries. We will present a precise semantics for relational queries over ULDBs in Section 2, and in Section 3 an operational description of our SQL-based query language that conforms to the semantics. Intuitively, the result of a relational query Q on a ULDB U is a result R whose possible instances correspond to applying Q to each possible instance of U. Internal lineage connects the data in result R with the data from which it was derived, as in the `Suspects` join query introduced in the "Lineage" discussion above. Confidence values in query results are defined in a standard probabilistic fashion.

TriQL (pronounced "treacle"), Trio's query language, adapts SQL to our possible-instances semantics in a straightforward and natural manner. TriQL also includes simple but powerful constructs for querying lineage (e.g., "find all witnesses contributing to our suspicion of `Jimmy`"), querying uncertainty (e.g., "find all high-confidence sightings"), or querying both together (e.g., "find all suspects whose lineage contains low-confidence sightings or ownerships").

Prototype. In our initial prototype, Trio is built on top of a conventional relational DBMS, although as we delve more into storage, access method, and query optimization issues, we are likely to work more and more inside the system. Currently, we encode ULDBs as conventional relations, with a two-way translation layer so users can view and manipulate the ULDB model without being aware of the encoding. TriQL queries are translated automatically to SQL commands over the encoding, and the translation is not difficult or complex. Confidence and lineage are queried through simple built-in functions and predicates. Furthermore, lineage enables confidence values in query results to be computed *lazily*, a noteworthy feature of our approach. The prototype is described in more detail in Section 4.

1.1 Related Work

Our initial motivation to pursue this line of research, described in [Wid05], was followed by an exploration of the space of models for uncertainty with an emphasis on usability [DBHW06], and more recently a study of several new theoretical problems in this space [DNW05]. We subsequently added lineage to uncertainty, proposing and formalizing ULDBs in [BDHW05]. These papers all contain more extensive discussion of related work than we have room for here.

There has been a large body of previous working studying representation schemes and query answering for uncertain databases, including but certainly not limited to [AKG91, BGMP92, BP82, BP93, FR97, Gra84, Gra89, IL84, LLRS97, Var85]. A recent paper [BDM$^+$05] describes a system for handling imprecisions in data, and the same research group has made considerable progress in query answering for probabilistic databases [BDM$^+$05, DMS05, DS04, DS05].

Data lineage was introduced for a scientific data visualization system [WS97], and has been studied for conventional relational databases, e.g., [BKT01, BKT00, BKT02], and for data warehouses, e.g., [CW00, CW03, CWW00]. References [BCTV04, CTV05] describe a recent system being developed around data provenance (lineage).

2 ULDBs: Uncertainty-Lineage Databases

We present the ULDB model primarily through examples. A more formal treatment appears in [BDHW05]. ULDBs extend the standard SQL (multiset) relational model with: (1) *alternatives*, representing uncertainty about the contents of a tuple; (2) *maybe* ('?') annotations, representing uncertainty about the presence of a tuple; (3) numerical *confidence* values optionally attached to alternatives and '?'; and (4) *lineage*, connecting tuple alternatives to other alternatives from which they were derived. We next discuss each of these four constructs, then we define the semantics of relational queries on ULDBs.

2.1 Alternatives

ULDB relations are comprised of *x-tuples* (and therefore are called *x-relations*). Each x-tuple consists of one or more *alternatives*, where each alternative is a regular tuple over the schema of the relation. For example, if a witness Amy saw either a Honda, Toyota, or Mazda, then in table `Saw` we have:

(witness, car)
`(Amy,Honda)` ‖ `(Amy,Toyota)` ‖ `(Amy,Mazda)`

This x-tuple logically yields three *possible instances* for table `Saw`, one for each alternative. In general, the possible instances of an x-relation R correspond to all combinations of alternatives for the x-tuples in R. For example, if a second tuple in `Saw` had four alternatives, then there would be 12 possible instances altogether.

