Trio-One: Layering Uncertainty and Lineage on a Conventional DBMS[*]

Michi Mutsuzaki, Martin Theobald, Ander de Keijzer[†], Jennifer Widom,
Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Raghotham Murthy, Tomoe Sugihara[‡]

Stanford University InfoLab
`http://infolab.stanford.edu/trio`

ABSTRACT

Trio is a new kind of database system that supports *data*, *uncertainty*, and *lineage* in a fully integrated manner. The first Trio prototype, dubbed *Trio-One*, is built on top of a conventional DBMS using data and query translation techniques together with a small number of stored procedures. This paper describes Trio-One's translation scheme and system architecture, showing how it efficiently and easily supports the Trio data model and query language.

1. INTRODUCTION

In the *Trio* project at Stanford, we are developing a new kind of database management system—one that handles *data*, *uncertainty* of the data, and data *lineage* together in a fully integrated manner [3, 6]. Some of the application domains targeted by Trio are data cleaning and integration, information extraction, and scientific data management [6].

Our first system prototype, dubbed *Trio-One*, is primarily layered on top of a conventional relational DBMS. From the user and application standpoint Trio-One appears to be a "native" implementation of the Trio data model, query language, and other features. However, Trio-One encodes the uncertainty and lineage present in Trio's data model in conventional relational tables, and it uses a rewrite-based approach for most data management and query processing. A small number of stored procedures are used for specific functionality and increased efficiency.

This paper, accompanying our system demonstration, captures the Trio system as of late 2006. A previous overview paper [3] captured an earlier snapshot of the project, and a previous system demonstration [1] included a subset of Trio's query language and a limited set of additional features. Motivation and technical justification for a data model and system that includes both uncertainty and lineage can be found in [2, 6]. Trio's query language is specified in detail in [5].

The remainder of this paper proceeds as follows:

- Section 2 introduces the overall Trio-One system architecture and briefly describes its application and user interfaces.
- Section 3 reviews Trio's *ULDB* data model (for *Uncertainty-Lineage Databases*), introduces a very small running example database, and shows how ULDB databases are encoded in conventional tables.
- Section 4 describes Trio's query language, *TriQL*, and shows how TriQL queries over ULDBs are translated automatically to SQL queries over the encoded tables.
- Section 5 describes some Trio-specific features—lineage tracing, on-demand confidence computation, coexistence checks, and extraneous data removal—covering their functionality and implementation.

We conclude in Section 6 with future directions for Trio, including the possibility of a *Trio-Two* system that would take a built-in rather than layered approach. Due to space constraints this paper does not include discussion of related work; we refer the reader to [2, 3, 6].

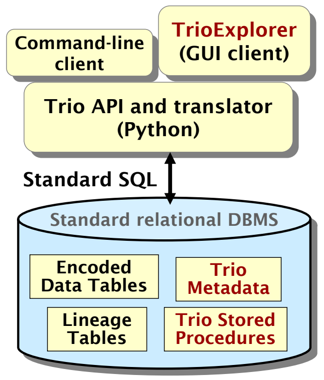

Figure 1: System Architecture.

2. THE TRIO-ONE SYSTEM

Figure 1 shows the basic three-layer Trio-One architecture. The core system is implemented in Python and mediates between the underlying relational DBMS (currently the *PostgreSQL* open-source DBMS) and Trio interfaces and applications. The Python layer presents a simple Trio API that extends the standard Python DB 2.0 API for database access (Python's analog of JDBC). The Trio API accepts TriQL queries in addition to regular SQL, and query results may be *x-tuples* in the ULDB model (see Section 3) as well as regular tuples. The API also exposes *lineage tracing*, along with the other Trio-specific features covered in Section 5. Using the Trio API, we built a generic command-line interactive client similar to that provided by most DBMS's, and a full-featured graphical user interface called *TrioExplorer*.

Trio DDL commands are translated via Python to SQL DDL commands based on the encoding described in Section 3. The translation is fairly straightforward, as is the corresponding translation of `insert` statements and bulk load.

[*]This work was supported by the National Science Foundation under grants IIS-0324431 and IIS-0414762, and by grants from the Boeing and Hewlett-Packard Corporations.
[†]Visiting from University of Twente, supported by Netherlands Organization for Scientific Research (NWO)
[‡]Visiting from NEC Corporation

3rd *Biennial Conference on Innovative Data Systems Research (CIDR)* January 7-10, 2007, Asilomar, California, USA.

