Chapter 1

TRIO: A SYSTEM FOR DATA, UNCERTAINTY, AND LINEAGE

Jennifer Widom
Dept. of Computer Science
Stanford University
widom@cs.stanford.edu

Abstract

This chapter covers the *Trio* database management system. Trio is a robust prototype that supports *uncertain data* and *data lineage*, along with the standard features of a relational DBMS. Trio's new *ULDB* data model is an extension to the relational model capturing various types of uncertainty along with data lineage, and its *TriQL* query language extends SQL with a new semantics for uncertain data and new constructs for querying uncertainty and lineage. Trio's data model and query language are implemented as a translation-based layer on top of a conventional relational DBMS, with some stored procedures for functionality and increased efficiency. Trio provides both an API and a full-featured graphical user interface.

Acknowledgments. Contributors to the Trio project over the years include (alphabetically) Parag Agrawal, Omar Benjelloun, Ashok Chandra, Julien Chaumond, Anish Das Sarma, Alon Halevy, Chris Hayworth, Ander de Keijzer, Raghotham Murthy, Michi Mutsuzaki, Tomoe Sugihara, Martin Theobald, and Jeffrey Ullman. Funding has been provided by the National Science Foundation and the Boeing and Hewlett-Packard Corporations.

Introduction

Trio is a new kind of database management system (DBMS): one in which *data*, *uncertainty* of the data, and data *lineage* are all first-class citizens. Combining data, uncertainty, and lineage yields a data management platform that is useful for data integration, data cleaning, information extraction systems,

scientific and sensor data management, approximate and hypothetical query processing, and other modern applications.

The databases managed by Trio are called *ULDBs*, for *Uncertainty-Lineage Databases*. ULDBs extend the standard relational model. Queries are expressed using *TriQL* (pronounced "treacle"), a strict extension to SQL. We have built a robust prototype system that supports a substantial fraction of the TriQL language over arbitrary ULDBs. The remainder of this Introduction briefly motivates the ULDB data model, the TriQL language, and the prototype system. Details are then elaborated in the rest of the chapter.

Examples in this chapter are based on a highly simplified "crime-solver" application, starting with two *base tables*:

- `Saw(witness,color,car)` contains (possibly uncertain) crime vehicle sightings.
- `Drives(driver,color,car)` contains (possibly uncertain) information about cars driven.

We will derive additional tables by posing queries over these tables.

The ULDB Data Model. Uncertainty is captured by tuples that may include several *alternative* possible values for some (or all) of their attributes, with optional *confidence* values associated with each alternative. For example, if a witness saw a vehicle that was a blue Honda with confidence 0.5, a red Toyota with confidence 0.3, or a blue Mazda with confidence 0.2, the sighting yields one tuple in table `Saw` with three alternative values for attributes `color`, `car`. 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 a red Mazda. Based on alternative values and confidences, each ULDB represents multiple *possible-instances* (sometimes called *possible-worlds*), where a possible-instance is a regular relational database.

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 `Drives` on attributes `color`, `car`. Lineage associated with a value in `Suspects` identifies the `Saw` and `Drives` values 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 (Section 4). If we generate further tables—`HighSuspects`, say—by issuing queries involving `Suspects` (perhaps together with other data), we get transitive lineage information: data in `HighSuspects` is derived from data

in `Suspects`, which in turn is derived from data in `Saw` and `Drives`. Trio supports arbitrarily complex layers of internal lineage.

As an example of external lineage, table `Drives` may be populated from various car registration databases, and lineage can be used to connect the data to its original source. Although Trio supports some preliminary features for external lineage, this chapter describes internal lineage only.

The TriQL Query Language. Section 1.5 specifies a precise generic semantics for any relational query over a ULDB, and Section 2 provides an operational description of Trio's SQL-based query language that conforms to the generic 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 to the data from which it was derived, as in the `Suspects` join query discussed above. Confidence values in query results are, by default, defined in a standard probabilistic fashion.

In addition to adapting SQL to Trio's possible-instances semantics in a straightforward and natural manner, TriQL includes a number of new features specific to uncertainty and lineage:

- Constructs for querying lineage, e.g., "find all witnesses contributing to `Jimmy` being a high suspect."
- Constructs for querying uncertainty, e.g., "find all high-confidence sightings," or "find all sightings with at least three different possible cars."
- Constructs for querying lineage and uncertainty together. e.g., "find all suspects whose lineage contains low-confidence sightings or drivers."
- Special types of aggregation suitable for uncertain databases, e.g., "find the expected number of distinct suspects."
- Query-defined result confidences, e.g., combine confidence values of joining tuples using *max* instead of multiplication.
- Extensions to SQL's data modification commands, e.g., to add new alternative values to an existing tuple, or to modify confidence values.
- Constructs for restructuring a ULDB relation, e.g, "flatten" or reorganize alternative values.

The Trio Prototype. The Trio prototype system is primarily layered on top of a conventional relational DBMS. From the user and application standpoint, the Trio system appears to be a "native" implementation of the ULDB model, TriQL query language, and other features. However, Trio encodes the

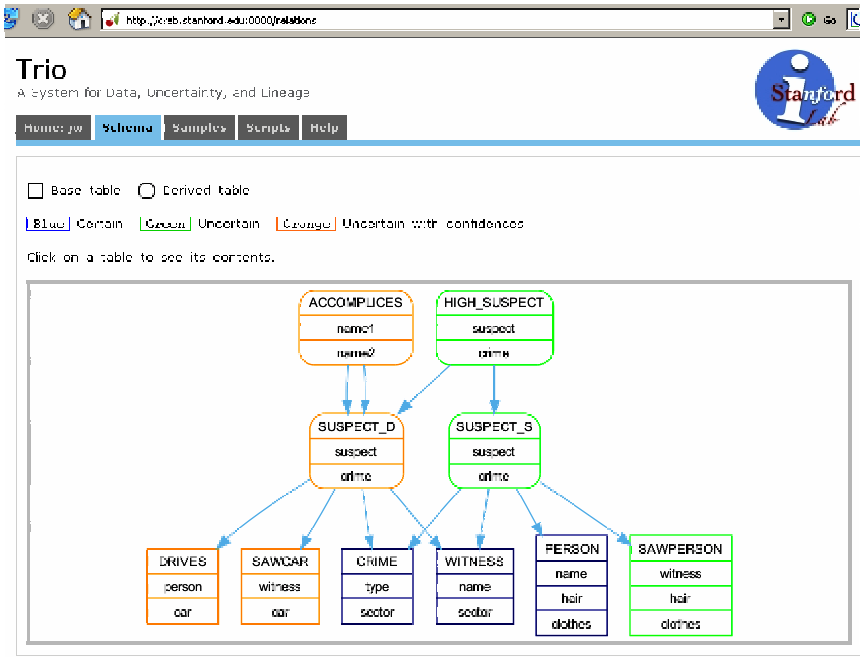

Figure 1.1. TrioExplorer Screenshot.

uncertainty and lineage in ULDB databases in conventional relational tables, and it uses a translation-based approach for most data management and query processing. A small number of stored procedures are used for specific functionality and increased efficiency.

The Trio system offers three interfaces: a typical DBMS-style API for applications, a command-line interface called *TrioPlus*, and a a full-featured graphical user interface called *TrioExplorer*. A small portion of the TrioExplorer interface is depicted in Figure 1.1. (The screenshot shows a *schema-level lineage graph*—discussed in Section 5—for a somewhat more elaborate crime-solver application than the running example in this chapter.) The Trio prototype is described in more detail in Section 6.

1. ULDBs: Uncertainty-Lineage Databases

The ULDB model is presented primarily through examples. A more formal treatment appears in [2]. ULDBs extend the standard SQL (multiset) relational model with:

1. *alternative values*, 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
4. *lineage*, connecting tuple-alternatives to other tuple-alternatives from which they were derived.

Each of these four constructs is specified next, followed by a specification of the semantics of relational queries on ULDBs.