Clearly the x-tuple above can be represented more naturally with attribute-level instead of tuple-level uncertainty. Using set notation to denote "one of" we could write:

witness	**car**
`Amy`	{`Honda,Toyota,Mazda`}

As in [DBHW06, Wid05], we call this attribute-level construct an *or-set*. Or-sets can be a convenient compact representation for x-tuples, e.g., for readability or for space-efficient storage, and we plan to support them in Trio for these reasons. However or-sets are less expressive than x-tuples: If a tuple contains or-sets in multiple attributes, the alternatives of the x-tuple it represents are all possible combinations of the values in each of the or-sets, i.e., dependencies across attributes cannot be expressed using or-sets.

2.2 '?' (Maybe) Annotations

Suppose a second witness, Betty, thinks she saw a car but is not sure. However, if she saw a car, it was definitely an Acura. In ULDBs, uncertainty about the existence of a tuple (more generally of an x-tuple) is denoted by a '?' annotation on the x-tuple. Betty's observation is thus added to table `Saw` as:

(witness, car)	
`(Amy,Honda)` ‖ `(Amy,Toyota)` ‖ `(Amy,Mazda)`	
`(Betty,Acura)`	**?**

The '?' on the second x-tuple indicates that this entire tuple may or may not be present (so we call it a *maybe x-tuple*). Now the possible instances of an x-relation include not only all combinations of alternatives, but also all combinations of inclusion/exclusion for the maybe x-tuples. This `Saw` table has six possible instances: three choices for Amy's car times two choices for whether or not Betty saw an Acura. For example, one possible instance of `Saw` is the two tuples `(Amy,Honda)`, `(Betty,Acura)`, while another instance is just `(Amy,Mazda)`.

2.3 Confidences

Numerical *confidence* values may be attached to the alternatives of an x-tuple. Suppose Amy's confidence in seeing a Honda, Toyota, or Mazda is 0.5, 0.3, and 0.2 respectively, and Betty's confidence in seeing a vehicle is 0.6. Then we have:

(witness, car)	
(Amy,Honda):0.5 ‖ (Amy,Toyota):0.3 ‖ (Amy,Mazda):0.2	
(Betty,Acura):0.6	?

In [BDHW05] we formalize an interpretation of these confidences in terms of probabilities. (Other interpretations may be imposed, but the probabilistic one is the current default for Trio.) Thus, if Σ is the sum of confidences for the alternatives of an x-tuple, then we must have $\Sigma \leq 1$, and if $\Sigma < 1$ then the x-tuple must have a '?'. Implicitly, '?' is given confidence $(1 - \Sigma)$ and denotes the probability that the tuple is not present.

Now each possible instance of an x-relation itself has a probability, defined as the product of the confidences of the alternatives and '?''s comprising the instance. It can be shown that for any x-relation: (1) The probabilities of all possible instances sum to 1; and (2) The confidence of an x-tuple alternative (respectively a '?') equals the sum of probabilities of the possible instances containing this alternative (respectively not containing any alternative from this x-tuple).

An important special case of ULDBs is when every x-tuple has only one alternative with a confidence value that may be < 1. This case corresponds to the traditional notion of probabilistic databases, as in, e.g., [BGMP92, CP87, DS04, LLRS97]. For simplicity in subsequent discussions we assume each ULDB includes confidences on all of its data or none of it, although "mixing and matching" is generally straightforward.

2.4 Lineage

Lineage in ULDBs is recorded at the granularity of tuple alternatives: lineage connects an x-tuple alternative to other x-tuple alternatives.[1] Specifically, we define lineage as a function λ over alternatives, such that $\lambda(t)$ gives the set of alternatives from which the alternative t was derived.

Consider again the join of `Saw` and `Owns` on attribute `car`, followed by a projection on `owner` to produce a table `Suspects(person)`. Let column ID contain a unique identifier for each x-tuple, and let (i, j) denote the jth alternative of the x-tuple with identifier i. Here is some sample data for all three tables, including lineage for the derived data in `Suspects`. For example, the lineage of Jimmy's presence in table `Suspects` is the second alternative of tuple 51 in table `Saw`, together with the second alternative of tuple 61 in table `Owns`.

