Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases*

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

ABSTRACT

We study the problem of computing query results with confidence values in *ULDBs*: relational databases with *uncertainty* and *lineage*. ULDBs, which subsume *probabilistic databases*, offer an alternative *decoupled* method of computing confidence values: Instead of computing confidences during query processing, compute them afterwards based on lineage. This approach enables a wider space of query plans, and it permits selective computations when not all confidence values are needed. This paper develops a suite of algorithms and optimizations for a broad class of relational queries on ULDBs. We provide confidence computation algorithms for single data items, as well as efficient batch algorithms to compute confidences for an entire relation or database. All algorithms incorporate memoization to avoid redundant computations, and they have been implemented in the *Trio* prototype ULDB database system. Performance characteristics and scalability of the algorithms are demonstrated through experimental results over a large synthetic dataset.

1. INTRODUCTION

We consider the problem of processing queries efficiently over uncertain data that includes *confidence* (or *probability*) values. Uncertain and probabilistic databases have enjoyed considerable recent interest (e.g., [2, 5, 7, 9]), due to their relevance for important current applications such as data cleaning and integration, information extraction, scientific and sensor data management, and others [33].

A central and particularly challenging aspect of processing queries over uncertain data is that of computing confidence values on query results. (Note that the terms *uncertain data* and *confidences* used here correspond to the terms *probabilistic data* and *probabilities* sometimes used elsewhere.) It has been shown previously [13] that the straightforward approach of propagating confidence values through the operators comprising a query plan can, in certain cases, yield incorrect confidence values in query results. Roughly, the flaw is that dependencies among the base data and confidence values contributing to the result data and confidences are lost during query execution.

One approach to handling this problem, studied in [13, 29], restricts the allowable query plans to "safe" ones, which ensure correct propagation of confidence values. When the best plan for a query is safe, then this approach yields a very efficient query processing strategy. However, sometimes the most efficient plan for a query is not safe—in fact, in extreme cases a query's safe plan may be arbitrarily worse than the best unsafe one [4]. Furthermore, for some queries there is no safe plan at all.

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

In this paper we explore an alternative approach to the problem of correct confidence computation. In the *Trio* project at Stanford we are developing a database system that incorporates *data lineage* as a first-class concept, along with uncertain data and confidence values [5, 24]. It turns out that the lineage information tracked by Trio can be used to compute confidence values. Specifically, data computation and confidence computation can be *decoupled*: First compute the data in the query result, ignoring confidence values. Lineage is tracked as well, but unlike confidence computation, lineage-tracking does not restrict the allowable query plans; thus, the most efficient query plan can be selected. After data computation, confidence values can be computed at any time, based on lineage. Effectively, lineage captures the dependencies that are needed for correct confidence computation, without restricting which query plans can be used.

In addition to enabling a wider space of query plans, the decoupled approach also enables *selective* confidence computation: If confidence values for some portions of the result data are not needed, they will never be computed. Since confidence computation can be an unavoidably expensive operation in some cases [13, 30], this feature can be very useful in practice.

At the most general level, one can envision a query optimizer that explores a space of execution strategies encompassing efficient safe plans when they exist, lineage-based schemes, and hybrids of the two approaches. We do not tackle a global query optimization problem of that scale in this paper. Rather, we focus on developing the framework and algorithms for efficient decoupled data and confidence computation based on lineage.

The outline and contributions of this paper are as follows.

- In Section 2 we review *ULDB*s (*Uncertainty-Lineage Databases*) and explain how they correspond to other models for uncertain and probabilistic data. (Sections 3–5 consider a restriction of ULDBs whose uncertainty component is equivalent to probabilistic databases. Section 6 extends to the full ULDB model.) We also motivate, through a running example, safe and unsafe plans, and how lineage in ULDBs enables the use of unsafe plans.

- In Section 3 we formalize query processing on ULDBs for a wide class of relational operators and arbitrary algebraic plans,[1] and we specify decoupled confidence computation. The basic algorithm for computing the confidence of a result tuple t expands t's lineage to generate a boolean formula B over the base data contributing to t, then evaluates B probabilistically using the base-data confidence values. The final

[1]Our previous algorithms [4] were limited to the duplicate-preserving "DL-monotonic" operators σ, Π, $\bowtie$, and $\cup$, which produce conjunctive lineage. In this paper we extend lineage to boolean formulas, and we capture full relational algebra including duplicate-elimination and aggregation.

evaluation is unavoidably a #P-hard problem [13, 30], but in practice the boolean formulas tend to be small and/or simple.

- In Section 4 we introduce algorithms that improve upon the basic confidence-computation algorithm when computing confidence values for tuples with complex or deep lineage. The improved algorithms are based on *memoizing* confidence values as they are computed, and determining when "intermediate" data in the lineage of a tuple is mutually independent. These techniques can reduce both the lineage expansion during confidence computation, and the complexity of the resulting boolean formulas.

- In Section 5 we turn to *batch* algorithms. These algorithms are suitable when confidence values for all (or even many) tuples in a query result are needed at once. The batch algorithms also can be used to compute confidence values for an entire lineage-interconnected database.

- In Section 6 we extend our results to the more general ULDB model as supported in the Trio system, which also includes *alternatives* [4, 5]. We show that only small modifications to our algorithms are needed for the more general model. We also discuss how our confidence computation algorithms can be adapted for other important Trio functionality, including finding *extraneous data* and performing *coexistence checks* [24].

- We have implemented all of our algorithms in the Trio prototype DBMS [24]. In Section 7 we report on performance experiments over a probabilistic version of the TPC-H database. Our performance experiments demonstrate scalability and confirm the efficiency gained by our optimizations over the basic approach. We also present experiments identifying crossover points between tuple-level and batch computations, i.e., determining at what fractions of the data batch algorithms become more efficient.

Related work is discussed next. In Section 8, we conclude with a discussion of future directions for efficient confidence computation, and more generally for query optimization over uncertain data.

1.1 Related Work

Modeling uncertain and probabilistic information in relational databases has been an area of research for nearly 20 years. Much work, especially earlier papers, focuses on theoretical foundations, e.g., [1, 3, 14, 16, 17, 18, 19, 20, 21], and not on practical considerations such as efficient query processing, the subject of this paper. Recent interest has turned to developing prototype systems, e.g., [2, 7, 9, 24]. *MayBMS* [2] uses a model different from ours for representing uncertain and probabilistic data, and efficient probability computation is not discussed. *MystiQ* [7] is a system for managing probabilistic data based on safe plans [13], as discussed earlier and to be discussed further in Section 2. The *Orion* system [9] focuses on continuous uncertainty for sensor applications, rather than discrete forms of uncertainty with confidence values. considers combining and contrasting tuple-level and batch confidence computation.

Lineage-tracking in relational databases has also been studied, e.g., [6, 8, 10, 11, 12, 25, 26, 34]. However, none of that work uses lineage in conjunction with uncertainty. ULDBs are unique in their exploitation of lineage to help model uncertainty in data and query results [4], and to improve query processing on uncertain data as explored in this paper.

Intensional query evaluation in probabilistic database systems, as introduced in [17] and most recently discussed in [13, 30, 31], generates a global boolean formula of constraints among tuples, call it B, which can then be used to compute probabilities. In our setting, lineage captures all of the information from B that is necessary for confidence computation. We focus on optimizing the process of gathering and exploiting this information.

Confidence computation based on lineage has some parallels with Artificial Intelligence inferencing algorithms, such as the *clique tree propagation* algorithm for Bayes Nets [27]. Reference [31] shows how to model probabilistic databases using *probabilistic graphical models* [27] and formulates confidence computation as an inference problem, which allows standard AI inferencing algorithms to be exploited. These algorithms are quite different in flavor from ours. For example, they require each data item to be associated with a probability table, and all intermediate results and their probability tables must be maintained during query processing. In our case, we store just tuples in the database and their lineage. Furthermore, we exploit the database query processor for our confidence-computation tasks. Notably, we can perform batch computations via standard SQL queries. A careful study comparing the two broad approaches—AI techniques (as in [31]) and database techniques (as in [13, 30] and this paper)—is not a subject of this paper and is left as future work.

Note that the decoupled confidence-computation approach was suggested by us originally and briefly in [4], with some motivating examples. However, no algorithms, implementation, or experiments were presented in that paper.