1.1 Alternatives

ULDB relations have a set of *certain* attributes and a set of *uncertain* attributes, designated as part of the schema. Each tuple in a ULDB relation has one value for each certain attribute, and a set of possible values for the uncertain attributes. In table `Saw`, let `witness` be a certain attribute while `color` and `car` are uncertain. If witness Amy saw either a blue Honda, a red Toyota, or a blue Mazda, then in table `Saw` we have:

witness	(color, car)
Amy	(blue,Honda) || (red,Toyota) || (blue,Mazda)

This tuple logically yields three possible-instances for table `Saw`, one for each set of alternative values for the uncertain attributes. In general, the possible-instances of a ULDB relation R correspond to all combinations of alternative values for the tuples in R. For example, if a second tuple in `Saw` had four alternatives for `(color,car)`, then there would be 12 possible-instances altogether.

Designating certain versus uncertain attributes in a ULDB relation is important for data modeling and efficient implementation. However, for presentation and formal specifications, sometimes it is useful to assume all attributes are uncertain (without loss of expressive power). For example, in terms of possible-instances, the `Saw` relation above is equivalent to:

(witness, color, car)
(Amy,blue,Honda) || (Amy,red,Toyota) || (Amy,blue,Mazda)

When treating all attributes as uncertain, we refer to the alternative values for each tuple as *tuple-alternatives*, or *alternatives* for short. In the remainder of the chapter we often use tuple-alternatives when the distinction between certain and uncertain attributes is unimportant.

1.2 '?' (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 a red Mazda. In ULDBs, uncertainty about the existence of a tuple is denoted by a '?' annotation on the tuple. Betty's observation is thus added to table `Saw` as:

witness	(color, car)	
Amy	(blue,Honda) || (red,Toyota) || (blue,Mazda)	
Betty	(red,Mazda)	?

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

1.3 Confidences

Numerical *confidence* values may be attached to the alternative values in a tuple. Suppose Amy's confidence in seeing the 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	(color, car)	
Amy	(blue,Honda):0.5 ||(red,Toyota):0.3 ||(blue,Mazda):0.2	
Betty	(red,Mazda):0.6	?

Reference [2] formalizes an interpretation of these confidence values in terms of probabilities. (Other interpretations may be imposed, but the probabilistic one is the default for Trio.) Thus, if Σ is the sum of confidences for the alternative values in a tuple, then we must have $\Sigma \leq 1$, and if $\Sigma < 1$ then the 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 a ULDB relation itself has a probability, defined as the product of the confidences of the tuple-alternatives and '?"s comprising the instance. It can be shown (see [2]) that for any ULDB relation:

1. The probabilities of all possible-instances sum to 1.
2. The confidence of a tuple-alternative (respectively a '?') equals the sum of probabilities of the possible-instances containing this alternative (respectively not containing any alternative from this tuple).

An important special case of ULDBs is when every tuple has only one alternative with a confidence value that may be < 1. This case corresponds to the traditional notion of *probabilistic databases*.

In Trio each ULDB relation R is specified at the schema level as either *with confidences*, in which case R must include confidence values on all of its data, or *without confidences*, in which case R has no confidence values. However, it is permitted to mix relations with and without confidence values, both in a database and in queries.

1.4 Lineage

Lineage in ULDBs is recorded at the granularity of alternatives: lineage connects a tuple-alternative to those tuple-alternatives from which it was derived. (Recall we are discussing only internal lineage in this chapter. External lineage also can be recorded at the tuple-alternative granularity, although for some lineage types coarser granularity is more appropriate; see [12] for a discussion.) Specifically, lineage is defined as a function λ over tuple-alternatives: $\lambda(t)$ is a boolean formula over the tuple-alternatives from which the alternative t was derived.

Consider again the join of `Saw` and `Drives` on attributes `color`, `car`, followed by a projection on `driver` to produce a table `Suspects(person)`. Assume all attributes in `Drives` are uncertain. (Although not shown in the tiny sample data below, we might be uncertain what car someone drives, or for a given car we might be uncertain who drives it.) Let column ID contain a unique identifier for each tuple, and let (i, j) denote the jth tuple-alternative of the tuple with identifier i. (That is, (i, j) denotes the tuple-alternative comprised of i's certain attributes together with the jth set of alternative values for its uncertain attributes.) Here is some sample data for all three tables, including lineage formulas for the derived data in `Suspects`. For example, the lineage of the Jimmy tuple-alternative in table `Suspects` is a conjunction of the second alternative of `Saw` tuple 11 with the second alternative of `Drives` tuple 21.

`Saw`

ID	**witness**	**(color, car)**
11	Cathy	(blue,Honda) || (red,Mazda)

`Drives`

ID	**Drives (driver, color, car)**	
21	(Jimmy,red,Honda) || (Jimmy,red,Mazda)	**?**
22	(Billy,blue,Honda)	
23	(Hank,red,Mazda)	

`Suspects`

ID	**person**		
31	Jimmy	**?**	λ (31,1) = (11,2) $\wedge$ (21,2)
32	Billy	**?**	λ (32,1) = (11,1) $\wedge$ (22,1)
33	Hank	**?**	λ (33,1) = (11,2) $\wedge$ (23,1)

An interesting and important effect of lineage is that it imposes restrictions on the possible-instances of a ULDB: A tuple-alternative with lineage can be present in a possible-instance only if its lineage formula is satisfied by the presence (or, in the case of negation, absence) of other alternatives in the same possible-instance. Consider the derived table `Suspects`. Even though there is a '?' on each of its three tuples, not all combinations are possible. If Jimmy is present in `Suspects` then alternative 2 must be chosen for tuple 11, and therefore Hank must be present as well. Billy is present in `Suspects` only if alternative 1 is chosen for tuple 11, in which case neither Jimmy nor Hank can be present.

Thus, once a ULDB relation R has lineage to other relations, it is possible that not all combinations of alternatives and '?' choices in R correspond to valid possible-instances. The above ULDB has six possible-instances, determined by the two choices for tuple 11 times the three choices (including '?') for tuple 21.

Now suppose we have an additional base table, `Criminals`, containing a list of known criminals, shown below. Joining `Suspects` with `Criminals` yields the `HighSuspects` table on the right:

`Criminals`

ID	**person**
41	Jimmy
42	Frank
43	Hank

`HighSuspects`

ID	**person**	
51	Jimmy	**?** λ (51,1) = (31,1) ∧ (41,1)
52	Hank	**?** λ (52,1) = (33,1) ∧ (43,1)

Now we have multilevel (transitive) lineage relationships, e.g., $\lambda(51,1) = (31,1) \wedge (41,1)$ and $\lambda(31,1) = (11,2) \wedge (21,2)$. Lineage formulas specify direct derivations, but when the alternatives in a lineage formula are themselves derived from other alternatives, it is possible to recursively expand a lineage formula until it specifies base alternatives only. (Since we are not considering external lineage, base data has no lineage of its own.) As a very simple example, $\lambda(51,1)$'s expansion is $((11,2) \wedge (21,2)) \wedge (41,1)$.

Note that arbitrary lineage formulas may not "work" under our model—consider for example a tuple with one alternative and no '?' whose lineage (directly or transitively) includes the conjunction of two different alternatives of the same tuple. The tuple must exist because it doesn't have a '?', but it can't exist because its lineage formula can't be satisfied. Reference [2] formally defines *well-behaved* lineage (which does not permit, for example, the situation just described), and shows that internal lineage generated by relational 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. With well-behaved lineage our interpretation of confidences carries over directly: combining confidences on the base data determines the probabilities of the possible-instances,

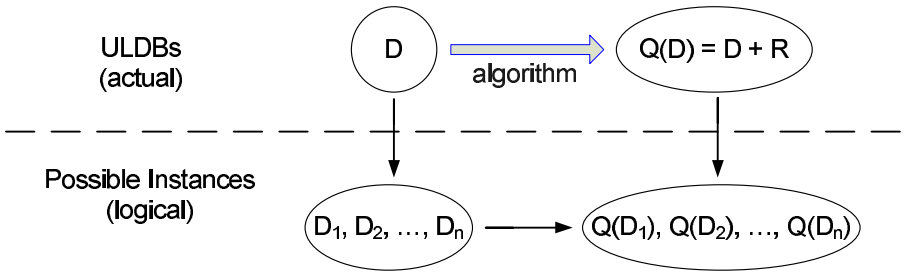

Figure 1.2. Relational Queries on ULDBs.

just as before. The confidence values associated with derived data items are discussed later in Section 4.