ID	Saw (witness, car)
51	(Cathy,Honda) ‖ (Cathy,Mazda)

ID	Owns (owner, car)	
61	(Jimmy,Toyota) ‖ (Jimmy,Mazda)	?
62	(Billy,Honda)	
63	(Hank,Honda)	

ID	Suspects (person)		
71	Jimmy	?	λ (71,1) = { (51,2), (61,2) }
72	Billy	?	λ (72,1) = { (51,1), (62,1) }
73	Hank	?	λ (73,1) = { (51,1), (63,1) }

[1] Recall we are considering only internal lineage in this paper. We expect that many types of external lineage will also be recorded at the tuple-alternative granularity, although for some lineage types coarser granularity will be more appropriate [Wid05].

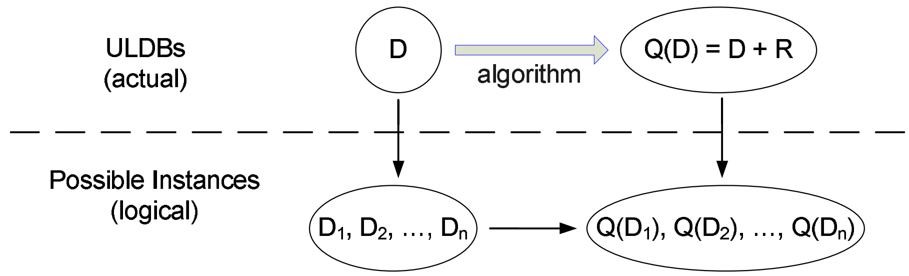

Figure 1: Relational queries on ULDBs.

An interesting and important effect of lineage is that it imposes restrictions on the possible instances of a ULDB. Consider the derived table `Suspects`. Even though there is a '?' on each of its three tuples, not all combinations are possible. If Billy is present in `Suspects` then alternative 1 must be chosen for tuple 51, and therefore Hank must be present as well. Jimmy is present in `Suspects` only if alternative 2 is chosen for tuple 51, in which case neither Billy nor Hank can be present. Note that choosing alternative $(51, 2)$ does not guarantee Jimmy's presence in `Suspects`, since tuple 61 has a '?'. Roughly speaking, a tuple alternative is present in a possible instance if and only if all of the alternatives in its lineage are present, although the actual constraints are somewhat more complex; see [BDHW05] for details.

In summary, once we add lineage, not all combinations of alternatives and '?' choices correspond to valid possible instances. The above ULDB has six possible instances, determined by the two choices for tuple 51 times the three choices (including '?') for tuple 61. Note that arbitrary lineage functions may not "work" under our model—consider for example a tuple whose lineage (directly or transitively) includes two different alternatives of the same x-tuple. In [BDHW05] we formally define *well-behaved* lineage and show that internal lineage generated by queries is always well-behaved. Under well-behaved lineage, the possible instances of an entire ULDB correspond to the possible instances of the "base" data (data with no lineage of its own), as seen in the example above. Now, with well-behaved lineage our interpretation of confidences carries over directly: combining confidences on the base data determines the probabilities of the possible instances, just as before. We will discuss confidences on derived data in Section 4.

2.5 Relational Queries

In this section we formally define the semantics of any relational query over a ULDB. Trio's SQL-based query language is presented in Section 3. The semantics for relational queries over ULDBs is quite straightforward but has two parts: (1) the possible-instances interpretation; and (2) lineage in query results.

Refer to Figure 1. Consider a ULDB D whose possible instances are $D_1, D_2, \ldots, D_n$, as shown on the left side of the figure. If we evaluate a query Q on D, the possible instances in the result of Q should be $Q(D_1), Q(D_2), \ldots, Q(D_n)$, as shown in the lower-right corner. For example, if a query Q joins x-relations `Saw` and `Owns`, then logically it should join all of the possible instances of these two x-relations. Of course we will not actually generate all possible instances and operate on them, so a query processing algorithm follows the top arrow in Figure 1, producing a ULDB query result $Q(D)$ that represents the possible instances.