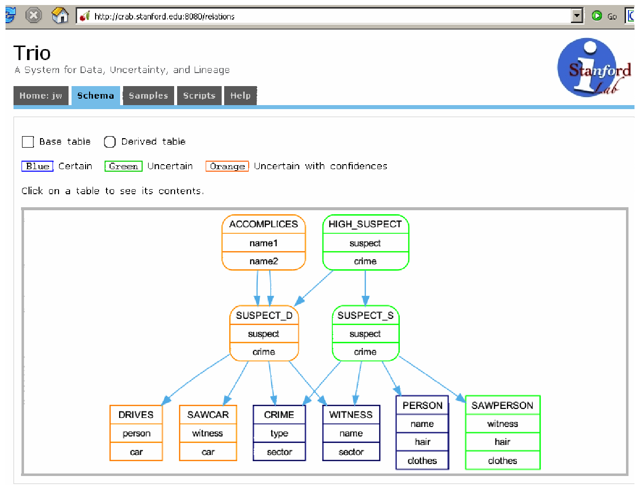

Figure 2: TrioExplorer Screenshot.

TriQL query processing proceeds in two phases. In the *translation* phase, a TriQL parse tree is created and progressively transformed into a tree representing one or more standard SQL statements, based on the data encoding scheme. In the *execution* phase, the SQL statements are executed against the relational database encoding. Depending on the original TriQL query, Trio stored procedures may be invoked and some post-processing may occur. For efficiency, most additional runtime processing is written in C and executes in the DBMS server via the Postgres *SPI* interface.

TriQL query results can either be *stored* or *transient*. Stored query results are placed in a new persistent table, and lineage relationships from the query's result data to data in the query's input tables also is stored persistently. Transient query results are accessed through the Trio API in a typical cursor-oriented fashion, with an additional method that can be invoked to explore the lineage of each returned tuple. For transient queries, query result processing and lineage creation occurs in response to cursor *fetch* calls, and neither the result data nor its lineage are persistent.

TrioExplorer offers a rich interface for interacting with the Trio system. It implements a Python-generated, multi-threaded web server using *CherryPy*, and it supports multiple users logged into private and/or shared databases. It accepts Trio DDL and DML commands and provides numerous features for browsing and exploring schema, data, uncertainty, and lineage. It also enables on-demand confidence computation, coexistence checks, and extraneous data removal. Finally, it supports loading of scripts, command recall, and other user conveniences. Figure 2 shows a snapshot of TrioExplorer's schema visualizer including schema-level lineage relationships among tables.

3. TRIO DATA

We briefly review *ULDB*s (*Uncertainty-Lineage Databases*), the data model forming the basis of the Trio system. More details and examples can be found in [2, 3]. ULDBs extend the standard SQL relational model with four new constructs:

1. tuple *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 "?"
4. *lineage*, connecting tuple alternatives to other alternatives from which they were derived

A formal semantics for ULDBs based on *possible instances* is specified in [2], which also shows that the ULDB model is *complete*: any finite set of possible instances can be represented as a ULDB.

The following example illustrates a ULDB for a highly simplified "crime-solver" application.[1] Tables `Saw(witness,car)` and `Drives(person,car)` capture (possibly uncertain) driver information and crime vehicle sightings, respectively. Table `Suspects(person)` is derived by joining `Saw` and `Drives`, so it contains (possible) drivers of (possibly) sighted cars. Confidence values are optional in ULDB tables—for now imagine that they are not present in tables `Saw` and `Drives`. Confidences will be discussed in Section 3.1.

ID	**Saw (witness, car)**
51	`(Cathy,Honda):0.6 ‖ (Cathy,Mazda):0.4`

ID	**Drives (person, car)**	
61	`(Jimmy,Mazda):0.3 ‖ (Freddy,Mazda):0.7`	
62	`(Billy,Honda):0.8`	**?**
63	`(Hank,Honda):1.0`	

ID	**Suspects (person)**		
71	`Jimmy ‖ Freddy`	**?**	$\lambda(71,1) = (51, 2) \wedge (61, 1)$ $\lambda(71,2) = (51, 2) \wedge (61, 2)$
72	`Billy`	**?**	$\lambda(72,1) = (51, 1) \wedge (62, 1)$
73	`Hank`	**?**	$\lambda(73,1) = (51, 1) \wedge (63, 1)$