Finally, note that lineage formulas need not be conjunctive. As one example, suppose `Drives` tuple 23 contained `Billy` instead of `Hank`, and the `Suspects` query performed duplicate-eliminating projection. Then the query result is:

ID	**person**	
61	Jimmy	**?** λ (61,1) = (11,2) $\wedge$ (21,2)
62	Billy	λ (62,1) = ((11,1) $\wedge$ (22,1)) $\vee$ ((11,2) $\wedge$ (23,1))

Note that the lineage formula for tuple 62 is always satisfied since one alternative of base tuple 11 must always be picked. Thus, there is no '?' on the tuple.

1.5 Relational Queries

In this section we formally define the semantics of any relational query over a ULDB. Trio's SQL-based query language will be presented in Section 2. 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 1.2. 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 tables `Saw` and `Drives`, then logically it should join all of the possible-instances of these two ULDB relations. Of course we would never actually generate all possible-instances and operate on them, so a query processing algorithm follows the top arrow in Figure 1.2, producing a query result $Q(D)$ that represents the possible-instances.

A ULDB query result $Q(D)$ contains the original relations of D, together with a new *result relation* R. Lineage from R into the 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, and to enable consistent further querying of the original and new ULDB relations. (Technically, the possible-instances in the lower half of Figure 1.2 also contain lineage, but this aspect is not critical here; formal details can be found in [2].) The example in the previous subsection, with `Suspects` as the result of a query joining `Saw` and `Drives`, demonstrates the possible-instances interpretation, and lineage from query result to original data.

The ULDB model and the semantics of relational queries over it has been shown (see [2]) to exhibit two desirable and important properties:

- **Completeness:** Any finite set of possible-instances conforming to a single schema can be represented as a ULDB database.
- **Closure:** The result of any relational query over any ULDB database can be represented as a ULDB relation.

2. TriQL: The Trio Query Language

This section describes *TriQL*, Trio's SQL-based query language. Except for some additional features described later, 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.

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

```
SELECT Drives.driver as person INTO Suspects
FROM Saw, Drives
WHERE Saw.color = Drives.color AND Saw.car = Drives.car
```

If this query were executed as regular SQL over each of the possible-instances of `Saw` and `Drives`, as in the lower portion of Figure 1.2, it would produce the expected set of possible-instances in its result. More importantly, following the operational semantics given next, this query produces a result table `Suspects`, including lineage to tables `Saw` and `Drives`, that correctly represents those possible-instances.

This section first specifies an operational semantics for basic SQL query blocks over arbitrary ULDB databases. It then introduces a number of additional TriQL constructs, with examples and explanation for each one.

2.1 Operational Semantics

We provide an operational description of TriQL by specifying direct evaluation of a generic TriQL query over a ULDB, corresponding to the upper arrow in Figure 1.2. We specify evaluation of single-block queries:

```
SELECT attr-list [ INTO new-table ]
FROM T1, T2, ..., Tn
WHERE predicate
```

The operational semantics of additional constructs are discussed later, when the constructs are introduced. Note that in TriQL, the result of a query has confidence values only if all of the tables in the query's `FROM` clause have confidence values. (Sections 2.8 and 2.9 introduce constructs that can be used in the `FROM` clause to logically add confidence values to tables that otherwise don't have them.)

Consider the generic TriQL query block above; call it Q. Let *schema*(Q) denote the composition *schema*(`T1`) $\uplus$ *schema*(`T2`) $\uplus \cdots \uplus$ *schema*(`Tn`) 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 "`*`", 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 6) must produce the same result as this description.

1 Consider every combination $t_1, t_2, \ldots, t_n$ of tuples in `T1`, `T2`,..., `Tn`, one combination at a time, just as in SQL.

2 Form a "super-tuple" T whose tuple-alternatives have schema *schema*(Q). T has one alternative for each combination of tuple-alternatives in t_1, $t_2, \ldots, t_n$.

3 If any of $t_1, t_2, \ldots, t_n$ has a '?', add a '?' to T.

4 Set the lineage of each alternative in T to be the conjunction of the alternatives $t_1, t_2, \ldots, t_n$ 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 `T1`, `T2`,..., `Tn` are all tables with confidence values, then either compute the confidence values for T's remaining alternatives and store them (*immediate confidence computation*), or set the confidence values to `NULL` (*lazy confidence computation*). See Sections 2.8 and 4 for further discussion.

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

It can be verified easily that this operational semantics produces the `Suspects` result table shown with example data in Section 1.4. More generally it conforms to the "square diagram" (Figure 1.2) formal semantics given in Section 1.5. Later we will introduce constructs that do not conform to the square diagram because they go beyond relational operations.

Note that this operational semantics generates result tables in which, by default, all attributes are uncertain—it constructs result tuples from full tuple-alternatives. In reality, it is fairly straightforward to deduce statically, based on a query and the schemas of its input tables (specifically which attributes are certain and which are uncertain), those result attributes that are guaranteed to be certain. For example, if we joined `Saw` and `Drives` without projection, attribute `witness` in the result would be certain.

2.2 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 drivers having confidence > 0.8. We write:

```
SELECT Drives.driver as person INTO Suspects
FROM Saw, Drives
WHERE Saw.color = Drives.color AND Saw.car = Drives.car
  AND Conf(Saw) > 0.5 AND Conf(Drives) > 0.8
```

In the operational semantics, when we evaluate the `predicate` over the alternatives in T in step 6, `Conf(Ti)` refers to the confidence associated with the t_i component of the alternative being evaluated. Note that this function may trigger confidence computations if confidence values are being computed lazily (recall Section 2.1).

Function `Conf()` is more general than as shown by the previous example—it can take any number of the tables appearing in the `FROM` clause as arguments. For example, `Conf(T1,T3,T5)` would return the "joint" confidence of the t_1, t_3, and t_5 components of the alternative being evaluated. If t_1, t_3, and t_5 are independent, their joint confidence is the product of their individual confidences. If they are nonindependent—typically due to shared lineage—then the computation is more complicated, paralleling confidence computation for query results discussed in Section 4 below. As a special case, `Conf(*)` is shorthand for `Conf(T1,T2,...,Tn)`, which normally corresponds to the confidence of the result tuple-alternative being constructed.

2.3 Querying Lineage

For querying lineage, TriQL introduces a built-in predicate designed to be used as a join condition. If we include predicate `Lineage`(T_1, T_2) in the `WHERE` clause of a TriQL query with ULDB tables T_1 and T_2 in its `FROM`

clause, then we are constraining the joined T_1 and T_2 tuple-alternatives to be connected, directly or transitively, by lineage. For example, suppose we want to find all witnesses contributing to Hank being a high suspect. We can write:

```
SELECT S.witness
FROM HighSuspects H, Saw S
WHERE Lineage(H,S) AND H.person = 'Hank'
```

In the `WHERE` clause, `Lineage(H,S)` evaluates to true for any pair of tuple-alternatives t_1 and t_2 from `HighSuspects` and `Saw` such that t_1's lineage directly or transitively includes t_2. Of course we could write this query directly on the base tables if we remembered how `HighSuspects` was computed, but the `Lineage()` predicate provides a more general construct that is insensitive to query history.

Note that the the `Lineage()` predicate does not take into account the structure of lineage formulas: `lineage`(T_1, T_2) is true for tuple-alternatives t_2 and t_2 if and only if, when we expand t_1's lineage formula using the lineage formulas of its components, t_2 appears at some point in the expanded formula. Effectively, the predicate is testing whether t_2 had any effect on t_1.

Here is a query that incorporates both lineage and confidence; it also demonstrates the "`==>`" shorthand for the `Lineage()` predicate. The query finds persons who are suspected based on high-confidence driving of a Honda:

```
SELECT Drives.driver
FROM Suspects, Drives
WHERE Suspects ==> Drives
  AND Drives.car = 'Honda' AND Conf(Drives) > 0.8
```

2.4 Duplicate Elimination

In ULDBs, duplicates may appear "horizontally"—when multiple alternatives in a tuple have the same value—and "vertically"—when multiple tuples have the same value for one or more alternatives. As in SQL, `DISTINCT` is used to merge vertical duplicates. A new keyword `MERGED` is used to merge horizontal duplicates. In both cases, merging can be thought of as an additional final step in the operational evaluation of Section 2.1. (`DISTINCT` subsumes `MERGED`, so the two options never co-occur.)

