# Data Modifications and Versioning in Trio

Anish Das Sarma, Martin Theobald, and Jennifer Widom
Stanford University
{anish,theobald,widom}@cs.stanford.edu

***Abstract*—**
**This paper presents the first DBMS for uncertain data that incorporates data modifications and a simple versioning system. Our work is in the context of *Trio*, a project at Stanford for managing data *uncertainty* and *lineage*. We establish SQL-based language constructs for data modifications, and an extended data model *ULDB*$^v$ that supports these modifications yielding versioned relations. We show that Trio's lineage feature enables answering a wide class of queries over ULDB$^v$s efficiently. We also give algorithms that leverage ULDB$^v$'s lineage for propagating data-modifications to Trio's derived relations efficiently and incrementally. We have incorporated the data modification and versioning capabilities in the Trio system, and we validate our techniques through experiments.**

## I. Introduction

Motivated by a diverse set of applications including data integration, deduplication, scientific data management, information extraction, and others, the *Trio* project at Stanford [29], [36] has been studying the combination of *uncertainty* and *lineage* as the basis for a new type of database management system. In this paper, we address data modifications and versioning in Trio. Prior to this paper, our work in Trio had been focused on data modeling and query processing.

The presence of uncertainty and lineage in Trio significantly changes the issues in data modification from those in traditional databases. First, there are new types of modifications to consider. In addition to the usual tuple inserts, deletes, and updates, in many applications the uncertainty in data may be revised to account for additional information: In a data-integration scenario, uncertainty may be reduced when source data or schema mappings are improved with user feedback [19], or uncertainty may be increased when new conflicting sources are added. In a sensor network, some sensors may be deemed unreliable, increasing the uncertainty associated with their readings, or a new algorithm may be introduced that resolves conflicting readings. Thus, new primitive modifications such as modifying *alternative* values and modifying *confidences* (*probabilities*) must be introduced.

A more interesting aspect of data modifications arises from Trio's approach to managing uncertain data. Trio's ability to use a data model that is both simple and expressive [7] relies crucially on *lineage*: query results are connected via lineage to the input data from which they were derived. (In brief, Trio uses lineage to correlate the uncertainty in query results with uncertainty in the input (base) data, and to compute confidence values correctly in the presence of correlations [16]. Of course applications may also wish to store and query lineage for its own sake.) Query results may be *transient* or they may be *stored* [29]. Stored results are connected permanently via lineage to the data from which they were derived. If the base data is modified, we have two options:

1. **Propagation:** Propagate the modifications to the query result (hereafter *derived relation*).
2. **No-Propagation:** Leave the derived relation as is. For correctness (and, we believe, for usability) the derived data's lineage should connect to the original input data from which it was derived, i.e., to a version of the data prior to modification. This option necessitates introducing a lightweight *versioning* feature into Trio.

This paper fully develops the first option, including complete integration of data modifications and versioning into the Trio data model, query processor, and prototype system. We will see that lineage plays a critical role in both modifications and versioning. We believe the overall combination of capabilities, and their ease of implementation based on existing Trio features, provides an elegant solution to data modifications and versioning in Trio.

First we establish language constructs for data modifications in Trio's query language, *TrioQL* [1] (Section III-A). We then introduce an extension to Trio's *ULDB* [7] data model, called ULDB$^v$ (Section III-B): In standard versioning fashion, each *alternative* in a Trio tuple now includes a *start version number* and an *end version number*, between which the alternative is "valid" as a possible value for the tuple. In Section III-C we tie together modifications and versioning by specifying how the primitive modification operations create versioned relations.

Our goal is not to develop a rich versioning system within Trio, but rather to incorporate "lightweight" versioning capabilities whose main function is to support meaningful data modifications in an environment with uncertainty and lineage. Thus we do not, for example, support historical modifications; modifications are always applied to the "current version" of the database. We also do not support rich constructs for referencing versions or histories. Queries over a ULDB$^v$ are expressed as regular queries, and they produce a "versioned answer:" the result is a ULDB$^v$ relation representing the query result across all versions. Queries can include a special clause that restricts the result to data valid in the current version—often more useful than the full versioned result, and far more efficient to compute. Furthermore, a user can view the "snapshot" of any relation (base or derived) as of any version. Again, these capabilities are not unusual or unique on their own; their novelty in Trio is how they elegantly support and are supported by Trio's management of uncertain data and

lineage.

Beyond defining data modifications and versioning in Trio, the main contributions of this paper are as follows:

- In Section IV we give the semantics of query answering in ULDB$^v$s and present detailed algorithms. We shall see that the ULDB$^v$ model introduces some subtleties with respect to defining lineage and *confidence values* on query results. However, determining start and end versions on query results adds little overhead: Trio's lineage feature enables a clean algorithm to compute versions on result tuples efficiently.
- In Section V we address the problem of modifying derived relations automatically when there are modifications to the relations from which they were derived, analogous to the well-known *materialized view maintenance* problem [22]. We refer to this process as "propagating modifications." We shall see that lineage provides us with critical information that enables efficient incremental view maintenance techniques for a very large class of derived relations.
- The suite of data modification and versioning capabilities described in this paper have been implemented in the Trio system. We believe ours is the first DBMS for uncertain data that incorporates data modifications and a simple versioning system. Section VI describes the implementation briefly and includes performance results.

Section II covers preliminaries, primarily a quick introduction to ULDBs and Trio. Related work is discussed at the end in Section VII, and we conclude with future work in Section VIII. To maintain the flow of the paper, proofs of all theorems are presented in an appendix.

## II. Preliminaries

We review Trio's *Uncertainty-Lineage Database* (ULDB) data model [7] through a running example. Each tuple in a ULDB relation consists of a set of mutually-exclusive *alternatives*, with associated *confidence* values.[1] Intuitively, the tuple takes the value of one of its alternatives, and the probability of taking a particular alternative is given by its confidence value. Consider a relation `Photo(Number,Name)` that lists names of people observed in a set of photos. Suppose Photo 1 clearly has `Amy` and also one other person who isn't very visible. This person looks most like Bob (60% chance), but may also be Carl (30%) or Dale (10%). We represent the relation as follows; "||" separates alternatives.

| ID | **Photo(Number, Name)** |
|---|---|
| 11 | `(1,Amy):1.0` |
| 12 | `(1,Bob):0.6` \|\| `(1,Carl):0.3` \|\| `(1,Dale):0.1` |

The sum of the confidence values of all alternatives in a tuple must be at most 1; if the sum is less than 1 (say, 0.8), then there is a 20% chance that none of the alternatives is present in the relation. Note the `ID` column attached to tuples in the relation above, which are maintained internally in Trio. We shall identify an alternative of a tuple by a pair $(i, j)$ where $i$ is the tuple ID, and $j$ is the index of the alternative in tuple $i$. For instance, `(1,Carl)` is identified by $(12, 2)$. These identifiers are needed by lineage, as we shall see shortly.

A ULDB relation has the standard *possible-worlds semantics* [4], [7], [14], [30], [33]. Each relation represents a set of possible worlds with associated probabilities. The possible worlds of a relation (without lineage) can be constructed by choosing exactly one alternative from each tuple. The probability of a possible world is given by the product of the confidences of all the alternatives chosen in that possible world.[2] For example. the probability of the possible world of `Photo` that contains alternatives `(1,Amy)` and `(1,Bob)` is 0.6.

While every possible world is a bag of tuples, a ULDB relation represents a set of possible worlds: If picking combinations of alternatives results in a set of possible worlds containing duplicates, the probabilities of duplicate possible worlds are added and one copy of each possible world is retained.

We add a second example ULDB relation to show how lineage is generated on performing queries. The relation `People(Name,State,Job)` lists people's state and occupation, which we know with certainty, i.e., each tuple has only one alternative and the confidence of that alternative is 1.0.

| ID | **People(Name, State, Job)** |
|---|---|
| 21 | `(Amy,CA,Engineer):1.0` |
| 22 | `(Bob,NY,Analyst):1.0` |
| 23 | `(Carl,IL,Teacher):1.0` |
| 24 | `(David,PA,Manager):1.0` |

Suppose we join `Photo` and `People` and project `Number` and `State` to obtain `States(Number,State)`, listing the states from which people appear in photos. We obtain the following relation:

| ID | **States(Number, State)** | |
|---|---|---|
| 31 | `(1,CA):1.0` | $\lambda((31,1)) = (11,1) \wedge (21,1)$ |
| 32 | `(1,NY):0.6` | $\lambda((32,1)) = (12,1) \wedge (22,1)$ |
| 33 | `(1,IL):0.3` | $\lambda((33,1)) = (12,2) \wedge (23,1)$ |

Lineage, denoted by $\lambda$, is shown alongside each tuple. Intuitively lineage captures when each alternative in the result is present. For instance, $\lambda((31,1)) = (11,1) \wedge (21,1)$, means alternative $(31,1)$ is present in `States` whenever alternatives $(11,1)$ and $(21,1)$ are present in their relations. The confidence of $(31,1)$ is given by the probability of the boolean formula represented by $\lambda((31,1))$; in this case we have $\Pr((31,1)) = 1.0 * 0.6 = 0.6$.