Tuples 51, 61, and 71 have two *alternatives* which are mutually exclusive in our model. Tuple 62 and all three `Suspects` tuples have *maybe* ("?") annotations. The Boolean λ functions on the derived `Suspects` table represent the lineage of individual alternatives. For example, Jimmy or Freddy may be a suspect, because one of them drives a Mazda and Cathy may have seen a Mazda (alternatives (71,1) and (71,2) with lineage $(51, 2) \wedge (61, 1)$ and $(51, 2) \wedge (61, 2)$ respectively). If Cathy saw a Honda, then neither Jimmy nor Freddy is a suspect, captured by the "?" annotation on tuple 71. Hank definitely drives a Honda, so he is a suspect if Cathy saw a Honda (tuple 73 with lineage $(51, 1) \wedge (63, 1)$), and Billy may also be a suspect (tuple 72 with lineage $(51, 1) \wedge (62, 1)$).

3.1 Confidence Values

Confidence values may optionally be attached to alternatives, as illustrated in tables `Saw` and `Drives` above. By default, ULDBs use a *probabilistic* interpretation of confidence values, although alternate user-specified interpretations are allowed (briefly discussed in Section 4). Using the sample confidence values on the base data, in the `Suspects` join result above, `Jimmy`, `Freddy`, `Billy`, and `Hank` have confidence values 0.12, 0.28, 0.48, and 0.6 respectively.

Particularly noteworthy is the fact that confidence values on join results (and other query results) can be computed based on lineage. For example, the lineage of tuple 72 (`Billy`) is the formula $(51, 1) \wedge (62, 1)$. Treating (51,1) and (62,1) as symbols with probabilities 0.6 and 0.8 (the confidence values associated with `(Cathy,Honda)` and `(Billy,Honda)`), the probability of the formula, and therefore the confidence of suspect `Billy`, is 0.48.

3.2 Disjunctive Lineage

Suppose we add a fourth tuple to table `Drives`:

64	`(Frank,Honda):0.7 ‖ (Frank,Mazda):0.3`

yielding a new `Suspects` tuple:

[1]Apologies for incessant use of the same toy application, but it's by far the best vehicle we have found for compact, easily-understood examples of Trio features. The actual system demonstration uses a more complex movie-recommendation application.

74	Frank ‖ Frank	?	λ(74,1) = $(51,1) \wedge (64,1)$
			λ(74,2) = $(51,2) \wedge (64,2)$

In the TriQL query language (Section 4), the default is for "horizontal duplicates" to be merged in query results, producing disjunctive lineage. Thus, tuple 74 would actually contain a single `Frank` alternative (still with a "?"), and its lineage would be:

$$((51,1) \wedge (64,1)) \vee ((51,2) \wedge (64,2))$$

We can still compute the confidence of `Frank` based on lineage, i.e., as the probability of the above formula. Notice that here the variables in the lineage formula are not independent: (51,1) and (51,2) are mutually exclusive, and so are (64,1) and (64,2). Taking mutual exclusion into account, the probability of the above formula is $(0.6 \cdot 0.7) + (0.4 \cdot 0.3) = 0.54$. In other words, the probability of Cathy seeing the same car Frank drives, and therefore Frank being a suspect, is 0.54.

3.3 Encoding ULDB Data

We now describe how ULDB databases are encoded in regular relational tables. Hereafter we use *x-tuple* to refer to a tuple in the ULDB model, i.e., a tuple that may include alternatives, "?", and confidence values, and *tuple* to denote a regular relational tuple.

Let $T(A_1, \ldots, A_n)$ be a ULDB table that may include both confidences and lineage. We store the data portion of T as a conventional table (which we will also refer to as T) with four additional attributes: T(`aid`, `xid`, `conf`, `num`, $A_1, \ldots, A_n$). Each alternative in the original ULDB table is stored as its own tuple in T, and the additional attributes function as follows:

- `aid` is a unique alternative identifier.
- `xid` identifies the x-tuple that this alternative belongs to.
- `conf` stores the confidence of the alternative, or `NULL` if there are no confidence values or if this confidence value has not yet been computed. (Each table either permits confidence values on all alternatives or on none of them; this *table type* is part of the schema information.)
- `num` efficiently tracks whether the alternative's x-tuple has a "?". (Some details are given in Section 4.1.)