In our model, a query result is more than just a set of x-tuples: $Q(D)$ contains the original x-relations of D, together with a new *result x-relation* R. Lineage from R into the x-relations of D reflects the derivation of the data in R. This approach is necessary for $Q(D)$ to represent the correct possible instances in the query result,[2] and to enable consistent further querying of the original and new x-relations. Our example in the previous subsection with `Suspects` as the result of query $Q = \pi_{\texttt{owner}}(\texttt{Saw} \bowtie \texttt{Owns})$ demonstrates the possible-instances

[2]Technically, the possible instances in the lower half of Figure 1 also contain lineage, but this aspect is not critical here; see [BDHW05] for formal details.

interpretation and lineage from query result to original data; details are left as a useful exercise for the reader.

The lineage of tuples in query results can be defined in different ways, as explored in other work [BKT01]. So far, we have considered a class of queries (subsuming SPJ queries and union, formally defined in [BDHW05]) for which the lineage is straightforward: every result tuple alternative t identifies a unique combination of base tuple alternatives from which t was derived.

3 TriQL: The Trio Query Language

In this section we describe *TriQL*, Trio's SQL-based query language. Except for built-in functions and predicates for querying confidence values and lineage, TriQL uses the same syntax as SQL. However, the interpretation of SQL queries must be modified to reflect the semantics over ULDBs discussed in the previous section.

In this paper (and in our current implementation), we limit ourselves to single-block queries without aggregation or `DISTINCT`. Thus, the general form of a TriQL query Q is:

```
SELECT attr-list [ INTO new-table ]
FROM X1, X2, ..., Xn
WHERE predicate
```

As an example, our join query producing `Suspects` is written in TriQL exactly as expected:

```
SELECT Owns.owner as person INTO Suspects
FROM Saw, Owns
WHERE Saw.car = Owns.car
```

If we execute this query as regular SQL over each of the possible instances of `Saw` and `Owns`, as in the lower portion of Figure 1, it produces the expected set of possible instances in its result. More importantly, following the operational semantics of TriQL given next, this query produces a result table `Suspects`, including lineage to tables `Saw` and `Owns`, that correctly represents those possible instances.

3.1 Operational Semantics

We provide an operational description of TriQL by specifying direct evaluation of an arbitrary TriQL query over a ULDB, corresponding to the upper arrow in Figure 1. Consider the generic TriQL query block above. Let *schema*(Q) denote the composition *schema*(`X1`) $\uplus$ *schema*(`X2`) $\uplus \cdots \uplus$ *schema*(`Xn`) of the `FROM` relation schemas, just as in SQL query processing. The `predicate` is evaluated over tuples in *schema*(Q), and the `attr-list` is a subset of *schema*(Q) or the symbol "$\star$", again just as in SQL.

The steps below are an operational description of evaluating the above query block. As in SQL database systems, a query processor would rarely execute the simplest operational description since it could be woefully inefficient, but any query plan or execution technique (such as our translation-based approach described in Section 4) must produce the same result as this description.

1. Consider every combination $x_1, x_2, \ldots, x_n$ of x-tuples in `X1`, `X2`, ..., `Xn`, one combination at a time, just as in SQL.

2. Form a "super-tuple" T whose alternatives have schema *schema*(Q). T has one alternative for each combination of alternatives in $x_1, x_2, ..., x_n$.

3. If any of $x_1, x_2, \ldots, x_n$ has a '?', add a '?' to T.

4. Set the lineage of each alternative in T to be the alternatives in $x_1, x_2, \ldots, x_n$ from which it was constructed.

5. Retain from T only those alternatives satisfying the `predicate`. If no alternatives satisfy the predicate, we're finished with T. If any alternative does not satisfy the predicate, add a '?' to T if it is not there already.