In general the lineage of an alternative in a derived relation is a boolean formula over alternative identifiers of other relations. We refer the reader to [7], [16] for detailed se-

[1] As a special case Trio also allows tuples with alternatives and no confidences. All results from this paper carry over to this special case as well.

[2] In case the sum of confidences of a tuple in a relation is $< 1$, then we need not pick any alternative from that tuple, and we multiply the confidence of not picking any alternative when computing the probability of the possible world.

mantics, algorithms for generation of lineage, and confidence computation in query answers.

## III. Overview of the System

We next provide an overview of the overall capabilities of the versioned Trio system by presenting the data modification language (Section III-A), data model (Section III-B), and how primitive modifications are executed (Section III-C).

### A. Data Modification Language

The following data modification statements extend TriQL [1], the SQL-based query language supported by Trio. In the interest of brevity, we provide an operational semantics in terms of Trio's ULDB data model for each construct. The semantics in terms of possible worlds are natural and unsurprising, hence presented formally only in the appendix.

**Insert Tuple:** A tuple can be inserted into ULDB relation $R$ as in SQL:

"Insert Into $R$ Values <tuple>"

In the query above, <tuple> specifies the tuple to be inserted by listing alternatives and confidence values, or by a subquery.

**Insert Alternative:** An alternative can be inserted into specific tuple(s) of $R$ as follows:

"Update $R$ AltInsert <alt>:<conf>
{Where <predicate>}"

This statement adds a new alternative <alt> with confidence <conf> to each tuple that has at least one alternative satisfying <predicate>. The insert statement is legal only if the sum of confidence values of all alternatives in each tuple to which <alt> is being added is at most $(1 - \texttt{conf})$.

**Delete:** Alternatives can be deleted from $R$ using the standard SQL syntax:

"Delete From $R$ {Where <predicate>}"

This statement deletes from $R$ each alternative satisfying <predicate>. When all alternatives of a tuple are deleted, the tuple itself is deleted from $R$.

**Update Alternative:** Alternative values can be updated in $R$ using the standard SQL syntax:

"Update $R$ Set <attr-list>=<expression-list>
{Where <predicate>}"

This statement updates every alternative of $R$ satisfying <predicate>. Each attribute in <attr-list> is updated with the result of the corresponding expression on the right-hand side of the =, just as in SQL.

**Update Confidences:** Finally, confidence values in $R$ can be modified as follows:

"Update $R$ Set conf=<expression>
{Where <predicate>}"

This statement modifies the confidence value of every alternative satisfying <predicate> based on <expression>. The modification is legal only if the sum of confidence values of all alternatives in each modified tuple is at most 1. (In Trio tables without confidence values, the symbol '?' is used to denote possible absence of a tuple [29]. An analogous Update statement to the one for modifying confidence values can be used to add or remove '?'.)

### B. Data Model

We define the ULDB$^v$ data model, a versioned extension to Trio's ULDB model. Some parts of the versioning model are standard, but subtleties do arise due to lineage and confidence. Also, in later sections we will see how the ULDB$^v$ model enables the data modification operations just described, in the Trio environment. We will also see how some typical operations on versioned databases are enabled by Trio's use of lineage.

Each alternative in a ULDB$^v$ database $D$ now includes a *start version* and an *end version*, which are non-negative integers or $\infty$. Suppose an alternative $a$ has start version $s$ and end version $e$, denoted by the interval $[s, e]$. Then, intuitively, the alternative is *valid* for all versions $v$ such that $s \leq v \leq e$. We use start$(a)$ and end$(a)$ to denote the start and end versions of alternative $a$. All other aspects of the original ULDB model, including lineage, remain unchanged.

The database $D$ maintains a current version $v_D$. Initially, the database starts at version 0, i.e. $v_D = 0$, and all alternatives have the interval $[0, \infty]$. ($\infty$ denotes the special version number greater than all integer versions.) We shall see in the next section that as data modifications are committed to $D$, the current version number $v_D$ is increased, and the intervals of modified alternatives are updated. (It is also possible to create an already-versioned database, but that seems to be a less likely scenario.)

Next we define the snapshot of a ULDB$^v$ and use it to define formally the semantics of ULDB$^v$s in terms of possible worlds.

*Definition 3.1 (Snapshot):* Given a ULDB$^v$ $D$ whose current version is $v_D$, the *snapshot* of $D$ at version $v$ (denoted $D_{@v}$) is a non-versioned ULDB obtained by eliminating from $D$ all alternatives $a$ such that $v <$ start$(a)$ or $v >$ end$(a)$. If all alternatives of some tuple are removed in the process, the tuple itself is removed. In the snapshot of a derived relation, the lineage of an alternative $a'$ is obtained by replacing every alternative identifier $i$ in $\lambda(a')$ with *false* if $i$'s version-interval does not include $v$. If now $\lambda(a') \equiv$ *false*, $a'$ is not included in the snapshot. The confidence value of $a'$ in the snapshot is determined by its lineage in the snapshot in the standard Trio fashion [16]. □

Intuitively, to obtain the snapshot of $D$ at $v$, we restrict data to alternatives whose version-intervals contain $v$. Lineage of

| ID | Photo(Number, Name) |
|---|---|
| 11 | (1,Amy)$^{[0,1]}$:1.0 |
| 12 | (1,Bob)$^{[0,\infty]}$:0.6 \|\| (1,Carl)$^{[0,0]}$:0.3 \|\| (1,Dale)$^{[1,1]}$:0.1 |
| 13 | (2,Eddie)$^{[0,1]}$:0.8 \|\| (2,Eddie)$^{[2,\infty]}$:1.0 \|\| (2,Frank)$^{[0,1]}$:0.2 |

(a)

| Photo$_{@1}$(Number, Name) |
|---|
| (1,Amy):1.0 |
| (1,Bob):0.6 \|\| (1,Dale):0.1 |
| (2,Eddie):0.8 \|\| (2,Frank):0.2 |

(b)

Fig. 1. (a) ULDB$^v$ Relation Photo with $v_D = 2$. (b) Snapshot of Photo at version 1.

each alternative $a'$ is obtained by restricting the lineage $\lambda(a')$ in $D$ to only contain alternatives that are present in $D_{@v}$. Alternatives that are not present in $D_{@v}$ do not contribute to $a'$ and hence are replaced by *false* in the lineage formula. The confidence value of each alternative is obtained as in non-versioned ULDBs [16], based on its lineage and confidences in $D_{@v}$. Note that the confidence value of the same alternative may be different in different snapshots; we discuss this point further in Section IV-C.

Note that for a snapshot to be a valid ULDB, the confidence values for each tuple must sum to at most 1. Thus, we consider a ULDB$^v$ database to be valid only if, for each version $v$, the sum of confidences for each tuple in the snapshot of $D$ at version $v$ is $\leq 1$. As we will see, if the aforementioned property is satisfied by base data imported into the system, our algorithms for data modifications and query answering preserve the validity of the ULDB$^v$ database.

Validity also requires that for any alternative $a$, start$(a) \leq v_D$, and if end$(a) \geq v_D$ then end$(a) = \infty$. (Recall $v_D$ denotes the current version.) Intuitively, all alternatives in the database must have come into existence at or before the current version. Similarly, any alternative that is valid at the current version is valid at and beyond the current version; i.e., no alternative could have been "killed" after the current version of the database. We then have the following straightforward result, which says that the database is constant from version $v_D$ onwards.

*Theorem 3.2:* Given a ULDB$^v$ $D$ whose current version is $v_D$, for versions $v_1, v_2 \geq v_D$, the snapshot $D_{@v_1}$ is the same as the snapshot $D_{@v_2}$. □

Just as ULDBs encode a set of possible worlds, ULDB$^v$s encode a list of sets of possible worlds: one set for each version $v \geq 0$.

*Definition 3.3 (Possible Worlds):* Given a ULDB$^v$ $D$ whose current version is $v_D$, the set of possible worlds of $D$ at version $v$ is the set of possible worlds of $D_{@v}$. □

*Example 3.4:* Figure 1 shows ULDB$^v$ relation Photo obtained by extending our running example from Section II with versioning, and adding a third tuple to it. Version-intervals for each alternative are marked as superscripts. The current version of the database is $v_D = 2$. For example, the alternative (1,Amy) is valid for versions 0 and 1, and the only alternatives valid in the current version are (1,Bob) and (2,Eddie).

We shall see later that confidence values can depend on versions, but for this example we assume the confidence value is same throughout an alternative's version-interval. The figure also shows the snapshot of Photo at version 1. (Tuple identifiers are omitted from the snapshot.) □

Just as the question of *completeness* arises in the theory of non-versioned uncertain databases [4], [6], [15], [24], [27], the corresponding question arises here: whether ULDB$^v$s are *complete for versioned uncertain databases*. That is, given some $v_D$ and a set of possible worlds $P_i$ for $0 \leq i \leq v_D$, is there a ULDB$^v$ $D$ whose current version is $v_D$ and $\forall i : 0 \leq i \leq v_D$, the possible worlds of $D_{@i}$ are equal to $P_i$?