As a very simple example of horizontal merging, consider the query:

```
SELECT MERGED Saw.witness, Saw.color FROM Saw
```

The query result on our sample data with confidences (recall Section 1.3) is:

witness	color
Amy	blue:0.7 || red:0.3
Betty	red: 0.6

?

Without merging, the first result tuple would have two `blue` alternatives with confidence values 0.5 and 0.2. Note that confidences are summed when horizontal duplicates are merged. In terms of the formal semantics in Section 1.5, specifically the square diagram of Figure 1.2, merging horizontal duplicates in the query answer on the top-right of the square corresponds cleanly to merging duplicate possible-instances on the bottom-right.

A query with vertical duplicate-elimination was discussed at the end of Section 1.4, where `DISTINCT` was used to motivate lineage with disjunction.

2.5 Aggregation

For starters, TriQL supports standard SQL grouping and aggregation following the relational possible-instances semantics of Section 1.5. Consider the following query over the `Drives` data in Section 1.4:

```
SELECT car, COUNT(*) FROM Drives GROUP BY car
```

The query result is:

ID	car	count
71	Honda	1 || 2
72	Mazda	1 || 2

λ(71,1) = $(22,1) \wedge \neg\,(21,1)$
λ(71,2) = $(21,1) \wedge (22,1)$
λ(72,1) = $(23,1) \wedge \neg\,(21,2)$
λ(72,2) = $(21,2) \wedge (23,1)$

Note that attribute `car` is a certain attribute, since we're grouping by it. Also observe that lineage formulas in this example include negation.

In general, aggregation can be an exponential operation in ULDBs (and in other data models for uncertainty): the aggregate result may be different in every possible-instance, and there may be exponentially many possible-instances. (Consider for example `SUM` over a table comprised of 10 maybe-tuples. The result has 2^{10} possible values.) Thus, TriQL includes three additional options for aggregate functions: a *low* bound, a *high* bound, and an *expected* value; the last takes confidences into account when present. Consider for example the following two queries over the `Saw` data with confidences from Section 1.3. Aggregate function `ECOUNT` asks for the expected value of the `COUNT` aggregate.

```
SELECT color, COUNT(*) FROM Saw GROUP BY car
SELECT color, ECOUNT(*) FROM Saw GROUP BY car
```

The answer to the first query (omitting lineage) considers all possible-instances:

color	count	
blue	1:0.7	?
red	1:0.54 || 2:0.18	?

The '?' on each tuple intuitively corresponds to a possible count of 0. (Note that zero counts never appear in the result of a SQL `GROUP BY` query.) The second query returns just one expected value for each group:

color	ecount
blue	0.7
red	0.9

It has been shown (see [9]) that expected aggregates are equivalent to taking the weighted average of the alternatives in the full aggregate result (also taking zero values into account), as seen in this example. Similarly, low and high bounds for aggregates are equivalent to the lowest and highest values in the full aggregate result.

In total, Trio supports 20 different aggregate functions: four versions (*full*, *low*, *high*, and *expected*) for each of the five standard functions (*count*, *min*, *max*, *sum*, *avg*).

2.6 Reorganizing Alternatives

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

- *Flatten* turns each tuple-alternative into its own tuple.
- *GroupAlts* regroups tuple-alternatives into new 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`" over the simple one-tuple `Saw` table from Section 1.4 gives:

witness	color	car	
Cathy	blue	Honda	?
Cathy	red	Mazda	?

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

color	car	person	
red	Honda	Jimmy	?
red	Mazda	Jimmy || Hank	
blue	Honda	Billy	

With `GROUPALTS`, the specified grouping attributes are certain attributes in the answer relation. For each set of values for these attributes, the corresponding tuple in the result contains the possible values for the remaining (uncertain)

attributes as alternatives. '?' is present whenever all of the tuple-alternatives contributing to the result tuple are uncertain.

FLATTEN is primarily a syntactic operation—if lineage is retained (i.e., if the query does not also specify NoLineage, discussed below), then there is no change to possible-instances as a result of including FLATTEN in a query. GROUPALTS, on the other hand, may drastically change the possible-instances; it does not fit cleanly into the formal semantics of Section 1.5.

2.7 Horizontal Subqueries

"Horizontal" subqueries in TriQL enable querying across the alternatives that comprise individual tuples. As a contrived example, we can select from table Saw all Honda sightings where it's also possible the sighting was a car other than a Honda (i.e., all Honda alternatives with a non-Honda alternative in the same tuple).

```
SELECT * FROM Saw
WHERE car = 'Honda' AND EXISTS [car <> 'Honda']
```

Over the simple one-tuple Saw table from Section 1.4, the query returns just the first tuple-alternative, (Cathy,blue,Honda), of tuple 11.

In general, enclosing a subquery in [] instead of () causes the subquery to be evaluated over the "current" tuple, treating its alternatives as if they are a relation. Syntactic shortcuts are provided for common cases, such as simple filtering predicates as in the example above. More complex uses of horizontal subqueries introduce a number of subtleties; full details and numerous examples can be found in [11]. By their nature, horizontal subqueries query "across" possible-instances, so they do not follow the square diagram of Figure 1.2; they are defined operationally only.

2.8 Query-Defined Result Confidences

A query result includes confidence values only if all of the tables in its FROM clause have confidence values. To assign confidences to a table T for the purpose of query processing, "UNIFORM T" can be specified in the FROM clause, in which case confidence values are logically assigned across the alternatives and '?' in each of T's tuples using a uniform distribution.

Result confidence values respect a probabilistic interpretation, and they are computed by the system on-demand. (A "COMPUTE CONFIDENCES" clause can be added to a query to force confidence computation as part of query execution.) Algorithms for confidence computation are discussed later in Section 4. A query can override the default result confidence values, or add confidence values to a result that otherwise would not have them, by assigning values in its SELECT clause to the reserved attribute name conf. Furthermore, a special "value" UNIFORM may be assigned, in which case confidence values are

assigned uniformly across the alternatives and '?' (if present) of each result tuple.

As an example demonstrating query-defined result confidences as well as `UNIFORM` in the `FROM` clause, suppose we generate suspects by joining the `Saw` table with confidences from Section 1.3 with the `Drives` table from Section 1.4. We decide to add uniform confidences to table `Drives`, and 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:

```
SELECT person, lesser(Conf(Saw),Conf(Drives)) AS conf
FROM Saw, UNIFORM Drives
WHERE Saw.color = Drives.color AND Saw.car = Drives.car
```

Let the two tuples in table `Saw` from Section 1.3 have IDs 81 and 82. The query result, including lineage, is:

ID	**person**		
91	`Billy:0.5`	**?**	λ (91,1) = (81,1) $\wedge$ (22,1)
92	`Jimmy:0.333`	**?**	λ (92,1) = (82,1) $\wedge$ (21,2)
93	`Hank:0.6`	**?**	λ (93,1) = (82,1) $\wedge$ (23,1)

With probabilistic confidences, `Jimmy` would instead have confidence 0.2. Had we used `greater()` instead of `lesser()`, the three confidence values would have been 1.0, 0.6, and 1.0 respectively.

With the "`AS Conf`" feature, it is possible to create confidence values in a tuple whose sum exceeds 1. ("`1.1 AS Conf`," assigning confidence value 1.1 to each result tuple-alternative, is a trivial example.) Although the Trio prototype does not forbid this occurrence, a warning is issued, and anomalous behavior with respect to confidence values—either the newly created values, or later ones that depend on them—may subsequently occur.

2.9 Other TriQL Query Constructs

TriQL contains a number of additional constructs not elaborated in detail in this chapter, as follows. For comprehensive coverage of the TriQL language, see [11].

- TriQL is a strict superset of SQL, meaning that (in theory at least) every SQL construct is available in TriQL: subqueries, set operators, `like` predicates, and so on. Since SQL queries are relational, the semantics of any SQL construct over ULDBs follows the semantics for relational queries given in Section 1.5.
- One SQL construct not strictly relational is `Order By`. TriQL includes `Order By`, but only permits ordering by certain attributes and/or the

special "attribute" `Confidences`, which for ordering purposes corresponds to the total confidence value (excluding '?') in each result tuple.