2. MOTIVATION AND EXAMPLES

This section introduces the ULDB model for representing uncertainty and lineage in relational databases [4, 5, 24]. Until Section 6 we focus on a subset of ULDBs whose uncertainty component is equivalent to probabilistic databases. Section 6 extends our approach to the more general ULDB model. This section also introduces a running example, used here to informally explain query processing with confidences and lineage, and to motivate the issues addressed by the remainder of the paper.

2.1 Data and Confidences

In the model we consider initially, a ULDB relation is similar to a conventional relation, except that each tuple is annotated with a numerical *confidence value* (equivalently *probability*) in the range $[0, 1]$, and with *lineage*, which is explained in the next subsection. (The general ULDB model permits *alternative values* for each tuple, with confidences and lineage attached to alternatives; see Section 6.) ULDB relations may contain duplicates, but we assume a (possibly internal) globally unique identifier $I(t)$ for each tuple t in the database. Note that we sometimes abuse notation and use t instead of its identifier $I(t)$, when the context is clear.

ULDBs have a conventional "possible-instances" semantics: Each ULDB relation R represents a set of possible conventional relations $R_1, \ldots R_n$. Each R_i has an associated probability P_i based on the confidences of the tuples that appear in it. In the absence of lineage, tuple appearances are independent of one another. $P_1, \ldots, P_n$ sum to 1. This possible-instances definition extends directly to databases comprised of several relations. Details of this semantics can be found in many sources, e.g., [4, 13, 30, 31].

We assume that confidence values on base data are provided by the user. Then, as queries are performed generating derived relations, result confidences are computed by the system. It is the latter computation that is the primary subject of this paper.

2.2 Lineage

Each tuple t in a ULDB relation also contains *lineage*, which intuitively captures "where t came from." Lineage is represented as a function λ that associates with each tuple identifier $I(t)$ a boolean formula whose symbols are other tuple identifiers in the database. Base relations are not derived from other data, so we define $\lambda(t) = t$ for every tuple t in a base relation. Now suppose a ULDB relation R is the result of a query over other ULDB relations $S_1, \ldots, S_m$, and consider a tuple $t \in R$. $\lambda(t)$ is a formula involving tuple identifiers from $S_1, \ldots, S_m$, and the formula reflects the query that produced t. We will shortly see an example of how lineage formulas are produced; the process is specified rigorously in Section 3.

Lineage constrains the possible-instances described in the previous section: Now a ULDB represents only those possible instances that are consistent with respect to lineage. Specifically, for a given instance D of a ULDB database U, we assign to each tuple identifier $I(t)$ in U the value *true* if t appears in D and *false* if it does not. D is consistent with respect to lineage iff for every tuple $t \in D$, the boolean formula $\lambda(t)$ evaluates to *true* using the above assignments.

We will see in the upcoming running example and then more formally in Section 3 how lineage can be used to compute confidence values for query results. Intuitively, the confidence of a result tuple t is the probabilistic evaluation of the boolean formula $\lambda(t)$, using the confidence values for the tuples identified in $\lambda(t)$. However, if $\lambda(t)$ contains derived tuples, in general the formula must be "expanded" recursively to refer only to base tuples, before computing its probability.

We sometimes abuse notation and use $\lambda(t)$ to refer to the set of variables in the boolean formula, rather than the formula itself. Also, we sometimes use $\Pr(i)$ to denote the confidence (probability) of the tuple with identifier i, and for a boolean formula f over identifiers we use $\Pr(f)$ to denote the probability of the formula f.

2.3 Running Example

We introduce an extremely simple and tiny conference database as a running example. Let base relation `Attends(person,day)` contain days on which people may attend the conference, and let `Events(day,event)` contain scheduled conference activities. Suppose our relations contain the following data, with confidence values shown to the right of each tuple. (The Reception and Banquet are outdoors, so their occurrence depends on weather.)

Attends

ID	person	day	
11	Garcia-Molina	Monday	0.8
12	Garcia-Molina	Wednesday	0.7
13	Ullman	Wednesday	0.6

Events

ID	day	event	
21	Monday	Reception	0.8
22	Tuesday	Museum	1.0
23	Wednesday	Banquet	0.9

Since `Attends` and `Events` are base relations, the lineage of their tuples is the tuple itself (e.g., $\lambda(11) = 11$). Now suppose we add the following two derived relations to the database. For compactness of examples in this section we assume projection Π is duplicate-eliminating. In our actual language, we use multiset operators and a separate duplicate-elimination operator δ; see Section 3.

`EventRoster` = $\Pi_{\texttt{person,event}}$`(Attends` $\bowtie$ `Events)`

`EventAttendees` = $\Pi_{\texttt{person}}$`(EventRoster)`

EventRoster

ID	person	event		
31	Garcia-Molina	Reception	0.64	$\lambda(31) = 11 \wedge 21$
32	Garcia-Molina	Banquet	0.63	$\lambda(32) = 12 \wedge 23$
33	Ullman	Banquet	0.54	$\lambda(33) = 13 \wedge 23$

EventAttendees

ID	person		
41	Garcia-Molina	0.8668	$\lambda(41) = 31 \vee 32$
42	Ullman	0.54	$\lambda(42) = 33$

The lineage of tuples in these relations identifies the base-relation tuples from which they were derived. For example, $\lambda(33) = 13 \wedge 23$ indicates that `(Ullman,Banquet)` appears in `EventRoster` because of tuples `(Ullman,Wednesday)` and `(Wednesday, Banquet)` in `Attends` and `Events`, respectively.

Joins produce conjunctive lineage: a result tuple is present because two tuples are both present in the input relations. The duplicate-eliminating projection generating `EventAttendees` produces disjunctive lineage: $\lambda(41) = 31 \vee 32$ indicates that `Garcia-Molina` is an event attendee because he attends the `Reception` (tuple 31) or the `Banquet` (tuple 32), or possibly both. Here we also see an instance of transitive lineage: tuple 41 is derived from tuples that are in turn derived from other tuples.

We maintain lineage at the granularity of queries, as in this example: When a relation R results from a query over $S_1, \ldots S_n$, then the lineage of tuples in R identifies tuples in $S_1, \ldots S_n$ (regardless of the complexity of the query). However, as mentioned earlier, correct confidence computation may require us to expand lineage transitively, when one or more of the S_i's are themselves derived relations. For example, the expansion of $\lambda(41)$ is $(11 \wedge 21) \vee (12 \wedge 23)$.

Now consider confidence values. The confidences in `EventRoster` are easily seen to be the product of the corresponding tuples' confidences in the base relations `Attends` and `Events`, since the join operates on independent base data. Next consider `EventAttendees`. Tuple 42 is derived from tuple 33 in `EventRoster`, so 33's confidence value carries over directly. Tuple 41 is more complex—it results from merging duplicate values from tuples 31 and 32. The confidence of tuple 41 is thus the probability that at least one of tuple 31 or 32 is present. Since tuples 31 and 32 are independent, we calculate $\Pr(31 \vee 32) = \Pr(31) + \Pr(32) - \Pr(31 \wedge 32) = 0.64 + 0.63 - (0.64 \cdot 0.63) = 0.8668$.

2.4 Safe/Unsafe Confidence Propagation

Now let us consider computing confidences for a query with two relationally-equivalent arrangements of operators. Looking at the previous example, one might think we can just compute confidences within each operator comprising a query, propagating result confidences to the next operator in the query and finally to the result. However, as shown originally in [13], this approach does not work for all query expressions. Here is an example showing that not all equivalent expressions (plans) for a query permit operator-based confidence computation.