*Theorem 3.5:* ULDB$^v$s are complete for versioned uncertain databases. □

### C. Executing Modifications

Next we address the problem of executing the primitive modifications described in Section III-A to a base ULDB$^v$ relation. While Section III-A presented the TriQL statements for requesting modifications, the modifications themselves, when executed, perform the following set of primitive operations:

- insert entire tuple
- insert alternative into existing tuple
- delete alternative from existing tuple (which, if there is only one alternative, deletes the entire tuple)
- update value or confidence in an existing alternative

We assume any number of modification statements may be performed together. A commit operation then increments the version number and installs the modifications permanently in the new version. (Again, the much-studied details of versioning systems, such as the interaction between versions and transactions, are neither a focus nor contribution of this paper.)

Consider a ULDB$^v$ relation $R$ whose current version is $v_D$:

1. **Insert Tuple:** When a tuple $t$ is inserted into $R$, all alternatives of $t$ are assigned the version-interval $[v_D + 1, \infty]$.
2. **Insert Alternative:** When an alternative $a$ is inserted into a tuple $t$, its version-interval is set to $[v_D + 1, \infty]$.
3. **Delete Alternative:** When an alternative $a$ is deleted, it is retained in $R$ as is, except end$(a)$, is modified to be $v_D$.
4. **Update Alternative:** When an alternative $a$ in tuple $t$ is updated to alternative $a'$, due to change in data value or confidence, end$(a)$ is modified to be $v_D$. Then, $a'$ is inserted as a new alternative in $t$ with version-interval $[v_D + 1, \infty]$.

| ID | People(Name, State, Job)[4] | |
|---|---|---|
| 22 | (Bob,NY,Analyst) [0,3] :1.0 \|\| (Bob,NY,Student) [4,∞] :1.0 | ← (4) update (Bob,NY,Analyst) to (Bob,NY,Student) |
| 23 | (Carl,IL,Teacher) [0,2] :1.0 | ← (3) delete alt. (Carl,IL,Teacher) |
| 24 | (David,PA,Manager) [0,∞] :0.6 \|\| (David,PA,CEO) [2,∞] :0.3 | ← (2) add alt. (David,PA,CEO):0.3 |
| 25 | (Frank,CA,Eng.) [1,∞] :0.3 \|\| (Frank,CA,Sr.Eng.) [1,∞] :0.7 | ← (1) insert (Frank,CA,Eng.):0.3\|\|(Frank,CA,Sr.Eng.):0.7 |

Fig. 2. People relation after executing a sequence of modifications.

*Example 3.6:* Consider the People relation from our running example, but now with versions and slightly modified data as below. The database is currently at version $v_D = 0$, indicated by the superscript on the relation name.

| ID | People(Name, State, Job)[0] |
|---|---|
| 22 | (Bob,NY,Analyst) [0,∞] :1.0 |
| 23 | (Carl,IL,Teacher) [0,∞] :1.0 |
| 24 | (David,PA,Manager) [0,∞] :0.6 |

Suppose we apply the following modifications in sequence, committing after each operation: (1) A new person is inserted: Frank from CA, whose job is either Engineer or Senior Engineer; (2) We decide that David is possibly (30% chance) a CEO, so an alternative is added to tuple 24; (3) Carl retires from his job, so alternative $(23, 1)$ is deleted; and (4) Bob decides to go to graduate school, so his job as Analyst is updated to Student.

Figure 2 shows the People relation after all the modifications have been applied. The final database version is $v_D = 4$. □

## IV. Query Answering

We now address the problem of query answering in ULDB$^v$s. Consider a query over a ULDB$^v$ database. In the query result, we are now interested not only in the data, confidence values, and lineage, but also in the version-intervals of each result alternative.

*Definition 4.1 (Query Answer):* Given a ULDB$^v$ $D$ whose current version is $v_D$ and a query $Q$ over relations $R_1, \ldots, R_n$ in $D$. The result of $Q$ is a ULDB$^v$ relation $R$ that has lineage to $R_1, \ldots, R_n$. The resulting ULDB$^v$ database $D'$ containing $D$ and $R$ still has current version $v_D$. For each version $v \leq v_D$, the possible worlds of $D'_{@v}$ are the possible worlds obtained by executing $Q$ on $D_{@v}$. □

Intuitively, the possible worlds of the result of a query correspond to applying the query to the set of possible worlds at each version of the database.

In Section IV-A we specify how version-intervals are computed in query results. Then, in Section IV-B we consider queries that specifically filter data based on their version-interval. Section IV-C discusses the subtleties surrounding confidence values in query results over ULDB$^v$s.

### A. Version-Interval Computation

In Section IV-A.1 we address query evaluation for queries that generate "positive" lineage, and extend our algorithm for all queries (i.e., also negative lineage) in Section IV-A.2. In Section IV-A.3 we briefly discuss pushing interval computation into the query execution engine.

*1) Positive Lineage:* We say that a relation $R$ has *positive* lineage if the lineage of every alternative in $R$ contains only positive literals. Results of queries involving select, project, join, union, and duplicate-elimination always have positive lineage.

Consider a query $Q$ over input relations $R_1, \ldots, R_n$ that generates positive lineage. Conceptually we answer $Q$ in two steps:

1. Data Computation: We compute the result $S$ of $Q$ by treating $R_1, \ldots, R_n$ as ULDB relations, disregarding the version-intervals of alternatives. The alternatives in $S$ and their lineage can be computed using any of Trio's query processing algorithms for ULDBs [7].
2. Version Computation: We use the lineage of each alternative in $S$ to compute its version-interval.

Let us look at performing Step 2. (We discuss performing Steps 1 and 2 together in Section IV-A.3.) Given an alternative $a$ in the result $S$, we want to know all the versions of $S$ in which $a$ is present in some possible world. Recall from the semantics of ULDBs that $a$ is present only if its lineage $\lambda(a)$ is *true*. Therefore, we need to know all versions in which $\lambda(a)$ can be evaluated to *true*. The following theorem describes how this set of all versions can be computed.

*Theorem 4.2 (Version Interval Computation):* Consider an alternative $a$ in a result relation, with lineage $\lambda(a)$ referring to alternatives $a_1, \ldots, a_m$ in the input relations. Let the version-intervals of $a_1, \ldots, a_m$ be $I_1, \ldots, I_m$. The version-interval $I$ of $a$ is computed by evaluating a formula $f$ obtained from $\lambda(a)$ as follows:

1. In $\lambda(a)$, replace every instance of each alternative $a_i$ by its version-interval $I_i$.
2. In the resulting expression, replace logical AND ("$\wedge$") with the intersection operator $\cap$ of intervals, and replace OR ("$\vee$") with the union operator $\cup$. □

*Corollary 4.3:* The version-interval of an alternative $a$ with positive lineage $\lambda(a)$ can be computed in PTIME in the number of alternatives in $\lambda(a)$. □

The above theorem translates the boolean lineage formula into an expression over version-intervals of alternatives, such that the result of the new expression gives the version-interval of the resulting alternative. If the lineage formula contains $(a_i \wedge a_j)$, replacing it by $I_i \cap I_j$ yields the versions in which $a_i \wedge a_j$

can be true. Similarly, $I_i \cup I_j$ gives the versions in which $a_i \vee a_j$ can be true. In general, replacing each boolean operator by the corresponding set operation and evaluating the formula yields the set of all versions in which $\lambda(a)$ can be true, i.e. all versions in which $a$ appears in some possible world.

Notice that the original query $Q$ that produced the result relation is not needed for computing version-intervals—they can be computed using just lineage and version-intervals for the input data. Lineage allowed us to decouple the two steps, instead of performing version computation as part of query evaluation. This decoupling is reminiscent of how Trio decouples data and confidence computations [16]. However, the second step of version computation is much easier than computing confidences: Confidence computation requires recursive expansion of lineage formulas, while version-intervals use only one level of lineage. As a result, version computation can be done in linear time while confidence computation is #P-complete [14].

