LIVE: A Lineage-Supported Versioned DBMS

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

***Abstract*— This paper presents LIVE, a complete DBMS designed for applications with many stored derived relations, and with a need for simple versioning capabilities when base data is modified. Target applications include, for example, scientific data management and data integration. A key feature of LIVE is the use of *lineage* (provenance) to support modifications and versioning in this environment. In our system, lineage significantly facilitates both: (1) efficient propagation of modifications from base to derived data; and (2) efficient execution of a wide class of queries over versioned, derived data. LIVE is fully implemented; detailed experimental results are presented that validate our techniques.**

I. Introduction

Motivated by data integration, information extraction, scientific data management, and other applications that require storing and processing both base and derived data, we have developed LIVE, a new kind of DBMS. LIVE incorporates a *lightweight versioning capability* and *lineage tracking* in an environment of base and derived relations. There has been considerable past work on data modifications and incremental view maintenance (refer to [1] for a survey) as well as on support for versioning and temporal aspects in databases (refer to [2], [3]). LIVE distinguishes itself from this previous work in the following ways:

- Derived data maintenance: Lineage in LIVE generalizes various kinds of information (such as keys [4], pointers [5], [6], and predicate properties [7]) traditionally exploited for incremental view maintenance. Thus, LIVE supports the best possible incremental view maintenance algorithms for a wide class of views and modifications.
- Versioning: LIVE provides a lightweight versioning system primarily to support data modifications in our motivating applications. LIVE's versioning system works together with lineage to enable efficient query processing over derived, versioned relations. In the future, LIVE's versioning system together with lineage will also support a seamless combination of propagating and nonpropagating base data modifications; in the nonpropagating case, the lineage of "current" derived data may identify "old" base data. This feature is of particular use in scientific applications.
- Probabilistic data: Some of our motivating applications require managing *probabilistic* (*uncertain*) data. Probabilistic data may arise as a result of imprecise information extraction, for example, or because of uncertain mappings in data integration. LIVE supports probabilistic data without additional mechanisms. Thus, all of the the lineage, versioning, and query processing capabilities of LIVE can be applied to probabilistic as well as conventional data.

LIVE is an offshoot of the *Trio* project at Stanford [8], which is a system for managing data, uncertainty, and lineage. The specific goal of LIVE is to support modifications and versioning in an environment of stored, derived relations. As we shall see, adopting Trio's lineage functionality significantly eases propagating modifications and answering queries in this environment.

We note that *SciDB* [9], a recent proposal outlining the requirements of a general-purpose DBMS for scientific applications, includes derived data, lineage, versioning, and probabilistic data as key features. Similarly, a recent paper [10] laying out challenges in data integration includes derived data, lineage, and probabilities. We believe LIVE's overall combination of capabilities, and how they work together seamlessly and efficiently, is an important step towards supporting these application areas.

In the following, we briefly describe the main contributions made by this paper. Related work is discussed at the end in Section VII, and we conclude with future work in Section VIII.

Unified Data Model (Section II)

LIVE is based on a unified data model, LDM, which is derived from Trio's ULDB data model [11]. LDM incorporates base and derived relations, probabilistic data, lineage, and versioning. As is typical [12], LDM captures probabilistic data by attaching a *probability* (or *confidence*) in the range [0,1] to each tuple, indicating the likelihood of that tuple being present. As queries are performed, LIVE creates the lineage of derived data items, connecting them to the input data from which they were derived. In the standard versioning fashion, each LDM tuple includes a *start version number* and an *end version number*, between which the tuple is "valid."

Modifications and Versions (Section III)

We extend the typical relational modification primitives (insert, delete, and update of tuples) to include modifications of tuple probabilities. We then formalize all of the modifications in the presence of probabilities, based on the accepted *possible-worlds* [13] semantics of probabilistic databases. Finally, we specify how the primitive modification operations create versioned relations. Again, our goal is to incorporate lightweight versioning capabilities whose main function is to support meaningful data modifications in an environment of base and derived relations. We do not support historical

modifications, for example; modifications are always applied to the "current version" of the database.

Query Processing (Section IV)

Also in the interest of supporting only a lightweight versioning capability, we do not support rich constructs for referencing versions or histories. Queries over an LDM database are expressed as regular queries, and they produce a *versioned answer*: the result is an LDM 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 more efficient to compute. Furthermore, a user can view the *snapshot* of any relation (base or derived) as of any version.

We specify the semantics of query answering in LDM and present detailed algorithms. We will see that the LDM model introduces some subtleties with respect to defining lineage and probabilities on query results, especially in the presence of negation. Determining start and end versions on query results adds little overhead: LIVE's lineage feature enables a clean algorithm to compute versions on result tuples efficiently. In the special case of negation, for which we prove intractability of computing exact version-intervals, lineage enables an efficient approximation.

Propagating Modifications (Section V)

Next 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. We shall see that lineage provides us with critical information enabling efficient incremental view maintenance techniques for a wide class of modifications and derived relations.

System and Experiments (Section VI)

LIVE is fully implemented, including the entire suite of data modification and versioning capabilities described in this paper. We believe ours is the first DBMS that incorporates data modifications and a lightweight versioning system for derived relations over ordinary and probbilistic data. Moreover, the lineage feature of LIVE enables elegant algorithms for propagating modifications and for query processing. We describe the implementation of LIVE briefly and include a number of performance results.