6. If we are operating on a ULDB with confidences, either compute the confidence values for T's remaining alternatives and store them (*eager confidence computation*), or set the confidence values to `NULL` (*lazy confidence computation*). See Section 4 for further discussion.

7. Project each alternative of T onto the attributes in `attr-list`, generating an x-tuple in the query result. If there is an `INTO` clause, insert T into table `new-table`.

We leave it to the reader to verify that this operational semantics produces the `Suspects` result table shown with example data in Section 2.4, and more generally that it conforms to the formal semantics given in Section 2.

3.2 Querying Confidences

TriQL provides a built-in function `conf()` for accessing confidence values. Suppose we want our `Suspects` query to only use sightings having confidence > 0.5 and ownerships having confidence > 0.8. We write:

```
SELECT Owns.owner as person INTO Suspects
FROM Saw, Owns
WHERE Saw.car = Owns.car AND conf(Saw) > 0.5 AND conf(Owns) > 0.8
```

In the operational semantics, when we evaluate the `predicate` over the alternatives in T in step 6, `conf(Xi)` refers to the confidence associated with the x_i component of the alternative being evaluated. Note that this function may trigger confidence computations in the lazy case.

3.3 Querying Lineage

For querying lineage, TriQL introduces a built-in predicate designed to be used as a join condition. If we include predicate `lineage`(R, S) in the `WHERE` clause of a TriQL query with x-relations R and S in its `FROM` clause, then we are constraining the joined R and S tuple alternatives to be connected by lineage. For example, suppose we want to find all witnesses contributing to Hank being a suspect. We can write:

```
SELECT Saw.witness INTO AccusesHank
FROM Suspects, Saw
WHERE lineage(Suspects,Saw) AND Suspects.person = 'Hank'
```

In the `WHERE` clause, `lineage(Suspects,Saw)` evaluates to true for any pair of alternatives x_1 and x_2 from `Suspects` and `Saw` such that x_1's lineage includes x_2. Of course we could write this query directly on the base relations if we remembered how `Suspects` was computed, but the `lineage()` predicate provides a more general construct that is insensitive to query history.

With the addition of derived x-relation `AccusesHank`, we now have two layers of lineage: from `AccusesHank` to `Suspects`, then from `Suspects` to the base x-relations. In general, over time a ULDB may develop many levels of lineage, so we also have a *transitive lineage* predicate: `lineage*`(R, S) evaluates to true for any pair of alternatives x_1 and x_2 from R and S such that there is a "path" via lineage from x_1 to x_2. This predicate introduces some interesting evaluation issues. Our current prototype keeps track of lineage structure and uses it to translate `lineage*()` into joins with `lineage()` predicates.

As a final TriQL example incorporating both lineage and confidence, the following query finds persons who are suspected based on high-confidence ownership of a Honda:

```
SELECT Owns.owner INTO HighHonda
FROM Suspects, Owns
WHERE lineage(Suspects,Owns) AND Owns.car = 'Honda' AND conf(Owns) > 0.8
```

4 Current Implementation

Our first-generation Trio prototype, which is up and running, is built on top of a conventional relational DBMS: ULDBs are represented in relational tables, and TriQL queries and commands are rewritten automatically into SQL commands evaluated against the representation.

The core system is implemented in Python and presents a simple API that extends the standard Python DB 2.0 API for database access (Python's analog of JDBC). The Trio API supports TriQL queries instead of SQL, query results are cursors enumerating x-tuple objects instead of regular tuples, and x-tuple objects provide programmatic access to their alternatives, including confidences and lineage. The first application we built using the Trio API is a generic command-line interactive client, similar to that provided by most DBMS's.

We are currently building on the *Postgres* open-source DBMS. However, we intentionally rely on very few Postgres-specific features, so porting to any other DBMS providing a DB 2.0 API should be straightforward. The next two sections describe our representation of ULDBs as conventional relations, and our rewrite techniques for query processing.