For further conciseness suppose `Attends` and `Events` contain only the tuples involving `Wednesday`, i.e., two tuples in `Attends` and one in `Events`. At our imagined conference, events are also canceled if they have no attendees (in addition to weather cancellations), and we want to determine the likelihood of Wednesday's `Banquet` taking place. We use the query:

`EventLikelihood` = $\Pi_{\texttt{event}}$ `(Attends` $\bowtie$ `Events)`

where again for now Π is duplicate-eliminating. The correct answer including lineage and confidence is:

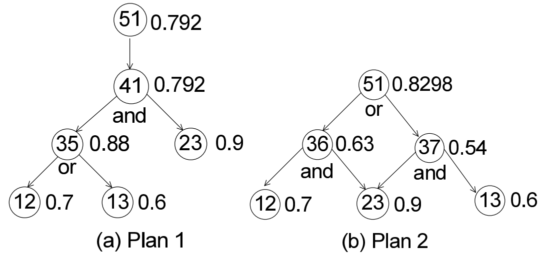

Figure 1: Safe and unsafe confidence propagation

EventLikelihood

ID	Event		
51	Banquet	0.792	$\lambda(51) = (12 \vee 13) \wedge 23$

We consider two relationally equivalent plans for evaluating this query. Plans are specified as an algebraic expression whose operators are evaluated one by one in the usual way, except lineage and confidence values are propagated through the operators along with the data. First consider:

Plan 1: $\Pi_{\texttt{event}}(\Pi_{\texttt{day}}(\texttt{Attends}) \bowtie \texttt{Events})$

$\Pi_{\texttt{day}}(\texttt{Attends})$ produces one intermediate tuple `(Wednesday)`; let's call it 35. Its lineage is $\lambda(35) = 12 \vee 13$, and its confidence is $\Pr(12 \vee 13) = \Pr(12) + \Pr(13) - (\Pr(12) \cdot \Pr(13)) = 0.7 + 0.6 - 0.7 \cdot 0.6 = 0.88$. Now, joining `(Wednesday)` with tuple 23, we get one intermediate tuple `(Wednesday,Banquet)`; let's call it 41. 41's "immediate" (operator-level) lineage is $\lambda(41) = 23 \wedge 35$, and we compute its confidence as $\Pr(23 \wedge 35) = \Pr(23) \cdot \Pr(35) = 0.9 \cdot 0.88 = 0.792$. Finally, we project onto `event` to produce result tuple `(Banquet)`; call it 51, whose confidence remains 0.792. The operator-level lineage of 51 is $\lambda(51) = 41$, but we can expand the final lineage formula to obtain $\lambda(51) = (12 \vee 13) \wedge 23$.

Now suppose we use the relationally equivalent plan:

Plan 2: $\Pi_{\texttt{event}}(\texttt{Attends} \bowtie \texttt{Events})$

The join produces two intermediate tuples: `(Garcia-Molina, Wednesday,Banquet)` and `(Ullman,Wednesday, Banquet)`; call them 36 and 37 respectively. $\lambda(36) = 12 \wedge 23$ and $\lambda(37) = 13 \wedge 23$, and the confidence values are $0.7 \cdot 0.9 = 0.63$ for 36 and $0.6 \cdot 0.9 = 0.54$ for 37. Next we project onto `event`, yielding result tuple `(Banquet)`, again denoted 51. At the operator level, $\lambda(51) = 36 \vee 37$, and its confidence is $\Pr(36 \vee 37) = \Pr(36) + \Pr(37) - \Pr(36 \wedge 37)$. Assuming independence of tuples 36 and 37 (which turns out to be false), operator-level confidence propagation computes $\Pr(36 \vee 37) = \Pr(36) + \Pr(37) - (\Pr(36) \cdot \Pr(37)) = 0.63 + 0.54 - 0.63 \cdot 0.54 = 0.8298$. Thus, we get an incorrect result.

In the terminology of [13], Plan 1 is *safe*, while Plan 2 is *unsafe*. Informally, a plan is safe if computing confidences operator-by-operator assuming independence of input tuples is guaranteed to produce the correct result. Notice what happens when we expand the final lineage formula from "incorrect" Plan 2: $\lambda(51) = 36 \vee 37 = (12 \wedge 23) \vee (13 \wedge 23) = (12 \vee 13) \wedge 23$. Thus, had we computed the final confidence value based on the expanded lineage, rather than operator-by-operator, it would have been correct.

This example motivates how lineage enables a wider space of query plans, without losing the ability to compute confidence values correctly. Effectively, lineage captures the data dependencies that are lost in the operator-by-operator approach. To further illustrate this fundamental issue, consider Figure 1. Figure 1(a) depicts the computation of Plan 1 as a boolean formula tree with nodes representing base, intermediate, and result tuples. Each intermediate node can safely assume its children are independent. Figure 1(b) depicts the computation of Plan 2. Here, it is incorrect for node 51 to assume its children are independent, because they both depend on node 23. This dependency is captured by lineage.

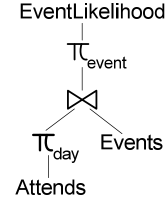

Figure 2: Expression tree for Plan 1

3. QUERY PROCESSING

In this section we formalize and provide algorithms for query processing with confidences and lineage, as motivated by the examples in Section 2. Section 3.1 discusses the semantics and evaluation of one query on a ULDB, prior to confidence computation. Section 3.2 describes what happens when a sequence of queries is executed, potentially resulting in deep and complex lineage. Section 3.3 provides our basic decoupled confidence-computation algorithm, which is then improved in the rest of the paper.

3.1 Query Execution

The formal semantics of queries over ULDBs is defined based on possible instances: The result of performing a query Q on a ULDB D with possible instances $D_1, \ldots, D_n$ is a ULDB D' containing the original relations and an additional *result relation* R. R must reflect the result of applying query Q to the possible instances of D; that is, the possible instances of R in D' must be $Q(D_1)$, ..., $Q(D_n)$. For details see [4].

Now we define query plans. Consider a query Q over ULDB relations $S_1, \ldots, S_n$ producing result relation R. A *query plan* to evaluate Q is an expression tree, with leaf nodes $S_1, \ldots, S_n$ and non-leaf nodes corresponding to relational algebra operators. For example, Figure 2 depicts the expression tree for Plan 1 of Section 2.4. Query plan evaluation is defined in the standard fashion: Each operator takes as input a set of relations and produces one relation as output; the final query result is output by the root. In our case, operators process ULDB relations, including both data and lineage.

We specify our operators so that each one "unfolds" lineage as it produces results. When the relations $S_1, \ldots, S_n$ at the leaves of the plan are base relations, by definition all of their tuples t have $\lambda(t) = t$. Thus, the "unfolded" lineage that is output by every operator, including the root, refers to data in the input relations. That is, we never generate lineage referring to intermediate results. If we have a derived input relation S, we want query result lineage to refer to tuples in S, rather than further "unfolding" S's lineage (which might be arbitrarily deep). Thus, when executing a query plan, we treat the lineage of all input tuples t as if $\lambda(t) = t$.

Algorithms for all relational operators (σ, π, $\bowtie$, $\cup$, $\cap$, $-$, δ, α), omitted due to space constraints, are provided in the online technical report [15]. (Operators δ and α are duplicate-elimination and aggregation respectively.) The following theorem, proved in the online technical report [15], tells us that for any query Q, any relationally-equivalent plan can be used to evaluate Q, i.e., the ULDB setting does not impose restrictions on the types of plans

we can use. In this paper we do not address the problem of exploring the plan space—building a cost-based optimizer that considers both data and lineage computations is an important subject of future work. Here we focus on confidence computation, which occurs after plan selection and execution.

THEOREM 3.1. Consider two relationally-equivalent query plans P_1 and P_2 evaluated over a set of input relations in a ULDB D, yielding ULDB result relations R_1 and R_2. $D + R_1$ and $D + R_2$ have identical possible instances. □

In most cases, result relations R_1 and R_2 are "isomorphic." However, there are counterexamples: In the online technical report [15] we show two equivalent plans for a query (involving set operators and duplicate-elimination) where the two result relations are not isomorphic, but they do have the same possible instances as guaranteed by the theorem.

3.2 Sequences of Queries

A single query on a ULDB generates a result (derived) relation R and lineage connecting R's tuples to tuples in the input relations $S_1, \ldots, S_n$. Lineage in query results is captured by function λ: $\lambda(t)$ for a tuple $t \in R$ is a boolean formula over tuples in $S_1, \ldots, S_n$ specifying the lineage of t. For instance, using either Plan 1 or Plan 2 for the `EventLikelihood` query in Section 2.4, the lineage of the one result tuple $t = (\texttt{Banquet})$ is given by $\lambda(t) = (12 \vee 13) \wedge 23$.

Consider an initial database D consisting of base relations, and a sequence $Q_1, Q_2, Q_3, \ldots$, of queries. Each query Q_i adds a new derived relation R_i to the database, and subsequent queries may be over base and derived relations; i.e., Q_i can be over any of $D \cup \{R_1, \ldots, R_{i-1}\}$. Let t be a tuple in a result relation R_i. $\lambda(t)$ may refer to a tuple t' in a previous derived relation R_j ($j < i$); t' may in turn refer to other derived or base relations. Thus, as a sequence of queries are performed on a ULDB, complex and "deep" lineage relationships may arise. However, since every Q_i can refer only to the original D and to R_j's with $j < i$, lineage is acyclic.