II. LIVE Data Model (LDM)

In this section we describe LDM, the data model on which LIVE is built. We begin by introducing how lineage is represented and created in LIVE (Section II-A), then we define LIVE's probabilistic data (Section II-B), and finally we add versioning (Section II-C). Many of these features are similar to previous work in their respective areas and not particularly novel, but they are a necessary basis for the system and for the new contributions in the remainder of the paper. Our model for lineage is adopted from Trio [11], [14], so it is only briefly reviewed here.

A. Lineage

Whenever a derived relation is created in LIVE lineage is tracked and stored automatically. Lineage connects derived tuples to the (base or derived) tuples from which they were derived.

Suppose we have two relations: `People(Name,State,Job)` lists people's state and occupation, while `Photo(Num,Name)` identifies the names of people appearing in numbered photos. (We call them `P(Name,S,J)` and `Ph(N,Name)` for short.). We begin with the following sample data, including tuple identifiers (denoted ID):

ID	**Ph(N, Name)**
11	(1,Amy)
12	(1,Bob)
13	(1,Carl)
14	(1,Dale)

ID	**P(Name, S, J)**
21	(Amy,CA,Engineer)
22	(Bob,NY,Analyst)
23	(Carl,IL,Teacher)
24	(David,PA,Manager)

Suppose we join `Photo` and `People`, then project attributes `Number` and `State`. We obtain derived relation `States(Number,State)`, listing the states of the people appearing in photos:

ID	**States(Num, State)**	
31	(1,CA)	$\lambda(31) = 11 \wedge 21$
32	(1,NY)	$\lambda(32) = 12 \wedge 22$
33	(1,IL)	$\lambda(33) = 13 \wedge 23$

Lineage, denoted by λ, is shown alongside each tuple. In general, the lineage of a tuple in a derived relation is a boolean formula over other tuple identifiers. Intuitively, a derived tuple exists because the existence of the tuples in its lineage formula make the formula evaluate to true. Joins generate conjunctions as seen in this example, while other constructs (e.g., duplicate elimination) generate disjunctions, and yet others (difference) generate negation. We will see shortly that lineage is also closely tied with the semantics of derived probabilistic data.

B. Probabilities

Each tuple in an LDM relation may optionally be annotated with a value in the range $[0, 1]$, denoting the probability that the tuple is present in the relation. (The absence of a probability is equivalent to probability 1.) We adopt the standard *possible-worlds* interpretation for probabilistic databases [15], [11], [16], [17], [18]: The possible-worlds of each base relation R consist of omitting the tuples in R that have probability < 1 in all possible combinations. Restricting ourselves to base relations, the possible-worlds of a database combine the possible-worlds of the base relations in all possible ways. Each possible-world has a probability associated with it. The probability of a possible-world is given by the product of the probabilities p_i of each tuple t_i present in the world, and $(1 - p_i)$ for each tuple omitted in the world. The possible-world probabilities are guaranteed to sum to 1.

When a query Q is executed on an LDM database D with probabilities, the result is an LDM relation R, including lineage, whose possible-worlds represent the result of applying Q to each possible-world of D. When R is added to D as a

derived relation, lineage is created as described in Section II-A. This semantics for derived data with lineage extends the possible-worlds model in a natural way: Possible-worlds for a derived relation are only combined with those possible-worlds for base relations such that the following property holds. A derived tuple is present in a possible-world if and only if its "transitive" lineage formula, i.e., expanded to refer to base tuples only, evaluates to *true* based on the presence (*true*) and absence (*false*) of the referenced base tuples. Note in particular that derived relations do not increase the number of possible-worlds for a database.

A very nice property observed elsewhere [11] is that the probability associated with a derived tuple t is the probability of its transitive lineage formula computed using the given probabilities for the base tuples. We will soon see that version intervals for derived tuples also can be computed through lineage, with an even more efficient algorithm.

Example 2.1: Let us add probabilities to the `Photo` relation (indicating uncertainty in identification), but suppose all tuples in the `People` relation still have a probability 1.0.

ID	**Ph(N, Name)**
11	`(1,Amy):1.0`
12	`(1,Bob):0.6`
13	`(1,Carl):0.3`
14	`(1,Dale):0.1`

The data and lineage in our join result `States` remains the same. To compute the probability of one of its tuples, say 32, we evaluat the probability of its lineage formula $\lambda(32) = 12 \wedge 22$; $\Pr(32) = \Pr(12 \wedge 22) = 1 * 0.6$. □

C. Versioning

Next we add versions to our data model. Most parts of the versioning model are standard, but a few subtleties do arise due to lineage and probabilities.

Each tuple in an LDM relation now includes a *start version* and an *end version*, which are non-negative integers or ∞. Suppose a tuple t has start version s and end version e, denoted by the interval $[s, e]$. Then, intuitively, the tuple is *valid* for all versions v such that $s \leq v \leq e$. We use start(t) and end(t) to denote the start and end versions of tuple t.

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

We defer until Section III a formal definition of how precisely modifications create versioned relations. Section IV covers (among other topics) more details on how versions interact with derived relations and lineage. In the remainder of this section we define the *snapshot* of an LDM relation, and we use snapshots to define formally the semantics of versioned LDM relations in terms of possible-worlds.