4.1 Representing ULDBs as Relations

Let $R\,(A_1, \ldots, A_n)$ be an x-relation with lineage and possibly confidences. We store the data portion of R as a conventional table (which we also call R) with three extra attributes: $R\,($`aid`, `xid`, `conf`, $A_1, \ldots, A_n)$. Each alternative in the original x-relation is stored as its own tuple in R, with a unique identifier `aid`. The alternatives of each x-tuple have the same x-tuple identifier, `xid`. Finally, `conf` stores the confidence of the alternative, with three special values: `NULL` denotes that the alternative's confidence has not yet been computed (but can be computed from the alternative's lineage); -1 denotes that the alternative does not have a confidence value but its x-tuple has a '?'; -2 denotes that the alternative does not have a confidence value and its tuple does not have a '?'. If an alternative has `conf` = -1 then so must all other alternatives with the same `xid`; similarly for `conf` = -2.

The lineage information for each x-relation R is stored in a separate table *lin:R*(`aid1`, `table`, `aid2`), whose tuples denote that R's alternative `aid1` contains alternative `aid2` from table `table` in its lineage. Here is an example of an encoded ULDB corresponding to a subset of the data from Section 2.4, with confidence values added to `Saw` for illustrative purposes only:

Saw

aid	xid	conf	witness	car
11	51	0.8	Cathy	Honda
12	51	0.2	Cathy	Mazda

Owns

aid	xid	conf	owner	car
21	61	-1	Jimmy	Toyota
22	61	-1	Jimmy	Mazda
23	62	-2	Billy	Honda

Suspects

aid	xid	conf	person
31	71	-1	Jimmy
32	72	-2	Billy

lin:Suspects

aid1	table	aid2
31	Saw	12
31	Owns	22
32	Saw	11
32	Owns	23

The Trio implementation does not require all relations to have uncertainty or lineage—conventional relations can coexist with x-relations, and they can be queried together. Currently, a single Trio metadata catalog keeps track of which relations include uncertainty, and records the general structure of the lineage present in the database, i.e., which x-relations include lineage to which other x-relations.

4.2 Query Processing

The Trio system evaluates TriQL queries by translating them into SQL operations over the representation described in the previous section. Consider a TriQL query Q whose result is to be stored in a new x-relation R. Q is rewritten into a query Q_r executed against the actual database. Q_r's result is post-processed in order to group alternatives into the x-tuples represented in R, and to generate the lineage table *lin:R*. Confidence values for result tuples are computed separately, possibly not until they are requested by the user or application. We detail each of these phases next, then describe how Trio produces *transient* results for queries without an `INTO` clause.

Query Rewriting. Consider one last time a variant on our favorite join query producing `Suspects`:

```
SELECT Owns.owner as person INTO Suspects
FROM Saw, Owns
WHERE Saw.car = Owns.car AND conf(Saw) > 0.5
```

This query is translated to the following SQL query Q_r, shown with a sample query result:

```
CREATE TABLE Suspects AS
   SELECT newOid() as aid, NULL as xid, NULL as conf,
          owner as person,
          Saw.aid as Saw_aid, Saw.xid as Saw_xid,
          Owns.aid as Owns_aid, Owns.xid as Owns_xid
    FROM Saw, Owns WHERE Saw.car = Owns.car AND conf(Saw) > 0.5
```

aid	xid	conf	person	Saw_aid	Saw_xid	Owns_aid	Owns_xid
32	NULL	NULL	Billy	11	51	23	62

The translation of queries with `lineage()` predicates requires joining in the corresponding *lin* tables in the expected way; details are omitted.

Post-processing. Next, the Trio engine processes the result table obtained from the translated query Q_r as follows. Each step is performed by simple, automatically-generated SQL commands.