*Example 4.4:* Consider `Photo` and `People` modified once again and omitting confidence values for this example:

| ID | **Photo(Number, Name)** |
|---|---|
| 12 | `(1,Bob)` $^{[2,\infty]}$ \|\| `(1,Carl)` $^{[4,6]}$ |
| 13 | `(2,Frank)` $^{[0,1]}$ |

| ID | **People(Name, State, Job)** |
|---|---|
| 22 | `(Bob,NY,Analyst)` $^{[0,3]}$ \|\| `(Bob,NY,Student)` $^{[4,\infty]}$ |
| 23 | `(Carl,IL,Teacher)` $^{[0,2]}$ |
| 25 | `(Frank,CA,Eng.)` $^{[1,\infty]}$ |

As before, suppose `States(Number,State)` is obtained by joining the two relations. After Step 1 (data computation), we have the following result alternatives and lineage:

| ID | **States(Number, State)** | |
|---|---|---|
| 41 | `(1,NY)` | $\lambda$((41,1)) = $(12,1) \wedge (22,1)$ |
| | \|\| `(1,NY)` | $\lambda$((41,2)) = $(12,1) \wedge (22,2)$ |
| 42 | `(1,IL)` | $\lambda$((42,1)) = $(12,2) \wedge (23,1)$ |
| 43 | `(2,CA)` | $\lambda$((43,1)) = $(13,1) \wedge (25,1)$ |

In Step 2 we compute the version-intervals of each alternative. For example, the interval of alternative (41,1) is $[2,\infty] \cap [0,3] = [2,3]$. Similarly, we compute the intervals for all other alternatives. We may obtain an empty interval for some alternatives, in which case the result alternative is *extraneous*, and is removed. For example, the interval of alternative (42,1) above is $[4,6] \cap [0,2] = \emptyset$. The final result after interval computation is:

| ID | **States(Number, State)** | |
|---|---|---|
| 41 | `(1,NY)` $^{[2,3]}$ | $\lambda$((41,1)) = $(21,1) \wedge (22,1)$ |
| | \|\| `(1,NY)` $^{[4,\infty]}$ | $\lambda$((41,2)) = $(12,1) \wedge (22,2)$ |
| 43 | `(2,CA)` $^{[1,1]}$ | $\lambda$((43,1)) = $(13,1) \wedge (25,1)$ |

Now let us consider a further derived `Region(State)`, obtained by projecting onto attribute `State` and eliminating duplicates from the above relation, which generates disjunctive lineage. The resulting relation after data and version computation is:

| ID | **Region(State)** | |
|---|---|---|
| 51 | `(NY)` $^{[2,\infty]}$ | $\lambda$((51,1)) = $(41,1) \vee (41,2)$ |
| 52 | `(CA)` $^{[1,1]}$ | $\lambda$((52,1)) = $(43,1)$ |

After duplicate elimination (or other queries) it is possible that the version-interval computed based on Theorem 4.2 consists of a set of disjoint intervals as opposed to a single interval. Disjoint sets of intervals are represented in Trio by creating multiple alternatives, each with a single interval. □

*2) Arbitrary Lineage:* Now we consider computing version-intervals of result relations with arbitrary lineage. Specifically, the lineage of an alternative may now also contain negation, typically generated by a query involving the difference operator (e.g., $R_1$ `EXCEPT` $R_2$).

As a first step, we ensure $\lambda(a)$ is a DNF formula with all negations pushed down to literals (using De Morgans laws), which is how lineage formulas are stored in Trio anyway. Just as before, we replace every conjunction with intersection, disjunction with union, and positive literal $a_i$ with $a_i$'s version-interval $I_i$.

However, we now replace every negated literal $\neg a_j$ with the interval $[0,\infty]$. One would have expected replacing $\neg a_j$ with the complement of the interval $I_j$, but that is incorrect. Consider an alternative $a_j$, whose confidence value is less than 1. Recall we are trying to find the version-interval in which $\neg a_j$ could be *true*. Clearly for any version $v$ not contained in $I_j$ we could have $\neg a_j$ *true* since $a_j$ is not present at version $v$; in addition, even for some version $v \in I_j$, $a_j$ has some probability of not being present, hence $\neg a_j$ could still be true. Therefore, $\neg a_j$ is replaced by $[0,\infty]$ while constructing the expression over version-intervals. After that, the version-interval of each alternative is computed as before. If we know $a_j$ has confidence 1 in $I_j$, then we can make the intuitive replacement of $\neg a_j$ with $I'_j$, the complement of $I_j$, to compute the exact version-interval. However, the following theorem shows that the hardness of confidence computation [16] makes exact version-interval computation also intractable. Further, it shows that the algorithm described above correctly computes version-intervals in PTIME when confidence values are known. The proof of this theorem appears in the Appendix.

*Theorem 4.5:* Computing the version-interval of an alternative $a$ with arbitrary lineage $\lambda(a)$ referring to alternatives $a_1, \ldots, a_m$ is:

- PTIME when all $a_i$'s confidence values are known.
- NP-hard when confidences of $a_i$'s are unknown. □

Despite this "negative" theorem, in the absence of negation our algorithm can always compute version-intervals in linear time. In the presence of negation, if confidence values are known then our algorithm is still linear. If confidence values are not known, our algorithm finds the best conservative interval based purely on lineage and input version-intervals.

*3) Pushing Interval Computation into Query Plans:* Lineage enables version-interval computation to be performed as a separate step, but we may also wish to perform version-

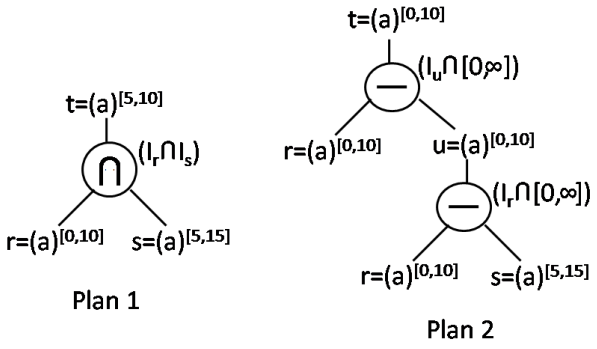


Fig. 3. Pushing version-interval computation into query plans with negation.

interval computation during query execution. In the absence of negation, we can do so in a standard fashion: `select` and `project` operators retain the version-interval of the input, the `duplicate-elimination` operator takes the union of the version-intervals of the input alternatives, and the `join` operator intersects the version-intervals of the input alternatives. If the version-interval of an intermediate alternative in the query plan is empty (for instance, as a result of a join), the alternative is dropped.

When a query includes negation, computing version-intervals within a query plan is not always possible, as shown in the following example.

*Example 4.6:* Consider performing an intersection of relations $R$ and $S$, each containing one tuple $(a)$ with probability $< 1$, using two query plans: (1) $(R \cap S)$, and (2) $(R-(R-S))$. (Assume $R$ and $S$ have no duplicates so the two query plans are equivalent.)

Let the tuples in $R$ and $S$ be denoted $r$ and $s$ and let their version-intervals be $[0, 10]$ and $[5, 15]$ respectively. Figure 3 shows the operator-by-operator interval computation based on both the query plans. The lineage formulas for the final result tuple are equivalent: $r \wedge s$ for the first plan and $r - (r - s) \equiv r \wedge \neg(r \wedge \neg s)$ for the second plan. Plan 1 produces the correct output version-interval $[5, 10]$. However, in the second plan an incorrect version-interval of $[0, 10]$ is produced. □

In the above example, intuitively the reason the interval computation is incorrect is that the same tuple is involved in multiple levels of negation, not taken into account in operator-by-operator interval manipulations. Even without double-negations, interval computations within a query plan with negation may be incorrect. Hence, for such queries it is necessary to support some "lineage-like" method for decoupling data and interval computation.

The system we have built pushes version-interval computation into the query plan whenever possible. We have also built suitable indexes (see Section VI) on Trio's start and end version columns to enable efficiently combining and intersecting version-intervals in the query plan.

### B. Filtering based on Versions

Next we enumerate two new constructs that enable filtering data based on their version.

**1. Valid-At Queries:** We can add the clause "`valid at <version-number>`" to any regular Trio query, to filter out input alternatives not valid at a particular version. For example, "`Select * from People valid at 3`" selects from `People` all alternatives whose version-interval contains 3. Note that while valid-at queries filter out data based on a specified version, their result is still a versioned ULDB$^v$ relation.

**2. Snapshot Viewing:** Our system allows users to view a snapshot of any ULDB$^v$ relation using the command: "`View <table-name> at <version-number>`". For example, "`View People at 4`" displays `People`$_{@4}$. Note that here the result is a (non-versioned) ULDB relation.

### C. Confidence Values

An important subtlety arises when combining derived relations, confidence values, and versioning. The subtlety occurs only in query results with disjunctive lineage, typically due to duplicate elimination. While an alternative can be present in multiple versions of data, its confidence value may vary in different versions of the data.

A simple example illustrates the issue. Suppose the `States` relation from earlier, but modified again, were a base relation as follows:

| ID | States(Number, State) |
|---|---|
| 41 | (1,NY) $^{[0,3]}$: 1.0 |
| 43 | (1,CA) $^{[0,\infty]}$: 0.5 |

If we perform a duplicate-eliminating projection onto `Number`, the result contains a single alternative `[1]`. The confidence value of `[1]` from versions 0 to 3 is 1.0, because alternative (41,1) is present with confidence 1.0. However, the confidence value from versions 4 onwards is 0.5, because from version 4 onwards only alternative (43,1) is present.

All information needed to determine confidence values for any version is included in result lineage, but there is a question of presentation and clarity. One possibility would be to split alternatives based on confidence values. For example, in the query result above we could have two result alternatives with value `[1]`, the first having version-interval $[0, 3]$ and confidence 1.0, the second having version-interval $[4, \infty]$ and confidence 0.5. In the Trio system we decided instead to display confidence values for the current version $v_D$ as default (though even these values are typically computed lazily; see [16]). Historical confidence values can be provided on request.

## V. Propagating Modifications

Next we address the problem of propagating the modifications on base data described in Section III-A to derived

relations that are dependent on the modified data. It suffices to consider propagating modifications to derived relations that are derived directly from modified base data. Modifications can be propagated to an entire ULDB$^v$ database in topological order: Once the set of relations, say $I(R)$, derived from a modified base relation $R$ are modified, then relations derived from $I(R)$ are modified, and so on.