The system always creates indexes on `aid` and `xid`. In addition, Trio users may create indexes on any of the original data attributes $A_1, \ldots, A_n$ using standard `CREATE INDEX` commands that are simply passed through Trio to the underlying DBMS.

The lineage information for each table T is stored in a separate table lin_T(`aid`, `src_aid`, `src_table`), indexed on `aid` and `src_aid`. A tuple (a_1, a_2, T_2) in lin_T denotes that T's alternative a_1 has alternative a_2 from table T_2 in its lineage. Additional flags (details omitted) encode whether multiple lineage relationships for alternatives are conjunctive or disjunctive.

4. TRIO QUERIES

TriQL [3, 5], Trio's query language for ULDBs, is an extension of SQL. TriQL queries return uncertain relations in the ULDB model, with lineage that connects query result data to the queried data. As mentioned in Section 2, a TriQL query result may be *transient*, offering a cursor interface and a special method for retrieving lineage, or the query result and its lineage may be stored in persistent tables according to the encoding scheme described in Section 3.3. As a first example, the join query from Section 3 with its result stored in table `Suspects` would be written in TriQL simply as:

```
TriQL>  CREATE TABLE Suspects AS
TriQL>    SELECT person
TriQL>    FROM Saw, Drives
TriQL>    WHERE Saw.car = Drives.car
```

In addition to modifying SQL semantics for ULDBs, TriQL adds a number of new constructs for querying and manipulating both uncertainty and lineage. A comprehensive specification for TriQL's query and update language appears in [5]. In the remainder of this section we use examples to illustrate TriQL semantics and functionality, and how TriQL queries are rewritten automatically into standard SQL over the relationally-encoded ULDB data.

4.1 Basic Rewriting Scheme

Consider the `Suspects` query shown above, first in its transient form (i.e., without `CREATE TABLE`). The Trio Python layer translates the TriQL query into the following SQL query, sends it to the underlying DBMS, and opens a cursor on the result:

```
SQL>  SELECT Drives.person,
SQL>         Saw.aid, Drives.aid,
SQL>         Saw.xid, Drives.xid,
SQL>         (Saw.num * Drives.num) AS num
SQL>  FROM Saw, Drives
SQL>  WHERE Saw.car = Drives.car
SQL>  ORDER BY Saw.xid, Drives.xid
```

Let *Tfetch* denote a cursor call to the Trio API for the original TriQL query, and let *Sfetch* denote a cursor call to the underlying DBMS for the translated SQL query. Each call to *Tfetch* must return a complete x-tuple, which may entail several calls to *Sfetch*: Each tuple returned from *Sfetch* on the SQL query corresponds to one alternative in the TriQL query result, and the set of alternatives with the same returned `Saw.xid` and `Drives.xid` pair comprise a single result x-tuple. (The TriQL operational join semantics presented in [3] makes this property very clear.) Thus, on *Tfetch*, Trio collects all SQL result tuples for a single `Saw.xid`/`Drives.xid` pair (enabled by the `ORDER BY` clause in the SQL query), generates a new `xid` and new `aid`'s, and constructs and returns the result x-tuple.

Note that the underlying SQL query also returns the `aid`'s from `Saw` and `Drives`. These values (together with the table names) comprise the lineage for the alternatives in the result x-tuple. As mentioned earlier, the `num` field is used to encode the presence or absence of "?".[2] Finally, since result confidence values for joins are not computed until they are explicitly requested (see Section 5), *Tfetch* initially returns `NULL` confidence for all alternatives, whether or not the query result logically contains confidence values.

For the stored (`CREATE TABLE`) version of the query, Trio first issues DDL commands to create new tables for the query result and its lineage. Trio then executes the same SQL query shown above, except instead of constructing and returning x-tuples one at a time, the system directly inserts the new alternatives and their lineage into the result and lineage tables, already in their encoded form. All processing occurs within an SPI stored procedure on the database server, thus avoiding unnecessary roundtrips between the Python module and the underlying DBMS.

4.2 Duplicate Elimination

Like the "horizontal" merging of duplicate alternatives shown in Section 3.2, TriQL queries can perform more conventional "vertical" duplicate elimination, which also results in disjunctive lineage:

```
TriQL>  SELECT DISTINCT car
TriQL>  FROM Drives
```

Considering the version of `Drives` without confidences, we get:

[2]Our scheme essentially maintains the invariant that an alternative's x-tuple has a "?" if and only if its `num` field exceeds the x-tuple's number of alternatives.