1. New xid's are generated for the `xid` column, to group the alternatives in the result into x-tuples. Recalling the operational semantics from Section 3.1, we can use a simple `GROUP BY` query on the xid's of the original tuples (`Saw_xid` and `Owns_xid` in our example) and a sequence to generate the new xid's.
2. The lineage information—`aid` values with table names—is moved to a separate newly-created table (`lin:Suspects` in our example).
3. The data needed for steps 1 and 2 (attributes `Saw_aid`, `Saw_xid`, `Owns_aid`, and `Owns_xid` in our example) are removed from the query result table by dropping the corresponding columns.
4. Metadata information is added to the Trio catalog about the new table `Suspects`, and the fact that it was derived from `Saw` and `Owns`.

Computing Confidences. Lastly, we discuss confidences on query results: their definition and their computation. For the class of queries we have considered so far, a very simple definition is enabled by lineage: The confidence of a tuple alternative t in a query result is the product of the confidences of the tuple alternatives in $\lambda(t)$. For example, if `Hank` in table `Suspects` was derived from a 0.8 confidence `Saw` alternative together with a 0.5 confidence `Owns` tuple, then `Hank` has confidence 0.4. This definition is consistent with the standard interpretation of probabilistic databases.

The fact that confidences can be computed via lineage enables us to compute and store confidence values *lazily* if we wish to do so—if an application typically does not use confidence values, or uses them very selectively. Of course we can also compute confidences *eagerly* during query execution; both options were presented in our operational semantics (Section 3.1). Many optimizations are possible for computing confidences, and new challenges arise when we extend the class of queries considered. This area constitutes an important avenue of current work, as mentioned in the next section.

Transient Query Results. So far in this section we have assumed query results are to be stored in a newly-created table. Trio also supports conventional (transient) query results, when the `INTO` clause is omitted: Translated query Q_r now sorts its result by the xid's of the original x-tuples, and is executed via a cursor instead of writing its result into a table. Trio returns a cursor for Q to its client. As the client iterates through the Q cursor, x-tuple objects (with lineage) are generated on-the-fly by iterating through the cursor for the sorted Q_r result.

5 Current and Future Directions

- **Theory:** There is a great deal of theoretical work to do on ULDBs. Effectively, nearly every topic considered in relational database theory can be reconsidered in ULDBs. Examples include (but are certainly not limited to) dependency theory and database design, query containment and rewritings, and sampling and statistics. We are attempting to address the most interesting and relevant of these problems.

- **Confidences:** We are currently exploring various algorithms and optimizations when computing confidences lazily, such as memoization and pruning the lineage traversal. We are also studying the fundamental tradeoffs between lazy and eager computation of confidences.

- **Updates:** The formal query semantics and TriQL language presented here are query-only. We are in the process of considering updates: identifying and formally defining an appropriate set of primitive update operations for ULDBs, and exposing them in SQL-like data modification commands.

- **Queries:** Our current Trio prototype supports only single-block "SPJ" queries. While some additional query constructs simply require additional programming, other constructs—such as duplicate elimination, aggregation, and negated subqueries—require more complex definitions and implementation of lineage. We also plan to add richer constructs for querying uncertainty, including "top-K" style queries based on confidence, and "horizontal" queries that aggregate or perform subqueries across tuple alternatives.

- **Storage, Auxiliary Structures, and Query Optimization:** The ULDB model introduces new options and tradeoffs in data layout, along with new possibilities for indexing, partitioning, and materialized views, and new challenges for query operators and optimizations. To fully explore many of these topics we will need to move our prototype away from its translation-based approach and begin custom implementation inside a DBMS.

- **Long-Term Goals:** There are several important features in the original Trio proposal [Wid05] that we have barely touched upon but remain on our agenda: Continuous uncertain values such as intervals and

Gaussians, incomplete relations, versioning, and various types of external lineage. In addition, the introduction of uncertainty and lineage introduces new challenges in both human and programmatic interfaces.

- **Applications:** Needless to say, a most important long-term goal is to deploy a wide variety of applications on the Trio system to test its functionality and scalability. We also may develop specialized versions of Trio with restricted and/or extended functionality suited to specific applications, such as for sensor data management or data integration.