As we worked on the propagation component of data modifications, and studied the considerable body of related previous work on materialized view maintenance, we realized that most algorithms for incremental view maintenance (particularly to handle deletions) exploit some technique or extra information that, if one looks carefully, is really a type of lineage. Thus, the lineage maintained by Trio anyway allows us to apply the most efficient propagation techniques, without requiring, gathering, or maintaining additional information. Specifically, we are able to identify data in derived relations that refers to modified data easily. Furthermore, we are able to propagate deletions for all types of queries, including negation, incrementally and without using the original query. (Deletions from the right side of a difference operator have traditionally been a particularly difficult case.)

First we discuss previous techniques in light of lineage. Then we give two examples of how our propagation algorithm handles deletions. Detailed algorithms for propagating all types of modifications are given in the appendix.

**Lineage and Traditional View Maintenance:** A large body of work in traditional materialized view maintenance (surveyed in [22]) studies a restricted class of queries for derived relations and modifications (e.g., insertions with SPJ derived relations) for which incremental maintenance is possible. A parallel body of work [9], [12], [31], [32] studies conditions under which incremental maintenance is possible, perhaps using some additional information such as keys. We elaborate on this work in Section VII, but the main insight in this section is that lineage in ULDB$^v$s freely provides us with the required information to leverage efficient view maintenance techniques for a large class of derived relations. For instance, lineage captures more information than keys and hence allows us to propagate modifications even when no base data contains any key. Similarly, propagating deletions in the presence of lineage can be done incrementally in our setting.

**Deletion with Positive Lineage:** Recall that to delete an alternative $a$ from a relation $R$, we simply set $\text{end}(a) = v_D$. Propagating deletes for any derived relation $T$ containing only positive lineage is very easy. (Recall that all combinations of select, project, join, union, and duplicate-elimination generate only positive lineage in their result relations.) When alternatives from $T$'s input relations are deleted, we simply recompute the version-intervals of all alternatives in $T$ whose lineage refers to deleted alternatives. Lineage allows us to easily determine which alternatives in $T$ refer to deleted alternatives.

**Deletion with EXCEPT:** Consider derived relation $T$ obtained by executing "$R_1$ EXCEPT $R_2$", where $R_1$ and $R_2$ are base relations. Deleted alternatives in $R_1$ are again propagated to $T$ by recomputing the version-intervals of affected alternatives in $T$. The only case when recomputation of version-intervals in $T$ doesn't give the correct answer is when we delete alternatives from $R_2$: Suppose there's an alternative $a$ in $R_1$ that also appears in $R_2$ with version-interval $[0, \infty]$ and confidence 1.0. Then when $R_1 - R_2$ is executed, since the alternative $a$ is certainly present in $R_2$, it does not appear in the result relation. However, if the alternative gets deleted from $R_2$, then $a$ needs to be included in the result.

If a set $S$ of alternatives is deleted from $R_2$, we modify $T$ as follows. For each distinct alternative value $a$ in $S$, check whether $T$ contains alternatives with value $a$:

1. If yes, recompute the version-intervals of the alternatives in $T$ with value $a$.
2. If not check whether any alternative in $R_1$ has the value $a$. For each such alternative $a_1$ from tuple $t_1$ in $R_1$:
   (a) If $T$ contains any tuple with lineage referring to alternatives of $t_1$, insert the alternative $a_1$ into it. If not, add a new tuple in $T$ containing $a_1$.
   (b) Let $s_1, \ldots, s_k$ be all alternatives in $R_2$ with the same value as $a_1$. Set the lineage of the newly added alternative in $T$ to $(a_1 \wedge \neg s_1 \wedge \ldots \wedge \neg s_k)$, and compute its version-interval accordingly.

Effectively, we only need to insert alternatives into $T$ that were not included because $R_2$ contained certain alternatives with the same value. For the remaining alternatives in $T$, if they refer to modified alternatives, we recompute their version-intervals; and if not, they are unchanged. Once again. the task of determining which alternatives in $T$ refer to modified alternatives is made simple by lineage.

## VI. System and Experiments

### A. System Description and Setup

Trio encodes ULDB$^v$ relations in standard relational tables, and lineage of each derived ULDB$^v$ relation is materialized into a separate table. Reference [29] gives details of our encoding prior to our extension of versioning. With versioning, in addition to data columns, each alternative in the encoding is associated with two system columns—`start` and `end`—corresponding to start and end versions respectively.

Trio uses the PostgreSQL 8.2 open-source DBMS as its relational back-end. Experiments were conducted on a Quad-Xeon server with 16 GB RAM and a large SCSI RAID. For all queries, we report actual wall-clock runtimes (in seconds) to execute the query and place the result in a new uncertain relation. Lineage and version-intervals, (but no confidences), are computed and stored. Query execution time is measured over a "hot" cache, i.e., by running a query once and then reporting the average runtime over three subsequent, identical executions. Our experiments focus on investigating the over-

head of version management in Trio and its effects on query processing, compared to a non-versioned Trio implementation.

Our dataset is based on a synthetically created set of TPC-H [35] tables using a scale-factor of 1. For modifications and queries, we consider different subsets of the `Lineitem` and `Orders` tables, by varying the selectivity of selections over the `Orderkey` attribute from 0.1% to 1% of the input table size. To make the TPC-H data uncertain, we group the original tuples into uncertain tuples according to a uniform-random distribution of between 1 and 10 alternatives for each tuple. This way, we obtain uncertain relations of 6,000 (at a selectivity of 0.1%) to 60,000 (at a selectivity of 1%) total alternatives for `Lineitem`, and between 1,500 and 15,000 alternatives for `Orders`, grouped into 1,107 to 10,979 uncertain tuples for `Lineitem` and 263 to 2,745 uncertain tuples for `Orders`, respectively. Within this setting, we are able to study systematically different series of smaller, repeated data modifications and their effect on query executions over various lineage- and version-enabled DBMS operators. We note that other recent work [5], [14], [16] on uncertain and probabilistic databases has used similarly generated synthetic data sets based on TPC-H for experimental studies.

In addition to indexes over data columns, to facilitate answering predicates over the version columns, we create a multi-attribute B+ tree index over the concatenation of $(start, end)$ and a B+ tree index over $end$ for each input table in the versioned Trio system. Ensuring nonempty intersection of a set of version-intervals $[s_1, e_1], \ldots, [s_n, e_n]$ then translates to the predicates $\texttt{max}_i(s_i) \leq \texttt{min}_i(e_i)$. For valid-at and snapshot queries referring to a given version $v$, we include predicates $s_i \leq v \leq e_i$ for each table in the `FROM` clause. Hence, queries may use range scans over the B+ tree indexes, in addition to other indexes that may be involved in the data-related part of the query processing. Unless mentioned otherwise, all versioned query executions in our experiments combine data and version computation using the predicates above, as described in Section IV-A.3.

### B. Experimental Results

*1) Baseline Runs Without Modifications:* We start by comparing the performance of a non-versioned Trio system with version-enabled Trio. In Figure 4 we use a join of `Lineitem` and `Orders` on `Orderkey`, varying join selectivity from 0.1% to 1%. No modifications have been performed at this point. As expected, Figure 4 shows join query executions as perfectly linear functions of the join selectivity (the join multiplicity is constant at 1). The gap between the two plots indicates the overhead of processing the two system columns for $start$ and $end$ intervals to the encoding: We notice only a small (10-12%) overhead in the execution times.

*2) Scalability:* Next we study the same join query, but when an increasing number of alternatives that go into the join have been modified in both the `Lineitem` and `Orders` tables. Alternatives were modified by only changing their internal identifier, but retaining the same value. Figure 5 displays

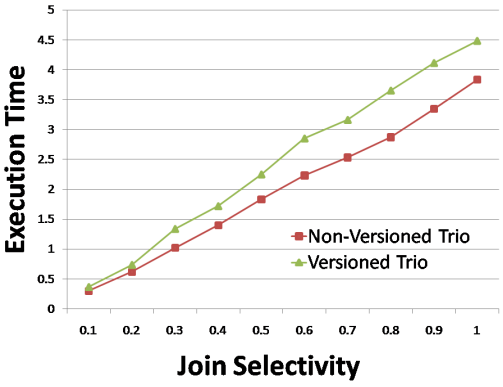


Fig. 4. Overhead of join query processing in versioned Trio, no modifications.

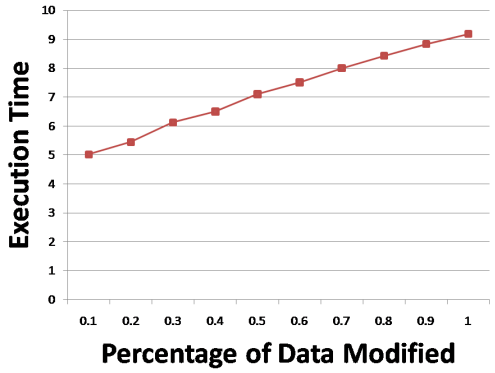


Fig. 5. Scalability of query processing for varying amounts of modified data.

runtime as a function of the number of alternatives that have been modified, varying modification selectivity from 0.1% to 1%. The join selectivity is fixed at 1. We see that our versioning algorithm clearly scales linearly with the amount of data modified. Figure 5 also shows that query execution time almost exactly doubles when all alternatives that go into the join have been modified once (at a modification selectivity of 1%), compared to executing the same join query without modified data as depicted before in Figure 4. This conforms to the best-possible case, since for each modified (and hence expired) alternative, a new alternative with a different version-interval is inserted into each of the input tables, which also exactly doubles the amount of versioned alternatives in the result at 1% update selectivity.