In this paper we do not consider the problem of handling updates to data or confidence values in base or derived relations, when other relations are derived from them. This topic is an important area of future work not yet addressed in the Trio system.

3.3 Confidence Computation

Confidence values for query results can be computed at any time, based on lineage. Specifically, suppose we've executed a sequence of queries as described in the previous section, capturing lineage for all query results. We may request at any time the confidence of any result tuple t. (Later we will discuss batch confidence computation for entire relations or databases.) Algorithm 1 provides pseudocode for our basic approach to computing the confidence of tuple t using its lineage. This algorithm is the basis for improvements in the following sections. Note that even this basic algorithm is an improvement over previous approaches in the sense that it does not limit the query plans we can use for data computation.

Algorithm 1 assumes two primitive functions supported by the ULDB: $\lambda(t)$ returns the lineage formula for t, and $c(t)$ returns the current confidence value of t stored in the database. We assume $c(t)$ returns `NULL` whenever the confidence of t has not been computed, and that confidences are always present for base data. As we shall see later, one obvious improvement is to memoize and reuse computed confidence values, although doing so still requires care.

The main function *Conf*(t) in Algorithm 1 returns t's confidence if already computed. Otherwise it calls the recursive function *Compute-Conf*(f) on t's lineage formula f. This function computes a final boolean formula B by recursively expanding the current formula until all variables refer to base tuples. Then, the probability of B is computed and returned using the confidence values of the base tuples. We assume a function *Eval*, which takes a boolean formula over independent variables and the probabilities of each variable as input and computes the probability of the boolean formula. *Eval* is discussed further after the following example.

```
1:  Conf(t) {
2:      if c(t) ≠ NULL then return c(t);
3:      else return Compute-Conf(λ(t)); }
4:  Compute-Conf( f(t_1, t_2, ..., t_n) ) {
5:      all-base = true;
6:      for (i = 1; i ≤ n; i++) {
7:          g_i = λ(t_i);
8:          if (g_i ≠ t_i) then all-base = false; }
9:      if all-base then {
10:         for (i = 1; i ≤ n; i++): c_i = c(t_i);
11:         return Eval(f, c_1, c_2, ..., c_n); }
12:     else return Compute-Conf(f(g_1, g_2, ..., g_n)); }
```

Algorithm 1: Basic confidence computation

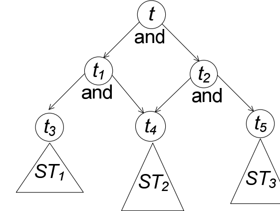

Figure 3: Lineage of tuple t

EXAMPLE 3.2. Consider the two query plans for `EventLikelihood` from Section 2.4, safe Plan 1 and unsafe Plan 2. Suppose we want to compute the confidence of the single projected result tuple `(Banquet)`. Since lineage of tuples in query results always refers to tuples in the query input, in the query results for both plans we have $\lambda(\texttt{Banquet}) = (12 \vee 13) \wedge 23$, where 12, 13, and 23 are base tuples. Thus $\Pr(\texttt{Banquet}) = \Pr((12 \vee 13) \wedge 23)$. Using $\Pr(12) = 0.7$, $\Pr(13) = 0.6$, and $\Pr(23) = 0.9$, we get $\Pr(\texttt{Banquet}) = 0.792$, which is the correct answer. □

In general, computing the probability of an arbitrary boolean formula of independent events (our function *Eval*) is known to have exponential worst-case complexity [23, 28]. This computational lower bound cannot be overcome regardless of the data model and algorithm, although in practice lineage formulas are frequently small and/or simple. For a complex lineage formula, if approximate answers suffice then we could employ Monte-Carlo simulations [22] for the final computation, as proposed in [13, 30].

Next we improve upon our basic confidence computation algorithm: Section 4 presents an improved algorithm for confidence computation of a single tuple t: It breaks t's lineage formula into independent pieces when possible and memoizes computed confidences for reuse. Section 5 presents algorithms for efficiently computing the confidences of an entire relation or database at once.

4. TUPLE-LEVEL CONFIDENCES

In Algorithm 1, consider the function *Compute-Conf*, which is called with argument $f = \lambda(t)$. In general *Compute-Conf* takes a boolean formula f over tuples $t_1, \ldots, t_n$ and expands the lineage

```
1:  Conf(t) {
2:      if c(t) ≠ NULL then return c(t);
3:      else {  S = Indep(t);
4:          return Compute-Conf(S, λ(t)); }
5:  Compute-Conf( S, f(t_1, t_2, ..., t_n) ) {
6:      all-indep = true;
7:      for (i = 1; i ≤ n; i++) {
8:          g_i = t_i;
9:          if t_i ∉ S then { g_i = λ(t_i);
10:                 all-indep = false; }
11:     if all-indep then {
12:         for (i = 1; i ≤ n; i++): c_i = Conf(t_i);
13:         return Eval(f, c_1, c_2, ..., c_n); }
14:     else return Compute-Conf(S, f(g_1, g_2, ..., g_n)); }
```

Algorithm 2: Confidence computation using *Indep*()

of each t_i by calling $\lambda(t_i)$. It then calls itself recursively to further expand the formula, until all variables refer to base tuples. At this point base tuple confidences are used to evaluate the probability of the fully-expanded formula.

In general the algorithm must expand lineage all the way to the base tuples to ensure independence among the variables in the formula, before *Eval* is called to compute its probability. Consider Figure 3, depicting the lineage of a tuple t as a graph. The ST_i's denote arbitrary but disjoint subgraphs. Our basic algorithm expands the lineage of t to the base data to obtain a single formula whose probability is evaluated. Notice, however, that the lineage of tuples t_3, t_4, and t_5 is disjoint, therefore their confidence values are independent. Thus, we can "break up" our computation as follows: We compute confidence values for t_3, t_4, and t_5 separately by calling *Conf*; suppose the results are c_3, c_4, and c_5. Then, when we expand the lineage formula for t, instead of expanding all the way to the base data, we "stop" at t_3, t_4, and t_5, since we have their independent confidence values. Essentially, we can treat t_3, t_4, and t_5 as base data. This technique can be applied to any set of tuples whose lineage "subtrees" are disjoint. Breaking up the computation in this fashion has two benefits:

1. **Cheaper Computation:** The original algorithm computes one formula f over base data, then incurs the expensive (usually exponential) cost of *Eval*. By replacing f with a set of smaller formulas that *Eval* operates on separately, usually the overall cost is reduced. For example, suppose f contains 40 variables, so the complexity of evaluating the probability of f is potentially $O(2^{40})$. If our technique replaces f with five formulas of 8 variables each, plus one formula combining the five, then the complexity is reduced to $5 \cdot O(2^8) + O(2^5)$.

2. **Memoization:** We modify our algorithm so that whenever any confidence value is computed we "memoize" it, i.e., we store it so that it can be used in other confidence computations. Without our new technique of identifying independent tuples during confidence computation, previously-computed values cannot be reused since we always expand to the base data. However, suppose in our example that t_3's confidence value had been computed before. Then instead of calling *Conf* on t_3, we simply use its existing confidence value.

When can we break up our lineage formulas as in the example above? That is, which tuples t during expansion permit us to call *Conf* on them, instead of expanding their lineage in-place? Intuitively, during expansion we can treat as independent any tuple whose lineage is disjoint within t's fully-expanded lineage. For now, let us assume a function *Indep*(t) that takes a tuple t and returns a set of tuples $\{t_1, \ldots, t_n\}$ in t's expanded lineage that are all independent. Further assume that this set is the "closest to t", so that we break the expansion as early as possible. (In the very worst case, when all lineage is interdependent, $\{t_1, \ldots, t_n\}$ are just the base tuples in t's fully-expanded lineage.) We will discuss the implementation of function *Indep* shortly, in Section 4.1.

The improvements are specified in Algorithm 2, with the following changes from Algorithm 1. We compute in line 4 the tuples at which we will break up confidence computation and pass them to function *Compute-Conf*. In *Compute-Conf*, we replace our test for base tuples with a test for independence, and we call *Conf* recursively to obtain their confidence values. Once a confidence value has been computed for a tuple t_i, we store it permanently in a variable we refer to as c_i (line 12). Although not shown in the algorithm, we also store the final confidence value for t.