Acknowledgments

We thank Parag Agrawal, Alon Halevy, Shubha Nabar, and the entire Trio group for many useful discussions.

References

[AKG91] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. In *Theoretical Computer Science*, pages 34–48, 1991.

[BCTV04] D. Bhagwat, L. Chiticariu, W.C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *Proc. of Intl. Conference on Very Large Data Bases (VLDB)*, 2004.

[BDHW05] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. The symbiosis of lineage and uncertainty. Technical report, Stanford InfoLab, December 2005. Available at http://dbpubs.stanford.edu/pub/2005-39.

[BDM+05] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *Proc. of ACM SIGMOD Intl. Conference on Management of Data*, 2005.

[BGMP92] D. Barbará, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, October 1992.

[BKT00] P. Buneman, S. Khanna, and W. Tan. Data provenance: Some basic issues. In *Proc. of Intl. Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 87–93, 2000.

[BKT01] P. Buneman, S. Khanna, and W.C. Tan. Why and where: A characterization of data provenance. In *Proc. of Intl. Conference on Database Theory (ICDT)*, 2001.

[BKT02] P. Buneman, S. Khanna, and W. Tan. On propagation of deletions and annotations through views. In *Proc. of ACM Intl. Conference on Principles of Database Systems (PODS)*, 2002.

[BP82] B. Buckles and F. Petry. A fuzzy model for relational databases. *International Journal of Fuzzy Sets and Systems*, 7:213–226, 1982.

[BP93] R.S. Barga and C. Pu. Accessing imprecise data: An approach based on intervals. *IEEE Data Engineering Bulletin*, 16(2):12–15, June 1993.

[CP87] R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In *Proc. of Intl. Conference on Very Large Data Bases (VLDB)*, 1987.

[CTV05] L. Chiticariu, W. Tan, and G. Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. *Proc. of ACM SIGMOD Intl. Conference on Management of Data*, 2005.

[CW00] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 2000.

[CW03] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB Journal*, 12(1):41–58, May 2003.

[CWW00] Y. Cui, J. Widom, and J. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, June 2000.

[DBHW06] A. Das Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, April 2006.

[DMS05] N. Dalvi, G. Miklau, and D. Suciu. Asymptotic conditional probabilities for conjunctive queries. In *Proc. of Intl. Conference on Database Theory (ICDT)*, 2005.

[DNW05] A. Das Sarma, S. U. Nabar, and J. Widom. Representing uncertain data: Uniqueness, equivalence, minimization, and approximation. Technical report, Stanford InfoLab, 2005. Available at http://dbpubs.stanford.edu/pub/2005-38.

[DS04] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proc. of Intl. Conference on Very Large Data Bases (VLDB)*, 2004.

[DS05] N. Dalvi and D. Suciu. Answering queries from statistics and probabilistic views. In *Proc. of Intl. Conference on Very Large Data Bases (VLDB)*, 2005.

[FR97] N. Fuhr and T. Rölleke. A probabilistic NF2 relational algebra for imprecision in databases. *Unpublished Manuscript*, 1997.

[Gra84] G. Grahne. Dependency satisfaction in databases with incomplete information. In *Proc. of Intl. Conference on Very Large Data Bases (VLDB)*, 1984.

[Gra89] G. Grahne. Horn tables - an efficient tool for handling incomplete information in databases. In *Proc. of ACM Intl. Conference on Principles of Database Systems (PODS)*, 1989.

[IL84] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, October 1984.

[LLRS97] L.V.S. Lakshmanan, N. Leone, R. Ross, and V.S. Subrahmanian. ProbView: A flexible probabilistic database system. *ACM Transactions on Database Systems*, 22(3):419–469, September 1997.

[Var85] M. Vardi. Querying logical databases. In *Proc. of ACM Intl. Conference on Principles of Database Systems (PODS)*, 1985.

[Wid05] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. of Conference on Innovative Data Systems Research (CIDR)*, 2005.

[WS97] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 1997.