Figure 6 addresses query processing behavior when different fractions of alternatives are modified progressively, i.e., more than once. The first point corresponds to 0.1% of the data being modified, the second point reflects performance when the initial 0.1% plus an additional 0.2% of data has been modified, and so on. The figure plots two lines: one for a constant join query with selectivity 0.1%, and the other for varying the selectivity of the join from 0.1% to 1%, thus corresponding to the number of alternatives being modified. As expected, we notice a nonlinear increase in execution time

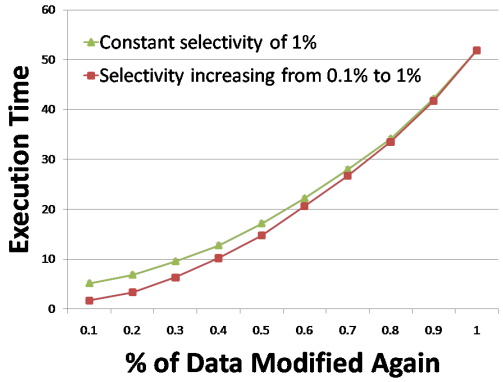


Fig. 6. Query execution time when the amount of modified data increases non-linearly.

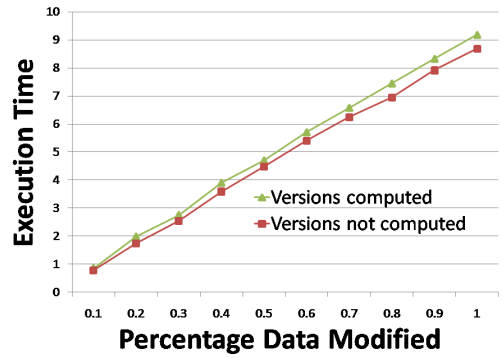


Fig. 7. Overhead of version computation in join query processing.

for both cases, because the total amount of data modified and hence the number of versioned alternatives, is increasing nonlinearly.

*3) Overhead of Version Computation:* In Figure 7 we study the overhead of computing version-intervals in query results, again when increasing fractions of data are modified. It turns out almost no overhead is induced by processing alternatives with both overlapping and non-overlapping versions, because we are able to push the version predicate into the query. In Figure 7, one plot refers to the execution time when version-intervals are computed on result tuples, and the other refers to the execution time to obtain the same result but without version computation. For the former plot, each alternative that goes into the join has been modified exactly once, while for the latter plot, the same number of alternatives were inserted (but not modified), so that version computation was not necessary.

Figure 8 depicts the overhead of version computation for different query operators, with 1% of the data being modified and queried in each case. As we can see, for all types of operators the overhead of version computation remains very small. Moreover, the plots show that the trend is similar for joins and other queries; hence our focus on join queries for the major part of our experiments.

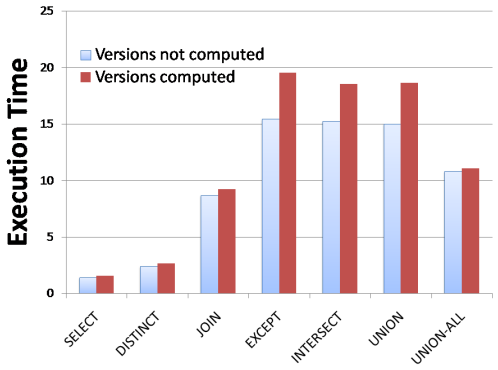


Fig. 8. Overhead of version computation for different operators.

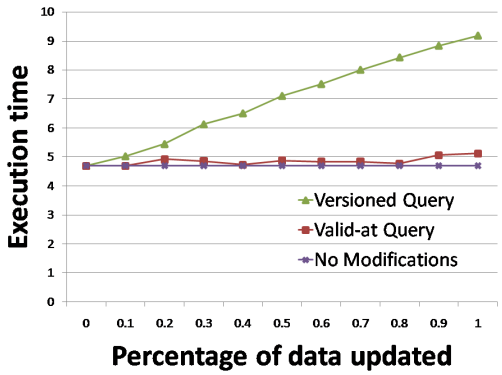


Fig. 9. Valid-at query execution time compared to versioned and non-versioned queries, for varying update selectivity.

*4) Versioned vs. Valid-at Queries:* Figures 9 compares the execution time of the versioned join query with selectivity 1%, with the corresponding valid-at query, which only selects data valid at the current version. The fraction of data modifications is varied from 0.1% to 1%. The execution time of the versioned query grows linearly with the amount of data modified. However, the execution time of the valid-at query remains almost the same as the execution time of the join query when no data is modified (marked by the horizontal line in Figure 9). Hence modifications and versioning in Trio adds little overhead when we want to query only the current (or any fixed) version: The execution time primarily depends on the amount of data valid at the specified version.

## VII. Related Work

While there has been a significant amount of work on uncertain databases in the past few decades (e.g., [3], [4], [20], [24], [27]), surprisingly little work has focussed on modifications, and none on versioning. Driven by new application needs, there has also been a flurry of more recent theoretical and practical work in managing uncertainty [6], [7], [10], [11], [13], [21], [33], [36], but none of this work we are aware of addresses data modifications or versioning either. To the best of our knowledge, ours is the first uncertain DBMS to support these capabilities.

We separately describe related work on data modifications

in uncertain databases, and versioning and view maintenance in traditional databases.

**Data Modifications:** Reference [2] considers various modification operations over incomplete databases, broadly defined as a transformation from one set of possible worlds to another. The paper studies a wide class of modifications, which could be incompletely specified, could insert or delete possible worlds, constrain the set of possible worlds, or apply updates preserving the set of possible worlds. The main result of the paper is to the study a set of data models for uncertainty in terms of their expressiveness in representing results of modifications. By contrast, our work consider the specific data modifications that can be expressed through Trio's SQL-like query language and defined over possible worlds of the ULDB model. It focuses not on the modifications themselves, but on efficient query answering, propagating modifications to derived relations, and implementing a working system that supports both modifications and versioning.

Reference [23] views the data modification problem as a programming language, and maps programs to operations on possible worlds (e.g., intersections, unions). While the language of [23] is again quite expressive, no query answering or propagation techniques are presented for any uncertain data model. Similarly, [25] considers approaches for modifying incomplete information, but doesn't consider the problems addressed by our paper. Some other work on model-based approaches for modifying incomplete databases represented by logical theories, e.g., [37] also does not consider propagation or systems issues.

**Traditional Databases:** A large body of work studies incremental view maintenance in traditional relational databases (refer to [22] for a survey), and also for other context such as data warehouses [26]. The basic premise of incremental view maintenance is to be able to modify a materialized view without recomputation, when the base data on which it depends is modified. While efficient incremental maintenance is always possible for certain classes of queries and modifications as surveyed by [22], a large body of previous work is devoted to exploiting additional information and features to enable incremental computation in other cases. For instance, reference [12] exploits information about keys in base and derived data, [31], [32] use pointers to base data and modify derived relations *lazily*, while [9] analyzes predicates in queries and properties of modified data to detect when derived relations are not affected by the modifications at all. We observed that lineage generalizes the kinds of information exploited by previous work, and thus enables applying similar propagation techniques for a wider class of queries and modifications.

Incorporating versions and notions of time in databases has been studied a great deal (refer to [17], [34]), and continues to be an emerging area with new systems such as [28] being built. Versioning in databases has also been used for other topics such as concurrency control [8]. However, it hasn't been applied in the context of databases with uncertainty and lineage. More importantly, our work is not to be confused as introducing a temporal aspect to Trio. As mentioned earlier, we are interested only in a *lightweight* versioning capability whose primary role is to support data modifications in an environment of derived relations and data lineage.

## VIII. Future Work

This paper provides a foundation for data modifications and versioning in a DBMS with uncertainty and lineage, supported by implementation in the Trio system. There are several major areas of future work:

- **Modifications to derived data:** We have not considered modifying derived relations in our system, which is the analogue of the traditional *view update problem* [18]. As with propagation, it is clear that lineage will play an important role, permitting modifications to a wide class of derived data, and propagating those modifications to base data efficiently.
- **No-propagation:** This paper addresses the case where all modifications to base data are propagated to derived relations. As discussed in the Introduction, applications may prefer not to propagate all modifications (for efficiency, usability, or both). Our versioning system provides a perfect basis for this case: Lineage of unmodified derived relations naturally points to old versions of base data. Through version-intervals, users can easily see which derived data is still valid, and which is no longer valid (and why). Fully developing this option requires additional foundations, algorithms, and implementation; it is an important area of future work.
- **Update lineage:** We plan to explore the idea of "update lineage," which connects newer versions of modified data to older versions. The benefits from a user perspective of efficiently navigating the update history of a data item are clear. We plan to explore the more general impact of update linage on semantics, query language, query processing, and implementation.