- In addition to built-in function `Conf()` and predicate `Lineage()`, TriQL offers a built-in predicate `Maybe()`. In a query, `Maybe(T)` returns true if and only if the tuple-alternative from table `T` being evaluated is part of a maybe-tuple, i.e., its tuple has a '?'.
- Horizontal subqueries (Section 2.7) are most useful in the `FROM` clause, but they are permitted in the `SELECT` clause as well. For example, the query "`SELECT [COUNT(*)] FROM Saw`" returns the number of alternatives in each tuple of the `Saw` table.
- As discussed in Section 2.8, preceding a table `T` in the `FROM` clause with keyword `UNIFORM` logically assigns confidence values to the tuple-alternatives in `T` for the duration of the query, using a uniform distribution. Similarly, "`UNIFORM AS conf`" in the `SELECT` clause assigns confidence values to query results using a uniform distribution. Another option for both uses is keyword `SCALED`. In this case, table `T` (respectively result tuples) must already have confidence values, but they are scaled logically for the duration of the query (respectively in the query result) so each tuple's total confidence is 1 (i.e., ?'s are removed). For example, if a tuple has two alternatives with confidence values 0.3 and 0.2, the `SCALED` confidences would be 0.6 and 0.4.
- Finally, three query qualifiers, `NoLineage`, `NoConf`, and `NoMaybe` may be used to signal that the query result should not include lineage, confidence values, or ?'s, respectively.

3. Data Modifications in Trio

Data modifications in Trio are initiated using TriQL's `INSERT`, `DELETE`, and `UPDATE` commands, which are in large part analogous to those in SQL. Additional modifications specific to the ULDB model are supported by extensions to these commands. The three statement types are presented in the following three subsections, followed by a discussion of how Trio incorporates *versioning* to support data modifications in the presence of derived relations with lineage.

3.1 Inserts

Inserting entirely new tuples into a ULDB poses no unusual semantic issues. (Inserting new alternatives into existing tuples is achieved through the `UPDATE` command, discussed below.) Trio supports both types of SQL `INSERT` commands:

```
INSERT INTO table-name VALUES tuple-spec
INSERT INTO table-name subquery
```

The `tuple-spec` uses a designated syntax to specify a complete Trio tuple to be inserted, including certain attributes, alternative values for uncertain attributes, confidence values, and/or '?,' but no lineage. The `subquery` is any TriQL query whose result tuples are inserted, together with their lineage (unless `NoLineage` is specified in the subquery; Section 2.9).

3.2 Deletes

Deletion also follows standard SQL syntax:

```
DELETE FROM table-name WHERE predicate
```

This command deletes each tuple-alternative satisfying the `predicate`. (Deleting a tuple-alternative is equivalent to deleting one alternative for the uncertain attributes; Section 1.1.) If all alternatives of a tuple are deleted, the tuple itself is deleted. A special qualifier "`AdjConf`" can be used to redistribute confidence values on tuples after one or more alternatives are deleted; without `AdjConf`, deleted confidence values implicitly move to '?.'

3.3 Updates

In addition to conventional updates, the TriQL `UPDATE` command supports updating confidence values, adding and removing '?'s, and inserting new alternatives into existing tuples. Consider first the standard SQL `UPDATE` command:

```
UPDATE table-name SET attr-list = expr-list WHERE predicate
```

This command updates every tuple-alternative satisfying the `predicate`, setting each attribute in the `attr-list` to the result of the corresponding expression in the `expr-list`.

There is one important restriction regarding the combination of certain and uncertain attributes. Consider as an example the following command, intended to rename as "Doris" every witness who saw a blue Honda:

```
UPDATE Saw SET witness = 'Doris'
WHERE color = 'blue' AND car = 'Honda'
```

In the `Saw` table of Section 1.1, the `WHERE` predicate is satisfied by some but not all of the `(color,car)` alternatives for witness `Amy`. Thus, it isn't obvious whether `Amy` should be be modified. Perhaps the best solution would be to convert `witness` to an uncertain attribute:

(witness,color, car)
(Doris,blue,Honda) || (Amy,red,Toyota) || (Amy,blue,Mazda)

However, Trio treats attribute types (certain versus uncertain) as part of the fixed schema, declared at CREATE TABLE time. A similar ambiguity can arise if the expression on the right-hand-side of the SET clause for a certain attribute produces different values for different alternatives. Hence, UPDATE commands are permitted to modify certain attributes only if all references to uncertain attributes, function Conf(), and predicate Lineage() in the WHERE predicate, and in every SET expression corresponding to a certain attribute, occur within horizontal subqueries. This restriction ensures that the predicate and the expression always evaluate to the same result for all alternatives of a tuple. For our example, the following similar-looking command updates every witness who *may* have seen a blue Honda to be named "Doris":

```
UPDATE Saw SET witness = 'Doris'
WHERE [color = 'blue' AND car = 'Honda']
```

To update confidence values, the special attribute conf may be specified in the attr-list of the UPDATE command. As with query-defined result confidences (Section 2.8), there is no guarantee after modifying conf that confidence values in a tuple sum to ≤ 1; a warning is issued when they don't, and anomalous behavior may subsequently occur. Finally, the special keywords UNIFORM or SCALED may be used as the expression corresponding to attribute conf in the SET clause, to modify confidence values across each tuple using uniform or rescaled distributions—analogous to the use of these keywords with "AS Conf" (Sections 2.8 and 2.9).

A variation on the UPDATE command is used to add alternatives to existing tuples:

```
UPDATE table-name ALTINSERT expression WHERE predicate
```

To ensure the predicate either holds or doesn't on entire tuples, once again all references to uncertain attributes, Conf(), and Lineage() must occur within horizontal subqueries. For each tuple satisfying the predicate, alternatives are added to the tuple, based on the result of evaluating the expression. Like the INSERT command (Section 3.1), the expression can be "VALUES tuple-spec" to specify a single alternative, or a subquery producing zero or more alternatives. Either way, the schema of the alternatives to add must match the schema of the table's uncertain attributes only. If adding alternatives to an existing tuple creates duplicates, by default horizontal duplicate-elimination does not occur, but it can be triggered by specifying UPDATE MERGED. As with other constructs that affect confidence values, creating tuples whose confidences sum to > 1 results in a warning.

Finally, the following self-explanatory UPDATE commands can be used to add and remove ?'s. These commands may only be applied to tables without confidences, and once again, in the predicate all references to uncertain attributes, Conf(), and Lineage() must be within horizontal subqueries.

```
UPDATE table-name ADDMAYBE WHERE predicate
UPDATE table-name DELMAYBE WHERE predicate
```

3.4 Data Modifications and Versioning

Trio query results include lineage identifying the input data from which the results were derived. Lineage is not only a user-level feature—it is needed for on-demand confidence computation, and it is critical for capturing the correct possible-instances in a query result (Section 1.4).

Suppose we run our `Suspects` query, store the result, then modifications occur to some alternatives in table `Saw` that are referenced by lineage in table `Suspects`. There are two basic options for handling such modifications:

(1) *Propagate* modifications to all derived tables, effectively turning query results into materialized views.

(2) *Don't propagate* modifications, allowing query results to become "stale" with respect to the data from which they were derived originally.

Option (1) introduces a variation on the well-known *materialized view maintenance problem*. It turns out Trio's lineage feature can be used here for broad applicability and easy implementation of the most efficient known techniques; see [6].

With option (2), after modifications occur, lineage formulas may contain incorrect or "dangling" pointers. Trio's solution to this problem is to introduce a lightweight *versioning* system: Modified data is never removed, instead it remains in the database as part of a previous version. The lineage formula for a derived tuple-alternative t may refer to alternatives in the current version and/or previous versions, thus accurately reflecting the data from which t was derived. Details of Trio's versioning system and how it interacts with data modifications and lineage can be found in [6].