Definition 2.2 (Snapshot): Given an LDM relation R whose current version is v_D, the *snapshot* of R at version $v \leq v_D$ (denoted $R_{@v}$) is a non-versioned LDM relation obtained by eliminating from R all tuples t such that $v <$ start(t) or $v >$ end(t). In the snapshot of a derived relation, the lineage of a tuple t' is obtained by replacing every tuple identifier i in $\lambda(t')$ with *false* if i's version-interval does not include v. If now $\lambda(t') \equiv$ *false*, t' is not included in the snapshot. The probability of t' in the snapshot is determined by its lineage in the snapshot in the standard fashion. □

A valid LDM relation R must satisfy some basic properties: Every tuple t in R must have start$(a) \leq v_D$ (recall v_D denotes the current version), and if end$(a) \geq v_D$ then end$(a) = \infty$. Intuitively, all tuples must have come into existence at or before the current version, and any tuple that is valid at the current version is valid at and beyond the current version; i.e., no tuple could already have been deleted after the current version. We then have the following straightforward result, which says that the database is constant from version v_D onwards.

Theorem 2.3: Given an LDM database 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 probabilistic databases encode a set of possible-worlds, LDM databases with versions encode a list of sets of possible-worlds: one set for each version $v \geq 0$.

Definition 2.4 (Possible-Worlds): Given an LDM database 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 2.5: Consider the relation `Photo` from our running example extended with versioning:

ID	**Ph(N, Name)**[2]
11	`(1,Amy)`$^{[0,1]}$`:1.0`
12	`(1,Bob)`$^{[0,\infty]}$`:0.6`
13	`(1,Carl)`$^{[0,0]}$`:0.3`
14	`(1,Dale)`$^{[1,1]}$`:0.1`

Version-intervals for each tuple are marked as superscripts. Suppose the current version of the database is $v_D = 2$. For example, the tuple `(1,Amy)` is valid for versions 0 and 1, and the only tuple valid in the current version is `(1,Bob)`. The snapshot of `Photo` at version 1 is shown below. (Tuple identifiers are omitted from the snapshot.)

Ph$_{@1}$(N, Name)
`(1,Amy):1.0`
`(1,Bob):0.6`
`(1,Dale):0.1`

□

Just as the question of *completeness* arises in the theory of non-versioned probabilistic databases [15], [19], [20], [21], [22], the corresponding question arises here: whether LDMis *complete for versioned probabilistic databases*. That is, given some v_D and a set of possible-worlds P_i for $0 \leq i \leq v_D$, is there an LDM database 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 2.6: LDM is complete for versioned probabilistic databases. □

A proof for this theorem appears in the Appendix.

III. Modifications

Recall that LIVE supports modification of the current version of the database and not historical modifications. In this section, we define LIVE's set of primitive modification operations. We provide a formal semantics based on possible-worlds, and we specify how the modification operations create versioned LDM relations.

In this paper we assume modifications are allowed on base relations only, then are propagated to all derived relations using the algorithms to be presented in Section V. We have some preliminary algorithms and results, not presented in this paper, on lineage-enabled propagation of modifications from derived relations to their input relations. Also, as discussed in Section I, our versioning system provides a basis for allowing derived and base relations to "diverge" after modifications, but retaining meaningful connections through lineage involving old versions. Neither of these features is implemented yet in LIVE, but they are an important direction of future work.

LIVE supports a declarative SQL-based language for modifying relations. As in SQL, a statement in this language results in a set of primitive modification operations executed by the system. In LIVE the primitive modifications are:

- insert a new tuple
- delete an existing tuple
- update the value of one or more attributes in an existing tuple
- update the probability of an existing tuple

A. Semantics

We define the semantics of each of the primitive modification operations in terms of their effect on the set of possible-worlds. Consider a relation R with possible-worlds $\{R_1, \ldots, R_n\}$. Let possible-world R_i have probability p_i, $i = 1..n$.

Insert tuple: Suppose we insert into R a tuple t with probability $p(t)$. The resulting set of R's possible-worlds is $\{R_1^1, R_1^2, R_2^1, R_2^2, \ldots, R_n^1, R_n^2\}$, where for $i = 1 \ldots n$: R_i^1 adds tuple t to R_i and has probability $p_i * p(t)$; $R_i^2 = R_i$ and has probability $p_i * (1 - p(t))$. If the new set of possible-worlds contains any duplicate worlds (i.e., all certain relations are identical), then their probabilities are summed and one world is retained.

Delete tuple: Suppose we delete a tuple t from R. The resulting set of R's possible-worlds for the new R is $\{R'_1, \ldots, R'_n\}$, where each R'_i deletes t from R_i if it is present. The probabilities remain unchanged, however when tuples are deleted from possible-worlds, duplicate worlds are always created. As with insertion, duplicate worlds are merged and probabilities added.

Update value: When one or more values in an existing tuple are updated, R's possible-worlds are modified by replacing the old tuple in every possible-world with the corresponding new tuple. Probabilities remain unchanged. Duplicate worlds, if present, are merged.

Update probability: Modifying the probability of a tuple does not change the set of possible-worlds of R, but only changes their probabilities, following the semantics of LDMs.

B. Execution

Now consider execution of modifications. 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. (Note that 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 an LDM relation R whose current version is v_D.