4.1 Finding *Indep*(t)

Now let us discuss computing the set *Indep*(t) that we need in our improved algorithm. We first create a graph that encodes the recursive structure of t's lineage, as in Figure 3 except we can omit the logical operators comprising the lineage formula. That is, the root is t, its children are the tuples $\{t_1, \ldots, t_n\} \in \lambda(t)$, t_1's children are the tuples in $\lambda(t_1)$, and so on. The leaves are base tuples. We call this graph *Lin*(t). Recall that lineage is always acyclic, so *Lin*(t) is a DAG.

We now define *Indep*(t) based on the structure of *Lin*(t). Note that we interchangeably refer to nodes in *Lin*(t) and the tuples they correspond to.

DEFINITION 4.1. *Indep*(t) is the unique set of tuples $t_i \neq t$ in *Lin*(t) satisfying the following two conditions:

1. (Disjoint lineage) None of t_i's descendants can be reached from any other node t_j outside of t_i's subtree, without passing through t_i.

2. (Closeness to t) No ancestor of t_i, except t, satisfies Condition 1. □

Clearly, any two tuples in *Indep*(t) have independent confidence values: they do not share any base data in their lineage (Condition 1). Furthermore, there are no "better" tuples by which to break our lineage expansion, because no tuples closer to t satisfy the independence property (Condition 2).

Now consider actually computing *Indep*(t) for any t. Regardless we must build the entire graph *Lin*(t), although it can be built just once: when we have a recursive call to *Indep*(t') from *Conf*, *Lin*(t') is a subgraph of the top-level *Lin*(t). For a given t, *Indep*(t) can be computed in one pass of *Lin*(t) as shown in Algorithm 3. The algorithm maintains four global arrays indexed by nodes in *Lin*(t): S_{in}, S_{out}, *Computed*, and I. $S_{in}[t]$ is the set of all nodes in the subtree rooted at t. $S_{out}[t]$ is the set of all nodes outside $S_{in}[t]$ with an edge into $(S_{in}[t] - \{t\})$. *Computed*[t], initialized to false, says whether *Check-Indep* has been called for t previously. $I[t]$ stores the return value of *Check-Indep*(t). We assume (in lines 4, 14, and 17) that, given a node in the graph, we can quickly find its parents and children.

Although *Lin*(t) may sometimes be small in practice so building the graph and computing *Indep*(t) is not too expensive, we next introduce an alternative, schema-level approach that allows us to approximate *Indep*(t) for any t. Instead of creating *Lin*(t) for individual tuples when their confidence is computed, we maintain a single schema-level lineage graph for the entire database. We then

```
1: Input: Lin(t) = G(V,E)
2: Global Arrays: S_in, S_out, Computed, I
3: Indep(t) {
4:     Let c_1, ..., c_n be the children of t;
5:     Return (Check-Indep(c_1) ∪ ... ∪ Check-Indep(c_n));}
6: Check-Indep(c) {
7:     if is-leaf(c) then {
8:         Computed[c] = true;
9:         S_in[c] = {c}
10:        S_out[c] = ∅;
11:        Return {c};}
12:    else {
13:        Computed[c] = true;
14:        Let c_1, ..., c_n be the children of c;
15:        ∀c_i: if !Computed[c_i] then I[c_i] = Check-Indep(c_i);
16:        S_in[c] = c ∪ S_in[c_1] ∪ ... ∪ S_in[c_1];
17:        S_out[c] = (S_out[c_1] ∪ ... ∪ S_out[c_n] ∪
                      {t' ∈ V | ∃c_i, (t', c_i) ∈ E}) − S_in(c);
18:        if S_out[c] == ∅ then return {c}
19:        else return (I[c_1] ∪ ... ∪ I[c_n]); } }
```

Algorithm 3: Computing *Indep*(t)

similarly compute break points for our expansion on the schema-level graph. The break points are always *correct*, in that their subtrees are independent. However, they may be *conservative*, in that it may have been possible to break earlier had we explored data-level lineage.

4.2 Approximating *Indep*(t)

We maintain a schema-level DAG *SLin* for the entire database, capturing relation-level lineage structure. First we define *SLin* and explain how it is maintained. Then we show how *SLin* can be used to replace *Indep*(t) in Algorithm 2.

To define *SLin*, we first logically label the relations in the database: *IND* relations are those whose tuples are always mutually independent, i.e., two different tuples can never have common base data in their lineage; *DEP* relations are those whose tuples may share base data in their lineage. All base relations are *IND*. Any query with a *DEP* relation as input results in a *DEP* relation. Query results over purely *IND* relations are *IND* if no two result tuples may refer to the same input tuple. In our query language, conservatively any query including $\bowtie$, $\cap$, $\cup$, $-$, δ, or α produces a *DEP* result. (I.e., we generally assume only Π-σ queries are guaranteed to produce an *IND* result. Analysis involving the specific query and/or data may identify more queries with *IND* results. For instance, a key-key join over two *IND* relations results in an *IND* relation.)

SLin contains one node for each *IND* relation, and one or more nodes for each *DEP* relation. Consider a database D consisting of base relations $S_1, \ldots, S_n$, and a sequence of queries Q_1, Q_2, Q_3, ..., generating result relations R_1, R_2, R_3, ... Initially *SLin* has n nodes corresponding to $S_1, \ldots, S_n$ (which recall are *IND*) and no edges. A query $Q_i(S_1, \ldots, S_n, R_1, \ldots, R_{i-1})$ generating result relation R_i modifies *SLin* as follows:

1. Add a new node R_i to *SLin*.
2. For every *IND* input relation T, add an edge from R_i to T.
3. For a *DEP* input relation T that is currently a root in *SLin*, check if tuples in R_i can have lineage referring to multiple tuples in T. A tuple in R_i may refer to multiple tuples in T if the expression tree for Q_i either has two paths from R_i to T, or one path including operator δ, $-$, $\cap$, or α.
   a. If no, add an edge from R_i to T.
   b. If yes, consider the subgraph G' of *SLin* with T as the root and including all paths from T to *IND* relations. Add a copy of G' to *SLin*, and add two edges from R_i to the two copies of T.
4. For a *DEP* input relation T that is not a root, consider subgraph G' of *SLin* induced with T as the root and including all paths from T to *IND* relations. As in Step 3, check if tuples in R_i can have lineage referring to multiple tuples in T.
   a. If no, make one copy of G' and add an edge from R_i to T in G'.
   b. If yes, make two copies of G' and add two edges, one edge from R_i to T in each copy of G'.

EXAMPLE 4.2. Consider all three of our example queries from Section 2, in sequence. Now we explicitly apply δ for duplicate-elimination.

Q1: EventRoster = $\Pi_{\text{person,event}}$(Attends $\bowtie$ Events)
Q2: EventAttendees = $\delta(\Pi_{\text{person}}$(EventRoster))
Q3: EventLikelihood = $\delta(\Pi_{\text{event}}$ (Attends $\bowtie$ Events))

Figure 4 shows *SLin* after each query. We use A, E, ER, EA, and EL to abbreviate the relation names. The base relations Attends and Events are *IND*. All derived result relations are *DEP*. Step 3(b) of *SLin* creation applies in Q2 because of the duplicate-elimination: tuples in EventAttendees can refer to multiple tuples in EventRoster. □

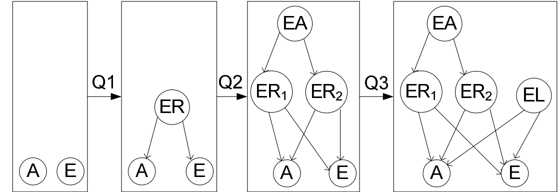

Figure 4: *SLin* for a sequence of queries

SLin encodes relation-level dependencies, with dependencies of tuples within a single relation encoded as multiple nodes for that relation. For a tuple t in a derived relation R, *Indep*(R) on *SLin* gives a set of independent relations containing tuples in the (partial) expansion of t's lineage. That is, expanding the lineage of any tuple in R up to the relations in *Indep*(R) gives a set of independent tuples. Using *SLin* may be conservative: For some tuples in R, even before expanding all the way to *Indep*(R) we may obtain a set of independent tuples. Thus, using *SLin* can be thought of as an efficient approximation for *Indep*(t).