4. Confidence Computation

Computing confidence values for query results is one of the most interesting and challenging aspects of Trio. In general, efficient computation of correct result confidence values in uncertain and probabilistic databases is known to be a difficult problem. Trio uses two interrelated techniques to address the problem:

1. By default, confidence values are not computed during query evaluation. Instead, they are computed on demand: when requested through one of Trio's interfaces, or as needed for further queries. This approach has two benefits: (a) Computing confidence values as part of query evaluation constrains how queries may be evaluated, while lazy computation frees

the system to select any valid relational query execution plan. (See [7] for detailed discussion.) (b) If a confidence value is never needed, its potentially expensive computation is never performed.

2. On-demand confidence computation is enabled by Trio's lineage feature. Specifically, the confidence of an alternative in a query result can be computed through lineage, as described below. Furthermore, a number of optimizations are possible to speed up the computation, also discussed below.

Suppose a query Q is executed producing a result table T, and consider tuple-alternative t in T. Assume all tables in query Q have confidence values (perhaps not yet computed), so t should have a confidence value as well. Formally, the confidence value assigned to t should represent the total probability of the possible-instances of result table T that contain alternative t (recall Section 1.3). It has been shown (see [7]) that this probability can be computed as follows:

1. Expand t's lineage formula recursively until it refers to base alternatives only: If $\lambda(t)$ refers to base alternatives only, stop. Otherwise, pick one t_i in $\lambda(t)$ that is not a base alternative, replace t_i with $(\lambda(t_i))$, and continue expanding.

2. Let f be the expanded formula from step 1. If f contains any sets $t_1, \ldots, t_n$ of two or more alternatives from the same tuple (a possible but unlikely case), then $t_1, \ldots, t_n$'s confidence values are modified for the duration of the computation, and clauses are added to f to encode their mutual exclusion; details are given in [7].

3. The confidence value for alternative t is the probability of formula f computed using the confidence values for the base alternatives comprising f.

It is tempting to expand formula $\lambda(t)$ in step 1 only as far as needed to obtain confidence values for all of the alternatives mentioned in the formula. However, expanding to the base alternatives is required for correctness in the general case. Consider for example the following scenario, where t_3, t_4, and t_5 are base alternatives.

$\lambda(t) = t_1 \wedge t_2 \quad \lambda(t_1) = t_3 \wedge t_4 \quad \lambda(t_2) = t_3 \wedge t_5$
$\texttt{Conf}(t_3) = \texttt{Conf}(t_4) = \texttt{Conf}(t_5) = 0.5$

Based on the specified confidences, we have $\texttt{Conf}(t_1) = \texttt{Conf}(t_2) = 0.25$. If we computed $\texttt{Conf}(t)$ using $t_1 \wedge t_2$ we would get 0.0625, whereas the correct value expanding to the base alternatives is 0.125. As this example demonstrates, lineage formulas must be expanded all the way to base alternatives because derived alternatives may not be probabilistically independent.

Trio incorporates some optimizations to the basic confidence-computation algorithm just described:

- Whenever confidence values are computed, they are *memoized* for future use.
- There are cases when it is not necessary to expand a lineage formula all the way to its base alternatives. At any point in the expansion, if all of the alternatives in the formula are known to be independent, and their confidences have already been computed (and therefore memoized), there is no need to go further. Even when confidences have not been computed, independence allows the confidence values to be computed separately and then combined, typically reducing the overall complexity. Although one has to assume nonindependence in the general case, independence is common and often can be easy to deduce and check, frequently at the level of entire tables.
- 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.

Reference [7] provides detailed coverage of the confidence-computation problem, along with our algorithms, optimizations, implementation in the Trio prototype.

5. Additional Trio Features

TriQL queries and data modifications 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 some interesting features beyond just queries and modifications.

Lineage. As TriQL queries are executed and their results are stored, and additional queries are posed over previous results, complex lineage relationships can arise. Data-level lineage is used for confidence computation (Section 4) and `Lineage()` predicates; it is also used for *coexistence checks* and *extraneous data removal*, discussed later in this section. The *TrioExplorer* graphical user interface supports data-level lineage tracing through special buttons next to each displayed alternative; the textual and API interfaces provide corresponding functionality.

Trio also maintains a schema-level lineage graph (specifically a DAG), with tables as nodes and edges representing lineage relationships. This graph is used when translating queries with `Lineage()` predicates (Section 6.7), and

for determining independence to optimize confidence computation (Section 4). This graph also is helpful for for users to understand the tables in a database and their interrelationships. A schema-level lineage graph was depicted in the Figure 1.1 screenshot showing the *TrioExplorer* interface.

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 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 (11,2) and (32,1) in our sample database of Section 1.4, the system would tell us these alternatives cannot coexist.

Checking coexistence is closely related to confidence computation. To check if alternatives t_1 and t_2 can coexist, we first expand their lineage formulas to reference base alternatives only, as in step 1 of confidence computation (Section 4). Call the expanded formulas f_1 and f_2. Let f_3 be an additional formula that encodes mutual exclusion of any alternatives from the same tuple appearing in f_1 and/or f_2, as in step 2 of confidence computation. Then t_1 and t_2 can coexist if and only if formula $f_1 \wedge f_2 \wedge f_3$ is satisfiable. Note that an equivalent formulation of this algorithm creates a "dummy" tuple t whose lineage is $t_1 \wedge t_2$. Then t_1 and t_2 can coexist if and only if $\texttt{Conf}(t) > 0$. This formulation shows clearly the relationship between coexistence and confidence computation, highlighting in particular that our optimizations for confidence computation in Section 4 can be used for coexistence checks as well.

Extraneous Data Removal. The natural execution of TriQL queries can generate *extraneous data*: an alternative is extraneous if it can never be chosen (i.e., its lineage requires presence of multiple alternatives 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), but checking can be expensive. Because we expect extraneous data and ?'s to be relatively uncommon, and users may not be concerned about them, by default Trio supports extraneous data removal as a separate operation, similar to garbage collection.

Like coexistence checking, extraneous data detection is closely related to confidence computation: An alternative t is extraneous if and only if $\texttt{Conf}(t) = 0$. A '?' on a tuple u is extraneous if and only if the confidence values for all of u's alternatives sum to 1.

6. The Trio System

Figure 1.3 shows the basic three-layer architecture of the Trio system. The core system is implemented in Python and mediates between the underlying re-

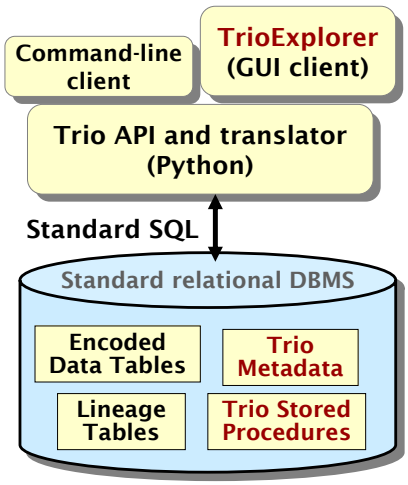

Figure 1.3. Trio Basic System Architecture.

lational 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 and modification commands in addition to regular SQL, and query results may be ULDB tuples as well as regular tuples. The API also exposes the other Trio-specific features described in Section 5. Using the Trio API, we built a generic command-line interactive client (*TrioPlus*) similar to that provided by most DBMS's, and the *TrioExplorer* graphical user interface shown earlier in Figure 1.1.

Trio DDL commands are translated via Python to SQL DDL commands based on the encoding to be described in Section 6.1. The translation is fairly straightforward, as is the corresponding translation of `INSERT` statements and bulk load.

Processing of TriQL queries 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 executes within the DBMS server. Processing of TriQL data modification commands is similar, although a single TriQL command often results in a larger number of SQL statements, since several relational tables in the encoding (Section 6.1) may all need to be modified.

TriQL query results can either be *stored* or *transient*. Stored query results (indicated by an `INTO` clause in the query) 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 (no `INTO` clause) 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.

It is not possible to cover all aspects of Trio's system implementation in this chapter. Section 6.1 describes how ULDB data is encoded in regular relations. Section 6.2 demonstrates the basic query translation scheme for `SELECT-FROM-WHERE` statements, while Sections 6.3–6.9 describe translations and algorithms for most of TriQL's additional constructs.