1. **Insert Tuple:** When a tuple t is inserted into R, t is assigned the version-interval $[v_D + 1, \infty]$.
2. **Delete Tuple:** When a tuple t is deleted from R, it is retained in R as is, except end(t) is modified to be v_D.
3. **Update Value:** When a tuple t is updated to t', it is treated as a deletion of t and an insertion of t'. That is, end(t) is modified to be v_D, and tuple t' is inserted with version-interval $[v_D + 1, \infty]$.
4. **Update Probability:** When the probability p of a tuple t is modified to be p', it is treated as a deletion of t with probability p and an insertion of $t' = t$ with probability p'. That is, end(t) is modified to be v_D, and tuple $t' = t$ is inserted with probability p' and version-interval $[v_D + 1, \infty]$.

Example 3.1: Consider the `People` relation from our running example, but now with probabilities, versions, and slightly modified data. 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)`$^{[0,\infty]}$`:1.0`
23	`(Carl,IL,Teacher)`$^{[0,\infty]}$`:1.0`
24	`(David,PA,Manager)`$^{[0,\infty]}$`:0.6`

Suppose we apply the following modifications in sequence, committing after each operation: (1) With probability 0.3 engineer Frank from CA is now known, so a tuple inserted. (2) We decide that David is possibly (30% chance) a CEO as well, so a tuple is inserted. (3) Carl retires, so his tuple is deleted. (4) Bob decides to go to graduate school in California, so his tuple is updated, changing his job from Analyst to Student and his state from NY to CA. We show below the `People` relation after all the modifications have been applied, marking

the modification alongside affected tuples. The final database version is $v_D = 4$.

ID	People(Name, State, Job)[4]	
22	(Bob,NY,Analyst) $^{[0,3]}$:1.0	← (4)
23	(Carl,IL,Teacher) $^{[0,2]}$:1.0	← (3)
24	(David,PA,Manager) $^{[0,\infty]}$:0.6	← (2)
25	(Frank,CA,Eng.) $^{[1,\infty]}$:0.3	← (1)
26	(David,PA,CEO) $^{[2,\infty]}$:0.3	← (2)
27	(Bob,CA,Student) $^{[4,\infty]}$:1.0	← (4)

□

IV. Query Evaluation

We now address the problem of query evaluation in LDM. In query results, we are now interested not only in the data, probabilities, and lineage, but also in the version-intervals of each result tuple.

Definition 4.1 (Query Answer): Given an LDM database 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 an LDM relation R whose tuples have lineage to tuples in $R_1, \ldots, R_n$. The resulting LDM 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 versioned 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. (Recall from Section II-B that, further, the result of a query over a set of possible-worlds is defined as logically applying the query to each possible-world.)

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 probability values in query results over LDMs.

A. Version-Interval Computation

In Section IV-A.1 we address query evaluation for queries that generate "positive" lineage. We 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 tuple 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 LDM relations without versions; i.e., we disregard the version-intervals of tuples in this stage. The tuples in S and their lineage can be computed using the query processing algorithms in [11].
2. Version Computation: We use the lineage of each tuple 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 a tuple 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 LDMs 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, whose proof follows directly from the discussion below, describes how this set of all versions can be computed.

Theorem 4.2 (Version Interval Computation): Consider a tuple a in a result relation, with lineage $\lambda(a)$ referring to tuples $a_1, \ldots, a_m$ 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 tuple 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 a tuple a with positive lineage $\lambda(a)$ can be computed in PTIME in the number of tuples in $\lambda(a)$. □

The above theorem translates the boolean lineage formula into an expression over version-intervals of tuples, such that the result of the new expression gives the version-interval of the resulting tuple. 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.

Example 4.4: Consider Photo and People modified once again and omitting probabilities for this example:

ID	Photo(Number, Name)
12	(1,Bob) $^{[2,\infty]}$
13	(1,Carl) $^{[4,6]}$
14	(2,Frank) $^{[0,1]}$

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

As before, suppose States(Number,State) is obtained by joining the two relations. After Step 1 (data computation),

we have the following result tuples and lineage:

ID	States(Number, State)	
41	(1,NY)	$\lambda(41) = 12 \wedge 22$
42	(1,IL)	$\lambda(42) = 13 \wedge 23$
43	(2,CA)	$\lambda(43) = 14 \wedge 25$
44	(1,CA)	$\lambda(44) = 12 \wedge 27$

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

ID	States(Number, State)	
41	(1,NY) $^{[2,3]}$	$\lambda(41) = 12 \wedge 22$
43	(2,CA) $^{[1,1]}$	$\lambda(43) = 14 \wedge 25$
44	(1,CA) $^{[4,\infty]}$	$\lambda(44) = 12 \wedge 27$

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

ID	Region(State)	
51	(NY) $^{[2,3]}$	$\lambda(51) = 41$
52	(CA) $^{[1,1] \cup [4,\infty]}$	$\lambda(52) = 43 \vee 44$

As illustrated by Tuple 52 in our example, 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 encoded in LIVE by creating multiple tuples, each with a single interval. □

2) Arbitrary Lineage: Now we consider computing version-intervals of result relations with arbitrary lineage. Specifically, the lineage of a tuple 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 LIVE 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 a tuple a_j, whose probability 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 tuple is computed as before. If we know a_j has probability 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.