## Acknowledgement

This work was supported by the National Science Foundation under grants IIS-0324431 and IIS-0414762, by grants from the Boeing and Hewlett-Packard Corporations, and by a Microsoft Graduate Fellowship.

## References

[1] TriQL: The Trio query language. Available at http://i.stanford.edu/widom/triql.html.

[2] S. Abiteboul and G. Grahne. Update semantics for incomplete databases. In *VLDB*, 1985.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] S. Abiteboul, P. Kanellakis, and G. Grahne. On the Representation and Querying of Sets of Possible Worlds. *Theoretical Computer Science*, 78(1), 1991.

[5] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *Proc. of ICDE*, 2008.

[6] L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing Incomplete Information with Probabilistic World-Set Decompositions. In *Proc. of ICDE*, 2007.

[7] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *Proc. of VLDB*, 2006.
[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing, 1987.
[9] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.
[10] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *Proc. of ACM SIGMOD*, 2005.
[11] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. OLAP over uncertain and imprecise data. *J. VLDB*, 16(1):123–144, 2007.
[12] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of VLDB*, 1991.
[13] R. Cheng, S. Singh, and S. Prabhakar. U-DBMS: A database system for managing constantly-evolving data. In *Proc. of VLDB*, 2005.
[14] N. Dalvi and D. Suciu. Efficient Query Evaluation on Probabilistic Databases. In *Proc. of VLDB*, 2004.
[15] A. Das Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working Models for Uncertain Data. In *Proc. of ICDE*, 2006.
[16] A. Das Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *Proc. of ICDE*, 2008.
[17] C. Date and H. Darwen. *Temporal Data and the Relational Model*. Morgan Kaufmann Publishers, 2002.
[18] U. Dayal and P. A. Bernstein. On the updatability of relational views. In *VLDB*, 1978.
[19] M. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. In *SIGMOD Record*, pages 27–33, 2005.
[20] N. Fuhr and T. Rölleke. A Probabilistic NF2 Relational Algebra for Imprecision in Databases. *Unpublished Manuscript*, 1997.
[21] T. J. Green and V. Tannen. Models for incomplete and probabilistic information. In *Proc. of IIDB Workshop*, 2006.
[22] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2), 1995.
[23] S. J. Hegner. Specification and implementation of programs for updating incomplete information databases. In *PODS*, 1987.
[24] T. Imielinski and W. Lipski. Incomplete Information in Relational Databases. *Journal of the ACM*, 31(4), 1984.
[25] A. M. Keller and M. Winslett. Approaches for updating databases with incomplete information and nulls. In *Proc. of ICDE*, 1984.
[26] W. Labio, J. Yang, Y. Cui, H.-G. Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proc. of VLDB*, 2000.
[27] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. Subrahmanian. ProbView: A Flexible Probabilistic Database System. *ACM TODS*, 22(3), 1997.
[28] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: transaction time support for SQL server. In *Proc. of ACM SIGMOD*, 2005.
[29] M. Mutsuzaki, M. Theobald, A. 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 CIDR*, 2007. Demonstration description.
[30] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. of ICDE*, 2007.
[31] N. Roussopoulos. View indexing in relational databases. *TODS*, 7(2), 1982.
[32] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A testbed for incremental access methods. *IEEE TKDE*, 5(5), 1993.
[33] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In *Proc. of ICDE*, 2007.
[34] R. T. Snodgrass. *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers, 2000.
[35] T. P. C. (TPC). TPC Benchmark H: Standard Specification, 2006. `http://www.tpc.org/tpch`.
[36] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Proc. of CIDR*, 2005.
[37] M. Winslett. A model-based approach to updating databases with incomplete information. *ACM TODS*, 13(2), 1988.

## APPENDIX

### A. Proofs

**Proof of Theorem 3.5:** The proof of ULDB$^v$ completeness follows from the completeness of ULDBs [7]. For each $i$, $0 \leq i \leq v_D$, we construct a ULDB relation $D_i$ that represents the set of possible worlds $P_i$. Then, we construct ULDB$^v$ $D$ as follows. Construct each relation $R$ in $D$ by including all tuples from the corresponding relations in each $D_i$. Set the version-interval of each alternative from $D_i$, $i < v_D$, to $[i, i]$, and set the version-interval of each alternative from relation $D_{v_D}$ to $[v_D, \infty]$. The resulting database exactly represents the set of possible worlds $P_i$ for $0 \leq i \leq v_D$. □

**Proof of Theorem 4.5:**

- **PTIME:** The algorithm for version-interval computation presented is clearly PTIME. We prove that it correctly returns the version-interval when confidence values are known. Consider a DNF lineage formula with negations pushed down to literals, $\lambda$, of some alternative $a$ in a ULDB$^v$. We prove the equality of the computed version-interval $I_\lambda$ and the actual version-interval $I_{correct}$ in two steps:
  1) $I_{correct} \subseteq I_\lambda$: Let $v_0 \in I_{correct}$ be a version at which $a$ is present. Consider the restriction $\lambda_{v_0}$ of $\lambda$ obtained by setting to *false* every alternative in $\lambda$ whose version-interval does not contain $v_0$. Since $v_0 \in I_{correct}$, $\lambda_{v_0}$ is satisfiable. Given a satisfying assignment of $\lambda_{v_0}$, we can find a disjunct in the DNF formula $\lambda$ that is satisfied. The version-interval of this disjunct alone contains $v_0$. Since $I_\lambda$ is the union of the version-intervals of each of $\lambda$'s disjuncts, $v_0 \in I_\lambda$.
  2) $I_{correct} \supseteq I_\lambda$: Now suppose $v_0 \in I_\lambda$. We show that $\lambda_{v_0}$, obtained by setting to *false* every alternative in $\lambda$ whose version-interval does not contain $v_0$, is satisfiable. Since $v_0 \in I_\lambda$, $v_0$ is contained in the version-interval of at least one disjunct of $\lambda$. Every alternative in this disjunct whose version-interval contains $v_0$ can be set to true to satisfy $\lambda$ and hence satisfy $\lambda_{v_0}$.
- **NP-hard:** To correctly compute the version-interval for an alternative, we need to check for each input alternative, whether its confidence value is 1 at any version number. Detecting whether the confidence value of a boolean formula is 1 or not is NP-hard as it is equivalent to checking the satisfiability of its complement. □

### B. Data Modification Language

We provide formal semantics for each data modification language construct in terms of possible worlds. Consider a relation $R$ with possible worlds $\{R_1, \ldots, R_n\}$ with probabilities $p_1, \ldots, p_n$ respectively.

**Insert Tuple:** A tuple can be inserted into ULDB relation $R$ as in SQL:

"Insert Into $R$ Values <tuple>"

In the query above, <tuple> specifies the tuple to be inserted by listing alternatives and confidence values, or by a query. Suppose we insert a tuple with alternatives $a_1, \ldots, a_m$ having probabilities $p(a_1), \ldots, p(a_m)$ respectively. The resulting set of possible worlds of $R$ would be $\{R_1^1, \ldots, R_1^m, \ldots, R_n^1, \ldots, R_n^m\}$, where $R_i^j$ is the conventional relation that adds tuple $a_j$ to $R_i$, and the probability of $R_i^j$ is $p_i * p(a_j)$. If $\sum_j p(a_j) < 1$, then the original possible worlds of $R$, i.e. $R_1, \ldots, R_n$, still remain possible worlds, with probabilities $p_i * (1 - \sum_j p(a_j))$ respectively.[3]

**Insert Alternative:** An alternative can be inserted into specific tuple(s) of $R$ as follows:

"Update $R$ AltInsert <alt>:<conf> {Where <predicate>}"

The above query translates to a set of primitive inserts of <alt> with confidence <conf> into each tuple that has some alternative satisfying <predicate>. The insert query is legal only if the sum of confidence values of all alternatives in each tuple in which <alt> is being inserted is at most $(1 - \texttt{conf})$. Suppose <alt> is inserted into a tuple $t$ in $R$, which originally had the sum of confidence values of all alternatives equal to $s$. For each such insert, the set of possible worlds of $R$ are modified as follows: Each original possible world $R_i$ of $R$, with probability $p$, that did not include any alternative of $t$ is replaced with two possible worlds: $R_i^1 = R_i$ with probability $\frac{(1-s-conf)*p_i}{(1-s)}$, and $R_i^2$ which adds alternative <alt> to $R_i$ and has probability $\frac{conf*p_i}{(1-s)}$.

**Delete:** Alternatives can be deleted from $R$ using the standard SQL syntax:

"Delete From $R$ {Where <predicate>}"

The above query translates to a set of primitive delete operations on each alternative in $R$ that satisfies <predicate>. When all alternatives of a tuple are deleted, the tuple gets deleted from $R$. Suppose we delete an alternative $a$ from $R$, the resulting set of possible worlds for the new $R$ would be $\{R_1', \ldots, R_n'\}$ where $R_i'$ is obtained by deleting that alternative $a$ from $R_i$ (if $a$ appears in $R_i$). The probability of $R_i'$ remains $p_i$. By definition, if removing alternatives from each possible world results in multiple possible worlds becoming equal, then their probabilities are added up.