6.1 Encoding ULDB Data

We now describe how ULDB databases are encoded in regular relational tables. For this discussion we use *u-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. We store the data portion of T in two relational tables, T_C and T_U. Table T_C contains one tuple for each u-tuple in T. T_C's schema consists of the certain attributes of T, along with two additional attributes:

- `xid` contains a unique identifier assigned to each u-tuple in T.
- `num` contains a number used to track efficiently whether or not a u-tuple has a '?', when T has no confidence values. (See Section 6.2 for further discussion.)

Table T_U contains one tuple for each tuple-alternative in T. Its schema consists of the uncertain attributes of T, along with three additional attributes:

- `aid` contains a unique identifier assigned to each alternative in T.
- `xid` identifies the u-tuple that this alternative belongs to.
- `conf` stores the confidence of the alternative, or `NULL` if this confidence value has not (yet) been computed, or if T has no confidences.

Clearly several optimizations are possible: Tables with confidence values can omit the `num` field, while tables without confidences can omit `conf`. If a table T with confidences has no certain attributes, then table T_C is not needed since it would contain only `xid`'s, which also appear in T_U. Conversely, if T contains no uncertain attributes, then table T_U is not needed: attribute `aid` is unnecessary, and attribute `conf` is added to table T_C. Even when both tables are present, the system automatically creates a virtual view that joins the two tables, as a convenience for query translation (Section 6.2).

The system always creates indexes on T_C.`xid`, T_U.`aid`, and T_U.`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, which are translated by Trio to `CREATE INDEX` commands on the appropriate underlying tables.

The lineage information for each ULDB table T is stored in a separate relational table. Recall the lineage $\lambda(t)$ of a tuple-alternative t is a boolean formula. The system represents lineage formulas in *disjunctive normal form* (DNF), i.e., as a disjunction of conjunctive clauses, with all negations pushed to the "leaves." Doing so allows for a uniform representation: Lineage is stored in a single table T_L(`aid`, `src_aid`, `src_table`,`flag`), indexed on `aid` and `src_aid`. A tuple (t_1, t_2, T_2, f) in T_L denotes that T's alternative t_1 has alternative t_2 from table T_2 in its lineage. Multiple lineage relationships for a given alternative are conjunctive by default; special values for `flag` and (occasionally) "dummy" entries are used to encode negation and disjunction. By far the most common type of lineage is purely conjunctive, which is represented and manipulated very efficiently with this scheme.

Example. As one example that demonstrates many aspects of the encoding, consider the aggregation query result from Section 2.5. Call the result table R. Recall that attribute `car` is certain while attribute `count` is uncertain. The encoding as relational tables follows, omitting the lineage for result tuple 72 since it parallels that for 71.

R_C:

xid	num	car
71	2	Honda
72	2	Mazda

R_U:

aid	xid	count
711	71	1
712	71	2
721	72	1
722	72	2

R_L:

aid	src_aid	src_table	flag
711	221	Drives	NULL
711	211	Drives	neg
712	211	Drives	NULL
712	221	Drives	neg

For readability, unique `aid`'s are created by concatenating `xid` and alternative number. The values of 2 in attribute `R_C.num` indicate no '?'s (see Section 6.2), and `R_U.conf` is omitted since there are no confidence values. The remaining attributes should be self-explanatory given the discussion of the encoding above. In addition, the system automatically creates a virtual view joining tables `R_C` and `R_U` on `xid`.

6.2 Basic Query Translation Scheme

Consider the `Suspects` query from the beginning of Section 2, 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. The translated query refers to the virtual views joining `Saw_C` and `Saw_U`, and joining `Drives_C`, and `Drives_U`; call these views `Saw_E` and `Drives_E` ("`E`" for encoding) respectively.

```
SELECT Drives_E.driver,
       Saw_E.aid, Drives_E.aid, Saw_E.xid, Drives_E.xid,
       (Saw_E.num * Drives_E.num) AS num
FROM Saw_E, Drives_E
WHERE Saw_E.color = Drives_E.color AND Saw_E.car = Drives_E.car
ORDER BY Saw_E.xid, Drives_E.xid
```

Let *Tfetch* denote a cursor call to the Trio API for the original TriQL query, and let *Dfetch* denote a cursor call to the underlying DBMS for the translated SQL query. Each call to *Tfetch* must return a complete u-tuple, which may entail several calls to *Dfetch*: Each tuple returned from *Dfetch* on the SQL query corresponds to one alternative in the TriQL query result, and the set of alternatives with the same returned `Saw_E.xid` and `Drives_E.xid` pair comprise a single result u-tuple (as specified in the operational semantics of Section 2.1). Thus, on *Tfetch*, Trio collects all SQL result tuples for a single `Saw_E.xid`/`Drives_E.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 u-tuple.

Note that the underlying SQL query also returns the `aid`'s from `Saw_E` and `Drives_E`. These values (together with the table names) are used to construct the lineage for the alternatives in the result u-tuple. Recall that the `num` field is used to encode the presence or absence of '?': Our scheme maintains the invariant that an alternative's u-tuple has a '?' if and only if its `num` field exceeds the u-tuple's number of alternatives, which turns out to be efficient to maintain for most queries. This example does not have result confidence values, however even if it did, result confidence values by default are not computed until they are explicitly requested (recall Section 4). When a "`COMPUTE CONFIDENCES`" clause is present, *Tfetch* invokes confidence computation be-

fore returning its result tuple. Otherwise, *Tfetch* returns placeholder `NULL`s for all confidence values.

For the stored (`CREATE TABLE`) version of the query, Trio first issues DDL commands to create the new tables, indexes, and virtual view that will encode the query result. Trio then executes the same SQL query shown above, except instead of constructing and returning u-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 a stored procedure on the database server, thus avoiding unnecessary round-trips between the Python module and the underlying DBMS.

The remaining subsections discuss how TriQL constructs beyond simple `SELECT-FROM-WHERE` statements are translated and executed. All translations are based on the data encoding scheme of Section 6.1; many are purely "add-ons" to the basic translation just presented.

6.3 Duplicate Elimination

Recall from Section 2.4 that TriQL supports "horizontal" duplicate-elimination with the `MERGED` option, as well as conventional `DISTINCT`. In general, either type of duplicate-elimination occurs as the final step in a query that may also include filtering, joins, and other operations. Thus, after duplicate-elimination, the lineage of each result alternative is a formula in DNF (recall Section 6.1): disjuncts are the result of merged duplicates, while conjunction within each disjunct represents a tuple-alternative's derivation prior to merging; a good example can be seen at the end of Section 1.4. How Trio encodes DNF formulas in lineage tables was discussed briefly in Section 6.1.

Merging horizontal duplicates and creating the corresponding disjunctive lineage can occur entirely within the *Tfetch* method: All alternatives for each result u-tuple, together with their lineage, already need to be collected within *Tfetch* before the u-tuple is returned. Thus, when `MERGED` is specified, *Tfetch* merges all duplicate alternatives and creates the disjunctive lineage for them, then returns the modified u-tuple.

`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; this query produces a temporary result T. One scan through T is required to merge duplicates and create disjunctive lineage, then T is reordered by `xid`'s to construct the correct u-tuples in the final result.

6.4 Aggregation

Recall from Section 2.5 that 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*). All of the *full* functions and some of the other options cannot be translated to SQL queries over the encoded data, and thus are implemented as stored procedures. (One of them, *expected average*, is implemented as an approximation, since finding the exact answer based on possible-instances can be extremely expensive [9].) Many of the options, however, can be translated very easily. Consider table `Saw` with confidence values. Then the TriQL query:

```
SELECT color, ECOUNT(*) FROM Saw GROUP BY car
```

is translated based on the encoding to:

```
SELECT color, SUM(conf) FROM Saw_E GROUP BY car
```

A full description of the implementation of Trio's 20 aggregate functions can be found in [9].

6.5 Reorganizing Alternatives

Recall *Flatten* and *GroupAlts* from Section 2.6. 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 u-tuples, it groups by the attributes specified in `GROUPALTS` and generates new `xid`'s.