ID	car
81	Mazda
82	Honda

λ(81,1) = $(61,1) \vee (61,2)$
λ(82,1) = $(62,1) \vee (63,1)$

In general, horizontal and/or vertical duplicate elimination occurs as the final step in a query that may also include filtering, joins, and other operations. Two related issues must be addressed: (1) how the resulting disjunctive lineage is encoded, and (2) how the TriQL queries are translated. In the currently supported version of TriQL, all lineage generated by a query prior to duplicate elimination is conjunctive. Thus, after duplicate elimination, the lineage of each result alternative is a formula in disjunctive normal form:

$$(a_1 \wedge \cdots \wedge a_i) \vee (b_1 \wedge \cdots \wedge b_j) \vee \cdots \vee (c_1 \wedge \cdots \wedge c_k)$$

Trio encodes these DNF formulas by introducing dummy identifiers for each disjunct and storing flags to indicate whether a set of lineage relationships is conjunctive or disjunctive. (Further details are omitted due to space constraints.)

Merging "horizontal" duplicates and creating the corresponding disjunctive lineage can occur entirely within the *Tfetch* method (recall the basic rewriting scheme in Section 4.1): All alternatives for each result x-tuple, together with their lineage, already need to be collected within *Tfetch* before the x-tuple is returned. Thus, *Tfetch* can merge all duplicate alternatives and create the disjunctive lineage for them, then return the modified x-tuple. If the query includes `UNMERGED`, indicating that horizontal duplicate-elimination should not occur, the extra steps are simply skipped.

`SELECT DISTINCT` is more complicated, requiring two phases. First, a translated SQL query is produced as if `DISTINCT` were not present, except the result is ordered by the data attributes instead of `xid`'s. One scan through this SQL result is required to merge duplicates and create disjunctive lineage. This intermediate result must then be reordered by `xid`'s, in order to construct the correct x-tuples in the final result. For our very simple example above, the following two SQL queries are generated. `Temp` holds the temporary result after the first query is used to eliminate duplicates and create disjunctive lineage.

```
SQL>  SELECT person, car, aid, xid, num
SQL>  FROM Drives
SQL>  ORDER BY person, car

SQL>  SELECT person, car, aid, xid, num
SQL>  FROM Temp
SQL>  ORDER BY xid
```

4.3 Aggregation

TriQL supports standard SQL grouping and aggregation. Consider the following query:

```
TriQL>  SELECT car, count(*)
TriQL>  FROM Drives GROUP BY car
```

The query result appears fairly straightforward for our very simple example, although notice that tuple 91 is the result of merging two duplicate alternatives.

ID	(car, count)
91	(Mazda,1)
92	(Honda,1) ‖ (Honda,2)

λ(91,1) = $(61,1) \vee (61,2)$
λ(92,1) = (63,1)
λ(92,2) = $(62,1) \wedge (63,1)$

In general, aggregation can be an exponential operation in ULDBs (and in other data models for uncertainty). Thus, TriQL includes built-in *approximate* aggregation functions, including *low* and *high* bounds for the aggregate result, and *expected* values that take confidence into account. For example, the following query returns expected values for the number of occurrences of each type of car in `Drives`.

```
TriQL>  SELECT car, ecount(*)
TriQL>  FROM Drives GROUP BY car
```

The result on our example `Drives` table with confidence values is `(Mazda,1.0)`, `(Honda,1.8)`.

TriQL supports 20 different aggregation functions: four versions (*full*, *low*, *high*, and *expected*) for each of the five standard functions (*count*, *min*, *max*, *sum*, *avg*). (*Distinct* versions of the aggregation functions currently are not supported.) All of the *full* functions and some of the approximations unfortunately cannot be translated to SQL queries over the encoded data, and thus are implemented as algorithmic stored procedures. Furthermore, several of the *low*/*high* bounds and one of the *expected* values are themselves approximations to the tightest bound or value, because finding the exact answer based on possible-instances can be extremely expensive. (We expect the approximations to do well in practice, but details are far beyond the scope of this description.) Many of the approximate functions can be implemented exactly and translated very easily. For example, for the `ecount` TriQL example above, the SQL query over the encoded data is simply:

```
SQL>  SELECT car, sum(conf)
SQL>  FROM Drives GROUP BY car
```

Note that with approximate aggregation, query results are regular tables and not ULDB tables: they do not include alternatives, "?", confidence values, or lineage.