Now we explain how the use of *SLin* modifies Algorithm 2. Suppose we call *Conf*(t), and t is in relation R. We compute *Indep*(R), then expand t's lineage until it reaches tuples in *Indep*(R), at which point we break-up the computation. As before, confidence of tuples in *Indep*(R) are computed recursively, then combined using *Eval*. The only two lines from Algorithm 2 that need to be modified are 4 and 10, changing *Indep*(t) and t_i to *Indep*(*Rel*(t)) and *Rel*(t_i),

respectively, where *Rel* identifies a tuple's relation. Analogous to using *Lin*(t) to find *Indep*(t), here *Indep*(R) is computed for the subgraph of *SLin* rooted at R.

SLin is maintained incrementally as queries are performed, and retained for the life of the database. Since *Indep*(R) information may be requested repeatedly for multiple tuples in the same relation R, we maintain a lookup table containing the relation list in *Indep*(R) for each R.

5. BATCH COMPUTATION

We now turn to computing the confidence values for an entire relation or database at once. Suppose we want to compute confidence values for all tuples in a relation R. We recursively generate SQL queries to compute confidence values for each relation in *Indep*(R). Then we generate an outer SQL query that includes the recursively-generated subqueries, to compute all confidences for R at once. The final SQL query can be optimized by the DBMS. Note that in this approach, confidences are computed only for those tuples in *Indep*(R) that appear in the lineage of some tuple in R.

To formulate a valid SQL query for batch confidence computation, we need the lineage from R to *Indep*(R) to satisfy a certain structure, as follows. First, each relation R_i on a path from R to *Indep*(R) (including R itself) must have "uniform" lineage to its input relations $S_1, \ldots, S_n$: The lineage formula of each tuple in R_i must be a conjunction or disjunction of n clauses, where the jth clause is in turn a conjunction or disjunction of tuples in S_j. This condition can be checked at query time, with nodes in *SLin* annotated as "uniform" or "non-uniform". For example, a query involving only σ, π, $\bowtie$, and $\cup$ satisfies the condition, as does a query that performs $\cap$ of two relations, or one that applies δ to a single relation. Second, if a relation S_j appears in a disjunctive clause in R_i's lineage, then there must be no sibling edge to (R_i, S_j), i.e., all paths from R to a descendant of S_j pass through the edge (R_i, S_j). We further annotate *Slin*, adding an "OR" label to each such edge.

As mentioned above, these restrictions are needed in order to generate valid SQL queries. When the conditions do not hold and we want to compute confidence values for all tuples in a relation R, we must compute them tuple-by-tuple. Broadening the class of query results whose confidence computation can be batched is a subject of future work, probably involving stored procedures in addition to standard SQL.

Next we provide an informal description of the single-relation batch algorithm (Section 5.1), and explain how it is used to compute confidences for an entire database (Section 5.2).

5.1 Relation-Level

Let $R(A_1, ..., A_n)$ be a ULDB relation. We assume the data portion of R is stored as a conventional relation (which we also call R) with two additional attributes: $R(\mathtt{id}, \mathtt{conf}, A_1, \ldots, A_n)$. Each tuple has a globally unique identifier `id` and a `conf` value that stores the confidence of the tuple, which is `NULL` if it has not been computed. We assume that for each derived relation R there is a separate *lineage table lin_R*`(id1, table, id2)`, whose entries denote that R's tuple `id1` includes tuple `id2` from table `table` in its lineage. Note that this relation does not capture lineage as boolean formulas. However, together with *SLin*, it is sufficient to perform batch confidence computation for the class of query results described above. These structures are similar to the encoding used in the Trio prototype [24].

Assume R, whose confidences we want to compute, has base relations $\{S_1, S_2, \ldots, S_n\}$ in its (transitive) lineage. We will extend our algorithm to use *Indep*(R), but for now let us assume we want to use the base relations to compute R's confidences. For each base

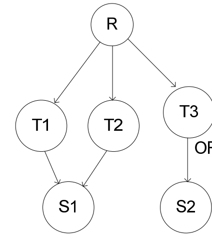

Figure 5: *SLin* for Example 5.1

relation S_i, we enumerate all paths from R to S_i, joining the lineage tables in each path to obtain the set of all tuples from S_i that contribute to tuples in R. For a particular tuple t in R, if all tuples in S_i contribute to t either conjunctively or disjunctively, we can combine their confidences to obtain the contribution of S_i to t. We then combine the contributions of each S_i to obtain the final confidence value for t. In SQL, we compute the contribution of each S_i to R in a subquery that consists of the union of joins over all distinct paths from R to S_i, and then combine their contributions in an enclosing query.

EXAMPLE 5.1. Consider the abstract *SLin* shown in Figure 5. $T1$ and $T2$'s lineage is conjunctive over $S1$ (possibly from selections), and $T3$'s lineage is disjunctive over $S2$ (possibly from duplicate-elimination). R's lineage is conjunctive over $T1$, $T2$, and $T3$ (possibly from a join). Suppose we want to compute the confidence of all tuples in R. All derived relations are "uniform", thus our conditions for batch computation are satisfied.

For each tuple t, we multiply the confidence of all tuples from $S1$ in t's lineage, and further multiply the result with the confidence of the disjunction of all tuples from $S2$ in t's lineage. The exact SQL query is shown in Figure 6. The subqueries producing $C1$ and $C2$ compute the confidence contribution due to $S1$ and $S2$ respectively. To obtain the confidence due to $S1$, we union the results of the two lineage paths: R through $T1$ to $S1$, and R through $T2$ to $S1$. We then multiply the confidences of the resulting $S1$ tuples. To obtain the confidence due to $S2$, we follow the single lineage path. Here, because of disjunction we combine the confidences c_i using $(1 - \Pi_i(1 - c_i))$. Since multiplication is not a built-in SQL aggregation function, in the query we obtain products by taking the exponent over the sum of the log-transformed confidences. Finally, note that we join $C1$ and $C2$ on their `id` attributes, assuming every tuple in R has lineage in both the components (as would be the case for a join). If this is not the case, we can apply an outer join instead of the natural join. □

The example above traverses lineage to base data. However, just like in tuple-level confidence computation, we can make use of independence and memoization. Furthermore, we can incorporate the improvement into our single SQL query. To compute R's confidence values, we first generate the batch query Q over relations in *Indep*(R). Then for relations in *Indep*(R) whose confidences have not already been computed, we recursively generate batch queries and embed them as subqueries in Q.

It is possible to combine batch and tuple-level computation when some but not all portions of the relevant lineage satisfy the requirements for batch computation. It is also possible in some cases to fold batch confidence computation into the original query that produces R, if all confidences are desired immediately. Details of these extensions are omitted due to space constraints.

Finally, suppose we want to compute confidences for a specific subset of the tuples in a relation. We can either compute the con-

```
SELECT C1.id, (C1.conf * C2.conf) AS conf
FROM
  (SELECT D1.id AS id,
     exp(sum(ln(D1.conf))) AS conf
   FROM
     (SELECT lin_R.id1 AS id, conf
      FROM  lin_R, lin_T1, S1
      WHERE lin_R.id2 = lin_T1.id1
        AND lin_R.table = 'T1'
        AND lin_T1.id2 = S1.id
        AND lin_T1.table = 'S1'
      UNION
      SELECT lin_R.id1 AS id, conf
      FROM lin_R, lin_T2, S1
      WHERE lin_R.id2 = lin_T2.id1
        AND lin_R.table = 'T2'
        AND lin_T2.id2 = S1.id
        AND lin_T2.table = 'S1'
     ) AS D1 GROUP BY D1.id) AS C1,
  (SELECT D2.id AS id,
     (1-exp(sum(ln((1-D2.conf))))) AS conf
   FROM (
     SELECT lin_R.id1 AS id, conf
     FROM  lin_R, lin_T3, S2
     WHERE lin_R.id2 = lin_T3.id1
       AND lin_R.table = 'T3'
       AND lin_T3.id2 = S2.id
       AND lin_T3.table = 'S2') AS D2
   GROUP BY D2.id
  ) AS C2
WHERE C1.id = C2.id;
```

Figure 6: Batch confidence computation query

fidence for each tuple individually, or we can perform batch confidence computation for the entire relation. Either approach could be more efficient, depending on the number of tuples whose confidence values are requested and the database characteristics. We study the crossover point between tuple-level and batch computation experimentally in Section 7.