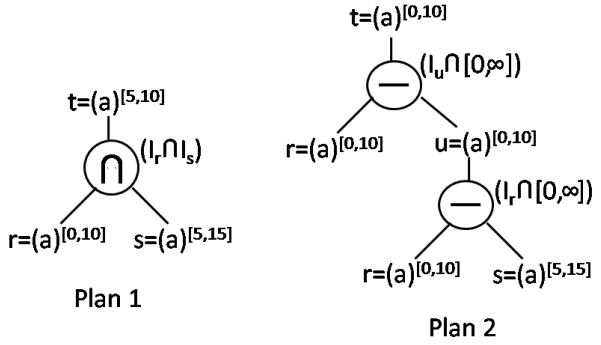

Fig. 1. Pushing version-interval computation into query plans with negation.

However, the following theorem shows that the hardness of probability computation in probabilistic databases [16] makes exact version-interval computation also intractable. Further, it shows that the algorithm described above correctly computes version-intervals in PTIME when probabilities are known. The proof of this theorem appears in the Appendix.

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

- PTIME when all a_i's probabilities are known.
- NP-hard when probabilities 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 probabilities are known then our algorithm is still linear. If probabilities 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-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 tuples, and the `join` operator intersects the version-intervals of the input tuples. If the version-interval of an intermediate tuple in the query plan is empty (for instance, as a result of a join), the tuple 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 1 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.

LIVE pushes version-interval computation into the query plan whenever possible. We have also built suitable indexes on start and end version columns to enable efficiently combining and intersecting version-intervals in the query plan.

B. Filtering based on Versions

LIVE provides two constructs that enable filtering data based on their version.

1. Valid-At Queries: The clause "`valid at <version-number>`" can be added to any regular LIVE query, to filter out input tuples not valid at a particular version. For example, "`Select * from People valid at 3`" selects from `People` all tuples 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 LDM relation.

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

C. Probabilities

An important subtlety arises when combining derived relations, probabilities, and versioning. The subtlety occurs only in query results with disjunctive lineage, typically due to duplicate elimination. While a tuple can be present in multiple versions of data, its probability may vary across versions.

A simple example illustrates the issue. Consider a modified `States` base relation.

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

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

All information needed to determine probabilities for any version is included in result lineage, but there is a question of presentation and clarity. One possibility would be to split tuples based on probabilities, remembering somehow that duplicate-elimination has been performed so the result is a set. For example, in the query result above we could have two result tuples with value `[1]`, the first having version-interval $[0, 3]$ and probability 1.0, the second having version-interval $[4, \infty]$ and probability 0.5. LIVE does not use this approach. Rather, LIVE displays the probability for the current version v_D as default, and historical probability values can be shown on request.

V. Propagating Modifications

Next we address the problem of propagating the modifications on base data described in Section III 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 LDM 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 LIVE 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 [1]) 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 [7], [4], [5], [6] 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 LDMs 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 a tuple a from a relation R, we simply set 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 tuples from T's input relations are deleted, we simply recompute the version-intervals of all tuples in T whose lineage refers to deleted tuples. Lineage allows us to easily determine which tuples in T refer to deleted tuples.

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 tuples in R_1 are again propagated to T by recomputing the version-intervals of affected tuples in T. The only case when recomputation of version-intervals in T doesn't give the correct answer is when we delete tuples from R_2: Suppose there's a tuple a in R_1 that also appears in R_2 with version-interval $[0, \infty]$ and probability 1.0. Then when $R_1 - R_2$ is executed, since the tuple a is certainly present in R_2, it does not appear in the result relation. However, if the tuple gets deleted from R_2, then a needs to be included in the result.

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

1. If yes, recompute the version-intervals of the tuples in T with value a.
2. If not check whether any tuple in R_1 has the value a. For each such tuple a_1 in R_1:
   (a) Insert a new tuple a_1 in T.
   (b) Let $s_1, \ldots, s_k$ be all tuples in R_2 with the same value as a_1. Set the lineage of the newly added tuple 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 tuples into T that were not included because R_2 contained certain tuples with the same value. For the remaining tuples in T, if they refer to modified tuples, we recompute their version-intervals; and if not, they are unchanged. Once again, the task of determining which tuples in T refer to modified tuples is made simple by lineage.

VI. System and Experiments

A. System Description and Setup

Like Trio [8], LIVE is layered on top of a conventional DBMS. It encodes LDM relations in standard relational tables, and lineage of each derived LDM relation is materialized into a separate table. In addition to data columns, each tuple in the encoding has a column for the probability, two system columns—`start` and `end`—corresponding to start and end versions respectively, and some extra columns to encode lineage. (Details of the exact encoding are omitted due to space constraints.)

LIVE 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 LDM relation. Lineage and version-intervals (but not probabilities) 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 overhead of modifications and version management in LIVE and its effects on query processing, compared to a non-versioned LIVE implementation.