4.4 Reorganizing Alternatives

TriQL has two constructs for reorganizing the alternatives in a query result:

- *Flatten* turns each alternative of a table into its own tuple.
- *GroupAlts* regroups alternatives into new x-tuples based on a set of attributes.

As simple examples, and omitting lineage (which in both cases is a straightforward one-to-one mapping from result alternatives to source alternatives), "`SELECT FLATTEN * FROM Saw`" gives us:

(witness, car)
(Cathy,Honda)
(Cathy,Mazda)

and "`SELECT GROUPALTS`(car) `* FROM Drives`" gives us:

(person, car)
(Jimmy,Mazda) ‖ (Freddy,Mazda)
(Billy,Honda) ‖ (Hank,Honda)

The translation scheme for queries with *Flatten* is a simple modification to the basic scheme in which each result alternative is assigned its own `xid`. *GroupAlts* is also a straightforward modification: Instead of the translated SQL query grouping by `xid`'s from the input tables to create result x-tuples, it groups by the attributes specified in `GROUPALTS`.

4.5 Horizontal Subqueries

"Horizontal" subqueries in TriQL enable querying across the alternatives that comprise individual x-tuples. As a (meaningless contrived) example, we can select from table `Saw` all vehicles sighted that are not Mazdas, but a Mazda sighting appears as another alternative of the same x-tuple:

```
TriQL>  SELECT car
TriQL>  FROM Saw
TriQL>  WHERE car <> 'Mazda'
TriQL>  AND EXISTS [car = 'Mazda']
```

On our example data, this query would return just the first alternative, Honda, of tuple 51.

In general, enclosing a subquery in [] instead of () causes the subquery to be evaluated over the "current" x-tuple, treating its alternatives as if they are a table. Syntactic shortcuts are provided for common cases, such as simple filtering predicates as in the example above. Full details of horizontal subqueries and numerous examples can be found in [5].

Horizontal subqueries are very powerful, but surprisingly easy to implement based on our data encoding. First, syntactic shortcuts are expanded. In our example above, [car = 'Mazda'] is a shortcut for [SELECT * FROM Saw WHERE car='Mazda']. Here, Saw within the horizontal subquery refers to the Saw alternatives in the current x-tuple being evaluated [5].) Second, the horizontal subquery is replaced with a standard SQL subquery that adds aliases for inner tables and a condition correlating xid's with the outer query:

```
SQL> ... AND EXISTS (SELECT * FROM Saw S
SQL>                 WHERE car = 'Mazda'
SQL>                 AND S.xid = Saw.xid)
```

S.xid=Saw.xid restricts the horizontal subquery to operate on the data in the current x-tuple. Translation for the general case involves a fair amount of context and bookkeeping to ensure proper aliasing and ambiguity checks, but all horizontal subqueries, regardless of their complexity, have a direct translation to regular SQL subqueries with additional xid equality conditions.

4.6 Built-In Predicates and Functions

TriQL currently includes three built-in predicates and functions: Conf(), Maybe(), and Lineage(). Function Conf() can be used to filter query results based on the confidence of the input data (e.g., Conf(Saw)) and the confidence of the result (Conf(*)). For example, if we want to compute suspects only considering sightings with confidence > 0.5 and only retaining results whose confidence would be > 0.4, we add the following conjuncts to our original join query:

```
TriQL>  AND Conf(Saw) > 0.5 AND Conf(*) > 0.4
```

Built-in predicate Maybe() takes no arguments and is true if and only if the current x-tuple has a "?".

Built-in predicate Lineage() allows lineage to be traced as part of a TriQL query. For example, we can ask for all witnesses contributing to Hank being a suspect:

```
TriQL>  SELECT Saw.witness
TriQL>  FROM Suspects, Saw
TriQL>  WHERE Lineage(Suspects,Saw)
TriQL>  AND Suspects.person = 'Hank'
```

Lineage(X,Y) (which can also be written as "X==>Y") is true whenever Y is reachable from X by one or more lineage steps. That is, it considers the transitive closure of the lineage function λ.