6.6 Horizontal Subqueries

Horizontal subqueries are very powerful yet surprisingly easy to implement based on our data encoding. Consider the example from Section 2.7:

```
SELECT * FROM Saw
WHERE car = 'Honda' AND EXISTS [car <> 'Honda']
```

First, syntactic shortcuts are expanded. In the example, `[car <> 'Honda']` is a shortcut for `[SELECT * FROM Saw WHERE car<>'Honda']`. Here, `Saw` within the horizontal subquery refers to the `Saw` alternatives in the current u-tuple being evaluated [11]. In the translation, 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:

```
... AND EXISTS (SELECT * FROM Saw_E S
                WHERE car <> 'Honda' AND S.xid = Saw_E.xid)
```

`S.xid=Saw_E.xid` restricts the horizontal subquery to operate on the data in the current u-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.

6.7 Built-In Predicates and Functions

Trio has three built-in predicates and functions: `Conf()` introduced in Section 2.2, `Maybe()` introduced in Section 2.9, and `Lineage()` introduced in Section 2.3.

Function `Conf()` is implemented as a stored procedure. If it has just one argument T, the procedure first examines the current T_E.`conf` field to see if a value is present. (Recall from Section 6.1 that T_E is the encoded data table, typically a virtual view over tables T_C and T_U.) If so, that value is returned. If T_E.`conf` is `NULL`, on-demand confidence computation is invoked (see Section 4); the resulting confidence value is stored permanently in T_E and returned.

The situation is more complicated when `Conf()` has multiple arguments, or the special argument "`*`" as an abbreviation for all tables in the query's `FROM` list (recall Section 2.2). The algorithm for arguments $T_1, \ldots, T_k$ logically constructs a "dummy" tuple-alternative t whose lineage is the conjunction of the current tuple-alternatives from $T_1, \ldots, T_k$ being considered. It then computes t's confidence, which provides the correct result for the current invocation of `Conf(`$T_1, \ldots, T_k$`)`. In the case of `Conf(*)`, the computed values usually also provide confidence values for the query result, without a need for on-demand computation.

The `Maybe()` and `Lineage()` predicates are incorporated into the query translation phase. Predicate `Maybe()` is straightforward: It translates to a simple comparison between the `num` attribute and the number of alternatives in the current u-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 for a brief discussion.)

Predicate `Lineage`(T_1,T_2) is translated into one or more SQL `EXISTS` subqueries that check if the lineage relationship holds: Schema-level lineage information is used to determine the possible table-level "paths" from T_1 to T_2. Each path produces a subquery that joins lineage tables along that path, with T_1 and T_2 at the endpoints; these subqueries are then `OR`'d to replace predicate `Lineage`(T_1,T_2) in the translated query.

As an example, recall table `HighSuspects` in Section 1.4, derived from table `Suspects`, which in turn is derived from table `Saw`. Then predicate `Lineage(HighSuspects, Saw)` would be translated into one subquery as follows, recalling the lineage encoding described in Section 6.1.

```
EXISTS (SELECT *
  FROM HighSuspects_L L1, Suspects_L L2
  WHERE HighSuspects.aid = L1.aid
  AND L1.src_table = 'Suspects' AND L1.src_aid = L2.aid
  AND L2.src_table = 'Saw' AND L2.src_aid = Saw.aid )
```

6.8 Query-Defined Result Confidences

The default probabilistic interpretation of confidence values in query results can be overridden by including "*expression* `AS conf`" in the `SELECT` clause of a TriQL query (Section 2.8). Since Trio's data encoding scheme uses a column called `conf` to store confidence values, "`AS conf`" clauses simply pass through the query translation phase unmodified.

6.9 Remaining Constructs

We briefly describe implementation of the remaining TriQL constructs and features.

- **Rest of SQL.** As mentioned in Section 2.9, since TriQL is a superset of SQL, any complete TriQL implementation must handle all of SQL. In our translation-based scheme, some constructs (e.g., `LIKE` predicates) can be passed through directly to the underlying relational DBMS, while others (e.g., set operators, some subqueries) can involve substantial rewriting during query translation to preserve TriQL semantics. At the time of writing this chapter, the Trio prototype supports all of the constructs discussed or used by examples in this chapter, as well as set operators `UNION`, `INTERSECT`, and `EXCEPT`.

- **Order By.** Because ordering by `xid`'s is an important part of the basic query translation (Section 6.2), `ORDER BY` clauses in TriQL require materializing the result first, then ordering by the specified attributes. When special "attribute" `Confidences` (Section 2.9) is part of the `ORDER BY` list, "`COMPUTE CONFIDENCES`" (Section 2.8) is logically added to the query, to ensure the `conf` field contains actual values, not placeholder `NULL`s, before sorting occurs.

- **UNIFORM and SCALED.** The keywords `UNIFORM` (Section 2.8) and `SCALED` (Section 2.9) can be used in a TriQL `FROM` clause to add or modify confidences on an input table, or with "`AS conf`" to specify confidences on the result. The "`AS conf`" usage is easy to implement within the *Tfetch* procedure (Section 6.2): *Tfetch* processes entire u-tuples one at a time and can easily add or modify confidence values before returning them.

  `UNIFORM` and `SCALED` in the `FROM` clause are somewhat more complex: Confidence computation for the query result must occur during query processing (as opposed to on-demand), to ensure result confidence values take into account the modifier(s) in the `FROM` clause. (Alternatively, special flags could be set, then checked during later confidence computation, but Trio does not use this approach.) Special process-

ing again occurs in *Tfetch*, which logically adds or modifies confidence values on input alternatives when computing confidence values for the query result.

- **NoLineage, NoConf, and NoMaybe.** These TriQL options are all quite easy to implement: `NoLineage` computes confidence values for the query result as appropriate (since no lineage is maintained by which to compute confidences later), then essentially turns the query result into a Trio base table. `NoConf` can only be specified in queries that otherwise would include confidence values in the result; now the result is marked as a Trio table without confidences (and, of course, does not compute confidence values except as needed for query processing). Finally, `NoMaybe` can only be specified in queries that produce results without confidences; all ?'s that otherwise would be included in the result are removed by modifying the `num` field in the encoding (Section 6.1).

- **Data modifications and versioning.** Recall from Section 3.4 that Trio supports a lightweight versioning system, in order to allow data modifications to base tables that are not propagated to derived tables, while still maintaining "meaningful" lineage on the derived data. Implementation of the versioning system is quite straightforward: If a ULDB table T is versioned, *start-version* and *end-version* attributes are added to encoded table T_U (Section 6.1). A query over versioned tables can produce a versioned result with little overhead, thanks to the presence of lineage. Alternatively, queries can request *snapshot* results, as of the current or a past version. Data modifications often simply manipulate versions rather than modify the data, again with little overhead. For example, deleting an alternative t from a versioned table T translates to modifying t's *end-version* in T_U. Reference [6] provides details of how the Trio system implements versions, data modifications, and the propagation of modifications to derived query results when desired.

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 Intl. Conference on Very Large Databases (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 Intl. Conference on*

Very Large Databases (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, P. Agrawal, S. Nabar, and J. Widom. Towards special-purpose indexes and statistics for uncertain data. In *Proc. of the Workshop on Management of Uncertain Data*, Auckland, New Zealand, August 2008.

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

[6] A. Das Sarma, M. Theobald, and J. Widom. Data modifications and versioning in Trio. Technical report, Stanford University InfoLab, March 2008. Available at: `http://dbpubs.stanford.edu/pub/2008-5`.

[7] A. Das Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, Cancun, Mexico, April 2008.

[8] A. Das Sarma, J.D. Ullman, and J. Widom. Schema design for uncertain databases. Technical report, Stanford University InfoLab, November 2007. Available at: `http://dbpubs.stanford.edu/pub/2007-36`.

[9] R. Murthy and J. Widom. Making aggregation work in uncertain and probabilistic databases. In *Proc. of the Workshop on Management of Uncertain Data*, pages 76–90, Vienna, Austria, September 2007.

[10] M. Mutsuzaki, M. Theobald, A. de Keijzer, J. Widom, P. Agrawal, O. Benjelloun, A. Das Sarma, R. Murthy, , and T. Sugihara. Trio-One: Layering uncertainty and lineage on a conventional DBMS. In *Proc. of Conference on Innovative Data Systems Research (CIDR)*, Pacific Grove, California, 2007.

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

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