Our dataset is based on a synthetically created set of TPC-H [23] 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 probabilistic, we independently and randomly assigned a probability in $[0, 1]$ to each tuple. In our dataset, `Lineitem` contained 6,000 (at a selectivity of 0.1%) to 60,000 (at a selectivity of 1%) tuples, and `Orders` contained between 1,500 and 15,000 tuples respectively. We note that other recent work [24], [16], [14] on 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 LIVE. Ensuring nonempty intersection of a set of version-intervals $[s_1, e_1]$, ..., $[s_n, e_n]$ 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 LIVE with versioning turned off against version-enabled LIVE. In Figure 2 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 2 shows join

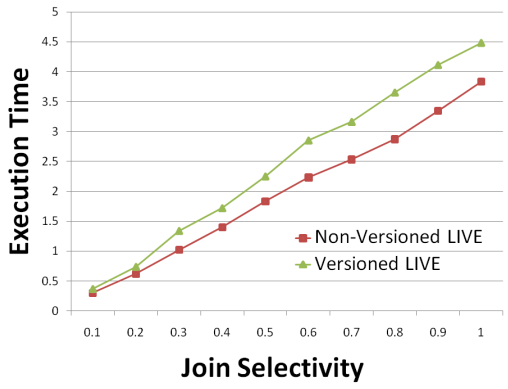

Fig. 2. Overhead of join query processing in LIVE with versions enabled, no modifications.

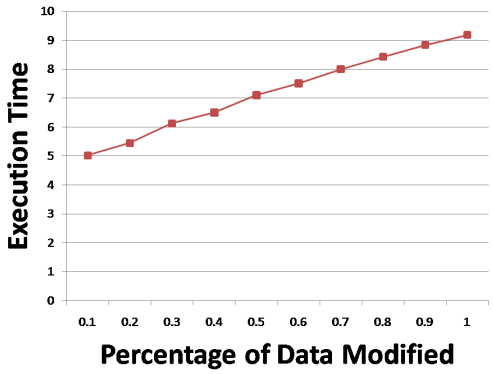

Fig. 3. Scalability of query processing for varying amounts of modified data.

query executions as 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 in 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 tuples that go into the join have been modified in both the `Lineitem` and `Orders` tables. Each modification performs a tuple update, whose effect is to close one interval and create a new tuple as described in Section III-B. Figure 3 displays runtime as a function of the number of tuples 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 3 also shows that query execution time almost exactly doubles when all tuples 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 2. This conforms to the best-possible case, since for each modified (and hence expired) tuple, a new tuple with a different version-interval is inserted into each of the input tables, which also exactly doubles the amount of versioned

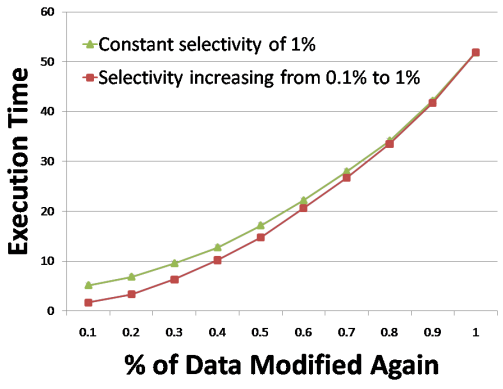

Fig. 4. Query execution time when the amount of modified data increases non-linearly.

tuples in the result at 1% update selectivity.

Figure 4 addresses query processing behavior when different fractions of tuples 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 tuples being modified. As expected, we notice a nonlinear increase in execution time for both cases, because the total amount of data modified and hence the number of versioned tuples, is increasing nonlinearly.

3) Overhead of Version Computation: In Figure 5 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 tuples with both overlapping and non-overlapping versions, because we are able to push the version predicate into the query. In Figure 5, 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 tuple that goes into the join has been modified exactly once, while for the latter plot, the same number of tuples were inserted (but not modified), so that version computation was not necessary.

Figure 6 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.

4) Versioned vs. Valid-at Queries: Figures 7 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

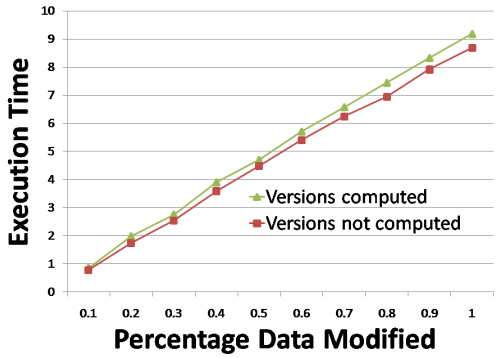

Fig. 5. Overhead of version computation in join query processing.

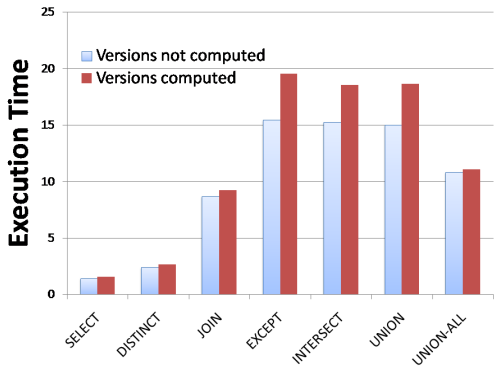

Fig. 6. Overhead of version computation for different operators.

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 7). Hence modifications and versioning in LIVE 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

To the best of our knowledge, LIVE is the first implemented DBMS with unified support for lightweight versioning, probabilistic data, and lineage, in an environment of stored derived relations over base data that is modified over time. We separately describe related work in probabilistic (and uncertain) data, lineage, versioning, and view maintenance.