Function Conf() is implemented as an SPI stored procedure. If it has just one argument T, the procedure first examines the current T.conf field to see if a value is present. If so, that value is returned. If the T.conf is NULL, on-demand confidence computation is invoked (see Section 5.2), and the resulting confidence value is stored permanently and returned. Conf(*) always activates confidence computation, and includes the resulting confidence value in the query result (instead of NULL) as well as returning it from the function. An "intermediate" version of Conf() can also be called, with multiple table arguments but not the full "*"; details are omitted due to space constraints [5].

The Maybe() and Lineage() predicates are incorporated into the query translation phase (recall Section 2). Predicate Maybe() is straightforward: It translates to a simple comparion between the num attribute and the number of alternatives in the current x-tuple. (One subtlety is that Maybe() returns true even when a tuple's question mark is "extraneous"—that is, the tuple in fact always has an alternative present, due to its lineage. See Section 5.4 for a brief discussion.)

Predicate Lineage(X,Y) is translated into one or more SQL subqueries that check if the lineage relationship holds: Schema-level lineage information is used to determine the possible table-level "paths" from X to Y. Each path produces a subquery that joins lineage tables along that path, with X and Y at the endpoints. Suppose for the sake of illustration that a table Saw2 was derived from Saw, and then Suspects was derived from Saw2. Then Lineage(Suspects,Saw) would be translated as follows, recalling the lineage encoding described in Section 3.

```
SQL>  EXISTS (SELECT *
SQL>    FROM lin_Suspects L1, lin_Saw2 L2
SQL>    WHERE Suspects.aid = L1.aid
SQL>    AND L1.src_table = 'Saw2'
SQL>    AND L1.src_aid = L2.aid
SQL>    AND L2.src_table = 'Saw'
SQL>    AND L2.src_aid = Saw.aid )
```

4.7 Query-Defined Result Confidences

By default, confidence values on query results respect a probabilistic interpretation, and they are computed by the system on-demand. (A "COMPUTE CONFIDENCES" clause can be added to a query force confidence compuation as part of query execution.) Algorithms for confidence computation are discussed in Section 5.2.

A query can override the default result confidence values by assigning values in its SELECT clause to the reserved attribute name conf. Suppose in our Suspects join query we prefer result confidences to be the lesser of the two input confidences, instead of their (probabilistic) product. Assuming a built-in function lesser, we write:

```
TriQL>  SELECT person,
TriQL>          lesser(Conf(S),Conf(D)) AS conf
TriQL>  FROM Saw S, Drives D
TriQL>  WHERE S.car = D.car
```

Referring back to Section 3.1 to see the difference, now Jimmy, Freddy, Billy, and Hank in the join result have confidence values 0.3, 0.4, 0.6, and 0.6 respectively.

Recall from Section 3.3 that our data encoding scheme adds a column conf to each underlying table to store confidence values. Consequently, "AS conf" clauses simply pass through the query translation phase unmodified.

5. ADDITIONAL TRIO FEATURES

TriQL queries and updates are the typical way of interacting with Trio data, just as SQL is used in a standard relational DBMS. However, uncertainty and lineage in ULDBs introduce several interesting features beyond just query execution.

5.1 Lineage

As TriQL queries are executed and their results are stored, and additional queries are posed over previous results, complex lineage relationships can arise. As we have seen, data-level lineage is used for confidence computation and Lineage() predicates; it is also used for coexistence checks (Section 5.3) and extraneous data removal (Section 5.4). Trio also maintains a *schema-level* *lineage graph* that is used for `Lineage()` predicate translation (Section 4.6) and for some confidence-computation optimizations. This graph can also be a useful tool for the user; it is depicted by TrioExplorer in Figure 2.

TrioExplorer supports data-level lineage tracing through special buttons next to each displayed alternative. This feature is built on a method `ExplainLineage()` in the Trio API: For any alternative a, `ExplainLineage(`a`)` returns a representation of the boolean formula $\lambda(a)$, containing the alternatives in a's immediate lineage. Lineage can be traced further by calling `ExplainLineage()` on the alternatives from the first-level result. Another method, `BaseLineage(`a`)`, returns a's lineage formula traced and "unfolded" all the way to the base data—the result of a `BaseLineage()` call is comprised of alternatives that have no further lineage.