5.2 Database-Level

Suppose we wish to compute the confidences for an entire database. Our technique is to apply batch computations to one relation at a time (or tuple-level as needed), in an order that allows us to use memoized results as much as possible. We traverse *SLin* in reverse topological order—i.e., a result relation appears in the sorted order only after all its input relations—computing the confidences of each relation as we go. During the computation of R, the confidences for each relation in *Indep*(R) have always previously been computed.

6. EXTENSIONS FOR ULDB'S

We now consider the more general ULDB model implemented in the Trio system [24]. Trio relations may include tuple *alternatives*, in addition to the forms of uncertainty we have already seen. Confidence values are attached to alternatives and sum to ≤ 1; lineage is captured at the level of alternatives. (Formal details can be found in [4]. Note that the model we have been considering so far is a special case of the general model.) In our running example, if `Ullman` attends the conference on Monday with 60% confidence or on Tuesday with 30% confidence, but not on both days, we have:

Attends(person,day)
`(Ullman,Monday):0.6 ‖ (Ullman,Tuesday):0.3`

In the next Section 6.1 we extend our confidence computation approach to encompass the more general ULDB model, and in Section 6.2 we discuss how additional features of the Trio system can exploit simple variations on the techniques introduced in this paper.

6.1 Confidence Computation

To encompass tuple alternatives, we begin by simply replacing "tuple" in all of our definitions, data structures, and algorithms with "alternative". (We assume globally unique identifiers for each alternative, as we required for tuples.) The only significant issue that arises is the introduction of a second kind of non-independence: multiple alternatives of the same tuple are mutually exclusive, i.e., they cannot coexist in any possible instance of the database. Thus, we cannot treat alternatives of the same tuple in a lineage formula as independent, even when they are base data. Our approach is to initially treat alternatives as if they are separate tuples, then modify lineage formulas before calling *Eval* to account for variables that correspond to alternatives from the same tuple. Note that optimizing via *Indep* is not affected by these modifications.

Consider a lineage formula f containing variables $a_1, a_2, \ldots, a_n$ that represent n alternatives of of the same tuple. Let the confidences on these alternatives be $c_1, c_2, \ldots, c_n$. We use the following Lemma 6.1, proved in the online technical report [15], to replace in f all occurrences of each a_i with a new variable v_i and a new confidence value for v_i. We perform this replacement for every set of multiple alternatives in f, to obtain a new lineage formula f'. We then apply *Eval* to f' instead of f, since all variables in f' are independent.

LEMMA 6.1. Let f be a formula in which variables $a_1, a_2, \ldots, a_n$ are mutually exclusive and have confidences $c_1, c_2, \ldots, c_n$ respectively. Introduce n new independent variables $v_1, v_2, \ldots, v_n$, with confidences given by:

$$c(v_i) = \frac{c_i}{(\sum_{j=i}^{j=n} c_j + \delta)}, \text{ where } \delta = 1 - \sum_{k=1}^{n} c_k$$

In f, replace a_1 with v_1, and replace each a_i, $i \geq 2$, with $(\neg v_1 \wedge \cdots \wedge \neg v_{i-1} \wedge v_i)$. The probability of the resulting boolean formula is equal to that of f. □

When lineage structures are "uniform", as required by our batch algorithms for example (recall Section 5), the situation is simpler: If multiple alternatives of the same tuple form a conjunct then its probability is zero (since the alternatives cannot coexist); if they form a disjunct, its probability is the sum of the confidence values (since they cannot co-occur). These checks and computations can be incorporated into the SQL queries for batch computation; details are omitted.

6.2 Other Trio Features

In addition to computing confidence values on demand, the Trio system supports *extraneous data detection* and *coexistence checks* [24], also invoked by the user. Both features are closely related to confidence computation and can adapt the techniques of this paper. Note that Trio relations need not include confidence values. In such relations, *maybe-tuples*, denoted by a "?" annotation, correspond to tuples whose total confidence across alternatives is < 1, i.e., tuples whose presence is uncertain [4].

- **Extraneous data detection:** A ULDB alternative is *extraneous* if it cannot occur in any possible instance of the database. Conversely, in a ULDB relation without confidences, a "?" annotation is extraneous when the annotated tuple is guaranteed to appear in every possible instance. It is easy to see informally that extraneous alternatives are those whose

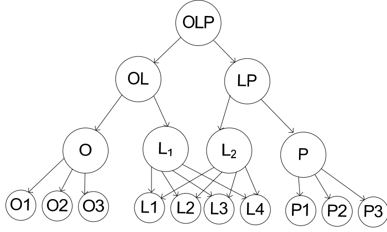

Figure 7: *SLin* for the TPC-H schema

expanded lineage formula is unsatisfiable, i.e., whose computed confidence is zero (under the assumption that no base data has zero confidence). Conversely, a "?" annotation on a tuple t is extraneous if the disjunction f of the lineage formulas for t's alternatives is always satisfied, i.e., f's probability is 1. These tests can be performed using straightforward variations on the algorithms in this paper, including optimizations and batch computations.

- **Coexistence checks:** Two ULDB alternatives a_1 and a_2 *can coexist* if some possible instance of the database contains both of them. Coexistence can be checked by creating a "dummy" alternative a whose lineage is $a_1 \wedge a_2$. If a's expanded lineage formula is satisfiable (i.e., its confidence is $>$ 0), then a_1 and a_2 can coexist. Here too, techniques from this paper can be applied directly. Similarly, we can test whether two alternatives a_1 and a_2 are guaranteed to always coexist: every possible instance of the database that contains one of them also contains the other. We again create a dummy alternative a whose lineage in this case is $(a_1 \wedge \neg a_2) \vee (\neg a_1 \wedge a_2)$. Here we check if a's lineage is unsatisfiable, i.e., whether a's confidence is zero.

7. EXPERIMENTS

All algorithms in this paper have been implemented in the Trio system [33]. The experiments presented here were conducted with Trio running on a Quad-Xeon server with 8 GB RAM and a large SCSI RAID. Trio uses the *PostgreSQL* open-source DBMS as its relational back-end [24]. Trio encodes ULDB relations in standard relational tables, and lineage for each derived ULDB relation is stored in its own table. For details see [24]. We report wall-clock query execution times with a warm cache, i.e., by running a query once and then reporting the average runtime over three subsequent, identical executions.

Our dataset is based on TPC-H [32] with a scale-factor of 1. For our experiments, we created a DAG structure of derived relations starting with the three largest TPC-H relations: `Orders`, `Lineitems`, and `Partsupps`. First, we partitioned these relations into $\texttt{O}_i$ ($i = 1..3$), $\texttt{L}_i$ ($i = 1..4$), and $\texttt{P}_i$ ($i = 1..3$). Then, we converted each of the partitions into a Trio relation (without alternatives) by adding confidence values. Since the computations performed by all of our algorithms are independent of specific confidence values, the distributions of confidence values were not varied in our experiments.

Three derived relations, `O`, `L`, and `P`, are obtained by joining the corresponding partitioned base relations. Then, `L` and `P` are joined to produce derived relation `LP`; `O` and `L` are joined to produce `OL`; and finally `LP` and `OL` are joined to produce `OLP`. Figure 7 depicts the *SLin* (recall Section 4.2) after all queries have been executed. Notice in particular that *Indep*(`OLP`) is the set of relations $\{$`O`, $\texttt{L}_1$–$\texttt{L}_4$, `P`$\}$.

We experimented with two partitioning schemes for the $\texttt{O}_i$'s, $\texttt{L}_i$'s, and $\texttt{P}_i$'s: In one, a vertical partitioning of `Orders`, `Lineitems`, and `Partsupps` creates a different base relation for each attribute; in the other, a horizontal partitioning splits the original relations into a number of similar base relations. Table 1 shows the sizes of the base and derived relations for both partitioning schemes, in millions of tuples.