Probabilistic and uncertain data: While there has been a significant amount of work on uncertain and probabilistic databases in the past few decades (e.g., [13], [15], [25], [21], [22]), surprisingly little work has focused 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 [19], [11], [26], [27], [28], [29], [18], [8], but none of this work we are aware of addresses data modifications or versioning either. While our

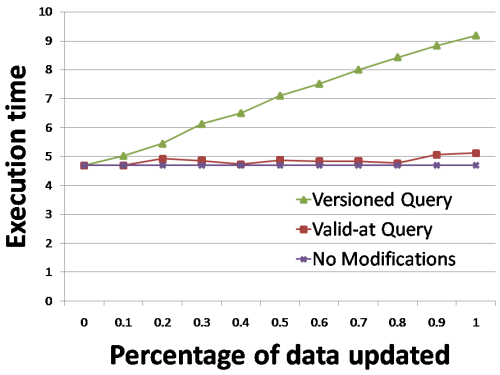

Fig. 7. Valid-at query execution time compared to versioned and non-versioned queries, for varying update selectivity.

paper does not make a new contribution in probabilistic data itself, our support of versioned probabilistic derived data using lineage is new.

In the area of modifications on uncertain and probabilistic databases, an early paper [30] 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. In other early work,[31] 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 [31] is again quite expressive, no query answering or propagation techniques are presented. Similarly, [32] 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., [33], also does not consider propagation or systems issues.

Lineage: There has been a great deal of work in lineage, including but not limited to [11], [34], [35], [36], [37], [38], [39], [40], [41]. We do not make a new contribution to lineage itself in this paper. Rather, we adopt Trio's model and algorithms for lineage [11], and we use it to support efficient management of modifications and versioning in our system, including propagation and query processing.

Versioning: Incorporating versions and notions of time in databases has been studied a great deal (refer to [2], [3]), and continues to be an emerging area with new systems such as [42]. Versioning in databases also has been used for other topics such as concurrency control [43]. However, to the best of our knowledge, no versioning system has been closely integrated with data lineage, derived relations, and probabilistic data. More importantly, our work is not to be confused as introducing a temporal aspect to probabilistic databases. Rather, we introduce a lightweight versioning capability whose primary role is to support data modifications in an environment

of derived relations and lineage.

View Maintenance: A large body of work studies incremental view maintenance in traditional relational databases (refer to [1] for a survey), and in other contexts such as data warehousing [44]. 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 [1], a large body of previous work is devoted to exploiting additional information and features to enable incremental computation in other cases. For instance, reference [4] exploits information about keys in base and derived data, [5], [6] use pointers to base data and modify derived relations *lazily*, while [7] analyzes predicates in queries and properties of modified data to detect when derived relations are not affected by the modifications at all. 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.

VIII. Future Work

This paper describes LIVE, a fully implemented DBMS designed for applications requiring base and derived relations that are modified over time. LIVE incorporates a lightweight versioning system and it supports probabilistic data, for both base and derived relations. The key feature integrating LIVE's variety of capabilities is *lineage*: lineage supports efficient propagation of modifications from base to derived data, and efficient processing of a wide variety of queries on versioned data. The suite of capabilities supported by LIVE are of particular interest in scientific data management (as well as other application domains discussed earlier), as evidenced by the recent *SciDB* proposal [9].

There are several major areas of future work:

- **Modifying derived data:** We have done some preliminary work, not yet complete, on the problem of modifying derived relations and propagating the modifications to the input relations from which they were derived (the analogue of the traditional *view update problem* [45]). As with propagating base relation modifications, it is clear that lineage will play an important role, permitting modifications to a wide class of derived relations, and propagating those modifications to base data efficiently.
- **Nonpropagated modifications:** This paper addresses the case where all modifications to base data are propagated to derived relations. Some 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.

References

[1] A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *IEEE Data Engineering Bulletin*, vol. 18, no. 2, 1995.

[2] C. Date and H. Darwen, *Temporal Data and the Relational Model*. Morgan Kaufmann Publishers, 2002.

[3] R. T. Snodgrass, *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers, 2000.

[4] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," in *Proc. of VLDB*, 1991.

[5] N. Roussopoulos, "View indexing in relational databases," *TODS*, vol. 7, no. 2, 1982.

[6] N. Roussopoulos, N. Economou, and A. Stamenas, "ADMS: A testbed for incremental access methods," *IEEE TKDE*, vol. 5, no. 5, 1993.

[7] J. A. Blakeley, P. Larson, and F. W. Tompa, "Efficiently updating materialized views," in *Proc. of ACM SIGMOD*, 1986.

[8] J. Widom, "Trio: A System for Integrated Management of Data, Accuracy, and Lineage," in *Proc. of CIDR*, 2005.

[9] "SciDB," *http://confluence.slac.stanford.edu/display/XLDB/SciDB*.

[10] L. Haas, "The theory and practice of information integration," in *Proc. of ICDT*, 2007.

[11] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom, "ULDBs: Databases with uncertainty and lineage," in *Proc. of VLDB*, 2006.

[12] R. Cavallo and M. Pittarelli, "The theory of probabilistic databases," in *Proc. of VLDB*, 1987.

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

[14] A. Das Sarma, M. Theobald, and J. Widom, "Exploiting lineage for confidence computation in uncertain and probabilistic databases," in *Proc. of ICDE*, 2008.

[15] S. Abiteboul, P. Kanellakis, and G. Grahne, "On the Representation and Querying of Sets of Possible Worlds," *Theoretical Computer Science*, vol. 78, no. 1, 1991.

[16] N. Dalvi and D. Suciu, "Efficient Query Evaluation on Probabilistic Databases," in *Proc. of VLDB*, 2004.