5.2 Confidence Computation

In the formalization of ULDBs [2], each *possible instance* has a probability based on the confidences of the data in that instance. In query results, lineage ties the possible result instances to the possible instances of the queried data. Thus, using lineage, each result alternative has a confidence value that captures the fraction of possible instances in which its lineage appears. This confidence value is correctly computed by constructing an alternative's lineage formula in terms of base data (i.e., the result of the `BaseLineage()` method described above) and then evaluating the probability of the formula using the confidence values on the base alternatives [2]. Some simple examples were given in Section 3.

Thus, when confidence computation is invoked for an alternative a, the system effectively invokes `BaseLineage(`a`)` and then evaluates the probability of the resulting formula using base-data confidences. We have developed several improvements to this naive approach:

- Whenever confidence values are computed, they are *memoized* for future use.
- It is not always necessary to traverse lineage all the way to the base data. If non-base alternatives in a's lineage are known to be independent, and their confidences have already been computed, there is no need to go further. Even when their confidences have not been computed, the lineage formula can be split, reducing overall complexity.
- We have developed algorithms for *batch* confidence computation that are implemented through SQL queries. These algorithms are appropriate and efficient when confidence values are desired for a significant portion of a result table.

5.3 Coexistence Checks

A user may wish to select a set of alternatives from one or more tables and ask whether those alternatives can all coexist. Two alternatives from the same x-tuple clearly cannot coexist, but the general case must take into account arbitrarily complex lineage relationships as well as tuple alternatives. For example, if we asked about alternatives (51,2) and (72,1) in our sample database, the system would tell us these alternatives cannot coexist. Coexistence checking can be performed by generating base-lineage formulas for the set of alternatives, augmenting them with formulas capturing mutual exclusion of tuple-alternatives, and then checking satisfiability.

5.4 Extraneous Data Removal

The natural execution of TriQL queries can generate *extraneous data*: a tuple alternative is extraneous if it can never be chosen (i.e., its lineage includes the conjunction of data that cannot coexist); a "?" annotation is extraneous if its tuple is always present. It is possible to check for extraneous alternatives and ?'s immediately after query execution (and, sometimes, as part of query execution). However, like confidence computation and coexisistence checks, extraneous data detection may require tracing lineage to the base data. Because we expect extraneous data and ?'s to be relatively uncommon, and users may not be concerned about their presence, we have chosen to implement extraneous data removal as a separate operation, roughly akin to garbage collection.

The astute reader may note that all of the features discussed in this section are interconnected. In fact they share code in the system, and they can share some of the optimizations discussed in Section 5.2 as well. For example, we can determine if an alternative is extraneous by computing its confidence and checking if it's $= 0$, while conversely a "?" is extraneous if the confidence values for its tuple sum to 1. Similarly, a set of alternatives can coexist iff, when treated as conjunctive lineage for a dummy alternative a, the confidence of a is > 0.

6. FUTURE DIRECTIONS

The Trio prototype is available online for anyone to experiment with; please visit the project home page (search "*stanford trio*") for a link. We are pursuing—or plan to pursue—a number of directions of future work, including:

- Efficiently processing queries that either provide a confidence threshold or ask for the top-k results by confidence.
- Extending the data model to include *continuous uncertainty* (e.g., intervals, Gaussians) and *incomplete relations*.
- Capturing *or-sets* [4] as a special-case of tuple alternatives that can be handled more efficiently.
- Completing the translation-based implementation of the full TriQL language [5], including regular ("vertical") subqueries, set operators, and data modification statements.
- Extending lineage to external relationships, and perhaps to track updates and support versioning.
- Considering a less layered and more "native" approach (*Trio-Two*), for which we would develop specialized storage methods, indexes, statistics, and query optimization techniques geared specifically to ULDB data and TriQL queries.

7. REFERENCES

[1] P. Agrawal, O. Benjelloun, A. Das Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *Proc. of VLDB*, pages 1151–1154, Seoul, Korea, September 2006. *Demonstration description*.

[2] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *Proc. of VLDB*, pages 953–964, Seoul, Korea, September 2006.

[3] O. Benjelloun, A. Das Sarma, C. Hayworth, and J. Widom. An introduction to ULDBs and the Trio system. *IEEE Data Engineering Bulletin, Special Issue on Probabilistic Databases*, 29(1):5–16, March 2006.

[4] A. Das Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *Proc. of ICDE*, Atlanta, Georgia, April 2006.

[5] TriQL: The Trio Query Language. Available from `http://infolab.stanford.edu/trio`.

[6] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. of CIDR*, Pacific Grove, California, 2005.