**Update Alternative:** Alternative values can be updated in $R$ as follows:

"Update $R$ Set <attr-list>=<expression-list>
{Where <predicate>}"

The statement above updates every alternative of $R$ satisfying <predicate>. Each attribute in <attr-list> is updated with the result of the corresponding expression on the right-hand side of the =, just as in SQL. $R$'s possible worlds are modified by replacing each old alternative value in every possible world with the corresponding new alternative value.

**Update Confidences:** Finally, confidence values in $R$ can be modified as follows:[4]

"Update $R$ Set conf=<expression> {Where <predicate>}"

The above query modifies the confidence value of every alternative satisfying <predicate> based on <expression>. The modification is legal only if the sum of confidence values of all alternatives in each resulting tuple is at most 1. Modifying the confidence values of alternatives does not change the set of possible worlds of $R$, but only changes their probabilities, which can be recomputed based on techniques from a previous paper [16]. Since modification of confidence values does not raise any additional challenges in versioning and query answering, we focus on only data modifications in the rest of the paper.

### C. Propagating Deletions

We first consider propagating deletions on base relations to derived relations. Recall that when an alternative is deleted from a relation $R$, the only change made is to change its end version to the current version $v_D$.

*1) Positive Lineage:* Propagating deletes for any query result which contains only positive lineage can be done very easily. (Recall combinations of all of select, project, join, union, and duplicate-elimination generate only positive lineage.) Suppose $T$ is a derived relation containing only positive lineage. When some alternatives from $T$'s input relations are deleted, we only need to recompute the version intervals of all alternatives in $T$ that depend on the deleted data. Since deletions don't change any base data, data in $T$ is also unaffected. Therefore, only recomputation of version intervals, which we know from Section IV is very inexpensive, gives us the correct modified $T$.

*2) EXCEPT:* Now consider derived relation $T$ obtained by executing "$R_1$ EXCEPT $R_2$", where $R_1$ and $R_2$ are base relations. Deleted alternatives in $R_1$ are again propagated to $T$ by recomputing the version intervals of affected alternatives in $T$. The only case when recomputation of version intervals in $T$ doesn't give the correct answer is when we delete alternatives from $R_2$: Suppose there's an alternative $a$ in $R_1$ that also appears in $R_2$ with version interval $[0, \infty]$ with confidence 1.0. Then when $R_1 - R_2$ is executed, since the alternative $a$ is certainly present in $R_2$, it does not appear in the result relation. However, if the alternative gets deleted from $R_2$, then $a$ needs to be included in the result.

If a set $S$ of alternatives is deleted from $R_2$, we modify $T$ as follows. For each distinct alternative value $a$ in $S$, check whether $T$ contains alternatives with value $a$:

1. If yes, recompute the version intervals of the alternatives in $T$ with value $a$.

[3]Note that tuples with sum $s$ of confidence values for all alternatives $< 1$ can be treated as tuples with an additional "empty" alternative with the residual probability $(1 - s)$. Hence, whenever obvious from the context, we do not specially distinguish tuples with $s = 1$ from tuples with $s < 1$.

[4]Trio also allows tables which have uncertainty but no confidence values. In such tables, we can add or delete "?"s, which denote absence of tuples; "?"s are a special case of specific confidence values, and hence not considered separately in this paper.

2. If not, check whether any alternative in $R_1$ has the value $a$. For each such alternative $a_1$ from tuple $t_1$ in $R_1$:
   (a) If $T$ contains any tuple with lineage referring to alternatives of $t_1$, insert the alternative $a_1$ into it. If not, add a new tuple in $T$ containing $a_1$.
   (b) Let $s_1, \ldots, s_k$ be all alternatives in $R_2$ with the same value as $a_1$. Set the lineage of the newly added alternative in $T$ to $(a_1 \wedge \neg s_1 \wedge \ldots \wedge \neg s_k)$, and compute its version interval accordingly.

Effectively, we only need to insert alternatives into $T$ that were not included because $R_2$ contained certain alternatives with the same value. For the remaining alternatives in $T$, if they refer to modified alternatives, we recompute their version intervals; and if not, they are unchanged.

### D. Propagating Insertions

Next consider propagating insertions to derived relations. We are able to leverage traditional incremental view maintenance techniques [22] and lineage in ULDB$^v$s. Due to space constraints we provide brief algorithms for certain cases that illustrate the use of lineage. We refer the reader to Appendix **??** of the full version of the paper for complete details.

*1) Join:* Consider a relation $T$ derived by a join query $Q$ over $R_1$ and $R_2$, and let sets $S_1$ and $S_2$ of alternatives be inserted into $R_1$ and $R_2$ respectively. Our query-answering algorithm ensures that for any pair of tuples $r_1, r_2$ from $R_1$ and $R_2$ respectively, there exists at most one tuple $t$ in $T$ whose alternatives have lineage referring to alternatives of both $r_1$ and $r_2$. When the set $S_1$ of alternatives are inserted in $R_1$, $T$ is modified as follows. For each inserted alternative $a$ into a tuple $r_1$:

1. For every alternative $a'$, say in tuple $r_2$, from $R_2$, do:
   (a) Evaluate the join query $Q$ over the pair of alternatives $a$ and $a'$. If the result is empty, $T$ isn't modified. If not, let the resulting alternative be $Q(a, a')$, whose lineage refers to alternatives $a$ and $a'$.
   (b) If there's any tuple $t$ in $T$ whose alternatives have lineage to alternatives in $r_1$ and $r_2$:
      i. Then, add $Q(a, a')$ into $t$.
      ii. If not, create a new tuple in $T$ with the alternative $Q(a, a')$.

Although the procedure above describes insertions of alternatives individually for each pair of alternatives from $R_1$ and $R_2$, Trio's encoding of ULDB relations into ordinary relations [29] allows us to combine the operations into a single SQL query over the relational encodings of $R_1$ and $R_2$. Once we've inserted into $T$ all alternatives due to the set $S_1$, we then use the same procedure to insert alternatives due to the set of alternatives $S_2$ added into $R_2$; in this step we consider the already modified $R_1$. Effectively, we re-evaluate the join for all and only the new pairs of alternatives due to the alternatives inserted into $R_1$ and $R_2$.

*2) DISTINCT:* Now suppose $T$ was obtained by a query containing a "`Select DISTINCT`" clause. Given a set of insertions (of tuples or alternatives) in $T$'s input relations, we first modify $T$ disregarding the duplicate-elimination. Then we eliminate duplicate alternative values from the modified $T$ as follows. For each distinct alternative value $s \in S$ that also appeared in some alternative in $T$ originally, do the following:

1. Let the alternative $a$ in tuple $t$ of $T$ have the value $s$. Split $t$ into two tuples: $t_1$, containing the original tuple but without $a$, and $t_2$ containing only alternative $a$. (The lineage of each alternative ensures that splitting the tuple doesn't alter the possible worlds [7].)
2. Let $s_1, \ldots, s_m$ be the set of all alternatives in $S$ having the value $s$. Let $L$ be the original lineage $\lambda(a)$. Modify the lineage of $a$ (now in $t_2$) to: $\lambda(a) = (L \vee \lambda(s_1) \vee \ldots \vee \lambda(s_m))$.
3. Recompute the version interval of $a$ using lineage.

The procedure above effectively does a DISTINCT operation on the modified $T$, but only considering tuples in $T$ that have duplicate alternative values.

*3) EXCEPT:* Finally suppose $T$ is obtained from the query "$R_1$ EXCEPT $R_2$." First let us consider a set $S$ of alternatives inserted (in existing or new tuples) into $R_1$. Intuitively we shall obtain the modified $T$ by adding the result of "$S$ EXCEPT $R_2$" into $T$. First we insert each alternative in $S$ into $T$: each alternative is added into the corresponding tuple in $T$, and new tuples in $S$ are added as is into $T$. Then, we set the lineage of each newly added alternative $a$ as follows: Find all alternatives $s_1, \ldots, s_k$ in $R_2$ with the same value as $a$. Set the lineage of $a$ to $\lambda(a) = (s_a \wedge \neg s_1 \wedge \ldots \wedge \neg s_k)$, where $s_a$ is the alternative in $R_1$ corresponding to $a$. The version interval of each alternative in $T$ is then computed as described in Section IV.

Next let us consider a set $S$ of alternatives inserted (in existing or new tuples) into $R_2$. These modifications are propagated simply by modifying the lineage of all alternatives in $T$ with their attribute values coinciding with that of some alternative in $T$. For each such alternative $a$ in $T$, we find all alternatives $s_1, \ldots, s_k$ in $S$ with the same value and add the conjunct $(\neg s_1 \wedge \ldots \wedge \neg s_k)$ to $\lambda(a)$. We then recompute the version interval of all alternatives whose lineage is modified.

### E. Propagating Updates

Finally, recall that a primitive update is executed by expiring the old alternative by modifying its end version, followed by inserting the new alternative. Hence, we note that updates to base relations are propagated exactly using the techniques for propagating deletions and insertions described above.