[17] C. Re, N. Dalvi, and D. Suciu, "Efficient top-k query evaluation on probabilistic data," in *Proc. of ICDE*, 2007.

[18] P. Sen and A. Deshpande, "Representing and Querying Correlated Tuples in Probabilistic Databases," in *Proc. of ICDE*, 2007.

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

[20] A. Das Sarma, O. Benjelloun, A. Halevy, and J. Widom, "Working Models for Uncertain Data," in *Proc. of ICDE*, 2006.

[21] T. Imielinski and W. Lipski., "Incomplete Information in Relational Databases," *Journal of the ACM*, vol. 31, no. 4, 1984.

[22] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. Subrahmanian, "ProbView: A Flexible Probabilistic Database System," *ACM TODS*, vol. 22, no. 3, 1997.

[23] T. P. C. (TPC), "TPC Benchmark H: Standard Specification, 2006," `http://www.tpc.org/tpch`.

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

[25] N. Fuhr and T. Rölleke, "A Probabilistic NF2 Relational Algebra for Imprecision in Databases," *Unpublished Manuscript*, 1997.

[26] 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.

[27] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan, "OLAP over uncertain and imprecise data," *J. VLDB*, vol. 16, no. 1, pp. 123–144, 2007.

[28] R. Cheng, S. Singh, and S. Prabhakar, "U-DBMS: A database system for managing constantly-evolving data." in *Proc. of VLDB*, 2005.
[29] T. J. Green and V. Tannen, "Models for incomplete and probabilistic information," in *Proc. of IIDB Workshop*, 2006.
[30] S. Abiteboul and G. Grahne, "Update semantics for incomplete databases," in *Proc. of VLDB*, 1985.
[31] S. J. Hegner, "Specification and implementation of programs for updating incomplete information databases," in *PODS*, 1987.
[32] A. M. Keller and M. Winslett, "Approaches for updating databases with incomplete information and nulls," in *Proc. of ICDE*, 1984.
[33] M. Winslett, "A model-based approach to updating databases with incomplete information," *ACM TODS*, vol. 13, no. 2, 1988.
[34] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *Proc. of ACM SIGMOD*, 2006.
[35] A. Chapman and H. V. Jagadish, "Issues in building practical provenance systems," *IEEE Data Engineering Bulletin*, 2007.
[36] L. Chiticariu, W. Tan, and G. Vijayvargiya, "DBNotes: a post-it system for relational databases based on provenance." in *Proc. of ACM SIGMOD*, 2005.
[37] Y. Cui and J. Widom, "Lineage tracing for general data warehouse transformations," *VLDB Journal*, vol. 12, no. 1, 2003.
[38] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proc. of ACM PODS*, 2007.
[39] C. Re and D. Suciu, "Approximate lineage for probabilistic databases," in *Proc. of VLDB*, 2008.
[40] W.-C. Tan, "Provenance in Databases: Past, Current, and Future," *IEEE Data Engineering Bulletin*, 2008.
[41] A. Woodruff and M. Stonebraker, "Supporting fine-grained data lineage in a database visualization environment," in *Proc. of ICDE*, 1997, pp. 91–102.
[42] 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.
[43] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing, 1987.
[44] W. Labio, J. Yang, Y. Cui, H.-G. Molina, and J. Widom, "Performance issues in incremental warehouse maintenance," in *Proc. of VLDB*, 2000.
[45] U. Dayal and P. A. Bernstein, "On the updatability of relational views," in *Proc. of VLDB*, 1978.
[46] 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.

APPENDIX

A. Proofs

Proof of Theorem 2.6: The proof of LDM completeness follows from the completeness of ULDBs [11], which are probabilistic databases with lineage (without versions). 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, which can be equivalently represented as a non-versioned LDM. Then, we construct LDM 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 tuple from D_i, $i < v_D$, to $[i, i]$, and set the version-interval of each tuple 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 probabilities are known. Consider a DNF lineage formula with negations pushed down to literals, λ, of some tuple a in an LDM database. We prove the equality of the computed version-interval I_λ 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 λ_{v_0} of λ obtained by setting to *false* every tuple in λ whose version-interval does not contain v_0. Since $v_0 \in I_{correct}$, λ_{v_0} is satisfiable. Given a satisfying assignment of λ_{v_0}, we can find a disjunct in the DNF formula λ that is satisfied. The version-interval of this disjunct alone contains v_0. Since I_λ is the union of the version-intervals of each of λ's disjuncts, $v_0 \in I_\lambda$.
  2) $I_{correct} \supseteq I_\lambda$: Now suppose $v_0 \in I_\lambda$. We show that λ_{v_0}, obtained by setting to *false* every tuple in λ 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 λ. Every tuple in this disjunct whose version-interval contains v_0 can be set to true to satisfy λ and hence satisfy λ_{v_0}.
- **NP-hard:** To correctly compute the version-interval for a tuple, we need to check for each input tuple, whether its probability is 1 at any version number. Detecting whether the probability of a boolean formula is 1 or not is NP-hard as it is equivalent to checking the satisfiability of its complement. □

B. 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.
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.

C. Propagating Insertions

Next consider propagating insertions to derived relations. We are able to leverage traditional incremental view maintenance techniques [1] and lineage in ULDB^{v}s.

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 [46] 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 [11].)
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.

D. 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.