TPC-H Horizontal Split

Relation	Data	Lineage
$\texttt{O}_1$–$\texttt{O}_3$	0.50	NA
$\texttt{L}_1$–$\texttt{L}_4$	1.50	NA
$\texttt{P}_1$–$\texttt{P}_3$	0.27	NA
O	1.10	3.29
L	0.68	2.70
P	0.77	2.32
OL	1.52	3.04
LP	0.08	0.17
OLP	1.54	3.07
Total	13.98	14.58

TPC-H Vertical Split

Relation	Data	Lineage
$\texttt{O}_1$–$\texttt{O}_3$	1.50	NA
$\texttt{L}_1$–$\texttt{L}_4$	6.00	NA
$\texttt{P}_1$–$\texttt{P}_3$	0.80	NA
O	1.50	4.50
L	6.00	24.00
P	0.80	2.40
OL	6.00	12.00
LP	6.00	12.00
OLP	7.61	15.22
Total	58.82	70.13

Table 1: Relation sizes in millions of tuples

7.1 Benefit of *Indep* and Memoization

Our first set of experiments measures the performance gained by exploiting independence and memoization. These experiments were performed using our batch algorithms, where memoization is exploited within one (batch) confidence computation. (Otherwise, we would have needed to synthesize a series of tuple-level confidence computations with shared lineage to see the benefits.) We measured confidence computation time for individual relations as well as the entire database, using the horizontally-partitioned version. We compared the following two algorithms.

1. `Batch-I` (Independent): This is the algorithm from Section 5.2. Batch confidence computation is performed on each relation in *SLin* in topological order with memoization.

2. `Batch-NI` (Non-Independent): Batch confidence computation is performed on each relation R but without exploiting *Indep*(R), i.e., by traversing to the base relations for all computations.

Table 2 shows the execution times for both cases, for each relation as well as the entire database. As expected, for `L`, `O`, and `P`, the execution times were the same. The total execution times for the entire database, consisting of about 14 million tuples, were approximately 72 minutes and 97 minutes for `Batch-I` and `Batch-NI` respectively. The sum of the execution times for `OL`, `LP`, and `OLP`—tables whose execution under `Batch-I` and `Batch-NI` differ—were approximately 57 minutes and 82 minutes, respectively. Thus, we observed roughly a 30% improvement due to memoization. Note that these performance gains strongly depend on the topology of derived relations. Had we generated a test database with much deeper structure, performance gains would grow proportionately.

To determine how our batch algorithm scales with data size, we varied join selectivities to obtain different sizes for `OLP` and then performed relation-level confidence computation on it. For this experiment we used the vertically-partitioned database, since it more readily enables varying selectivities. Table 3 shows that the algorithm's performance is roughly linear in data size. Even for our largest instance of `OLP` with 7.6 million tuples, the batch confidence computation took about 16 minutes. (The apparent discrepancy with Table 2 is due to the vertically-partitioned versus

Table	L	O	P	OL	LP	OLP	Total
Batch-I	370.57	88.15	430.36	359.51	57.49	3,014.49	4320.57
Batch-NI	367.18	83.31	428.19	989.28	158.09	3,771.76	5797.81

Table 2: Batch execution times (in seconds)

horizontally-partitioned database. In our setting, joining vertical partitions is faster than horizontal.)

Selectivity at OLP	1%	7%	30%	100%
Batch-I	44.15	78.08	287.09	995.60

Table 3: Batch-I varying join selectivity (in seconds)

7.2 Per-Tuple versus Batch

Next we consider tuple-level versus batch computation, when confidences for a fraction of tuples in a particular relation are required. We compared Batch-NI against Tuple-NI, the latter applying Algorithm 1 to each tuple. Our goal was to find *crossover* points: fractions of tuples above which it is more efficient to compute confidences for the entire relation using the batch algorithm than to compute individual confidences selectively using the per-tuple algorithm. We used relations LP and OLP over the horizontally-partitioned database for our experiments; results are shown in Figure 8. In both experiments, the batch algorithm became more efficient for relatively small fractions of the relations: 23.4% and 10.87% for LP and OLP respectively.

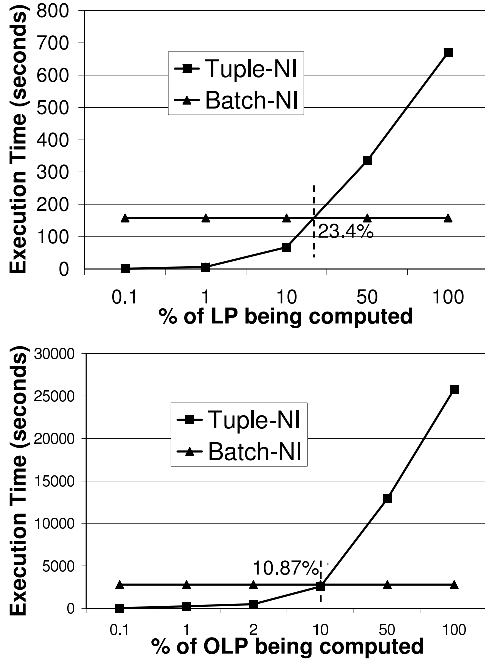

Figure 8: Tuple-level versus batch computation

7.3 Effect of Join Multiplicity

Consider relation-level confidence computation for our sample data, or any derived relation based on joins. The benefit of memoization is directly affected by join multiplicity: If a join is one-one, we compute the confidence value for each joining tuple once, and we use it that one time. If a join is many-one or many-many, we need the confidence values of some input tuples over and over, so we expect significant benefit from memoization. We verified this

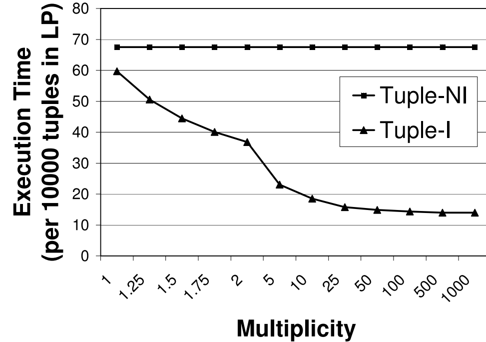

Figure 9: Execution times for varying join multiplicity

intuition by varying the many-one join multiplicities in derived relation LP, in the horizontally-partitioned database. As can be seen in Figure 9, Tuple-NI was unaffected by join multiplicity, since it didn't exploit memoization. However, Tuple-I (which used Algorithm 2) saw significant performance gains over Tuple-NI as multiplicity increased.

8. CONCLUSIONS & FUTURE WORK

We propose decoupling data and confidence computation in databases with uncertainty and lineage. Decoupling provides two benefits: First, it allows data computation to proceed using any plan relationally-equivalent to the query posed, rather than being restricted to plans that propagate confidence values correctly. Second, if not all confidence values are needed by the user or in further queries, the potentially expensive operation of computing confidences can be performed selectively.

After formalizing query processing for a language that includes all standard relational operations along with duplicate-elimination and aggregation, we presented an initial basic algorithm for computing confidences based on lineage. We then improved our algorithm with techniques that identify and exploit independent lineage subtrees, and memoize confidence values as they are computed. Finally we presented algorithms for batch confidence computations, extensions to a more general model for uncertainty, and a suite of experimental results.

A first obvious extension to our work is a cost model that supports deciding between tuple-level and batch computations when a set of confidence values are desired. Far more interesting and ambitious is incorporating our algorithms as one part of a comprehensive query optimization framework for uncertain and probabilistic data. In this setting, the optimizer should use a cost model that reliably compares "safe plans" that compute confidences operator-by-operator (recall Section 2.4) against our approach, which uses the best available plan for data computation and lineage tracking, then computes confidences based on lineage. Such a general-purpose optimizer is one of the long-term goals of the Trio project.

Other avenues of future work include incorporating AI inferencing algorithms (recall Section 1.1) into the general query optimization framework proposed in the previous paragraph, and handling updates to confidence values and perhaps lineage relationships in our setting of persistent derived result relations.

9. REFERENCES

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

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

[3] D. Barbará, H. Garcia-Molina, and D. Porter. The Management of Probabilistic Data. *IEEE Trans. Knowl. Data Eng.*, 4(5), 1992.

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

[5] O. Benjelloun, A. Das Sarma, C. Hayworth, and J. Widom. An Introduction to ULDBs and the Trio System. *IEEE Data Engineering Bulletin*, 29(1), 2006.

[6] D. Bhagwat, L. Chiticariu, W. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *Proc. of VLDB*, 2004.

[7] 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.

[8] P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. In *Proc. of ICDT*, 2001.

[9] R. Cheng, S. Singh, and S. Prabhakar. U-DBMS: A database system for managing constantly-evolving data. In *Proc. of VLDB*, 2005.

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

[11] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB Journal*, 12(1), 2003.

[12] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2), 2000.

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

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

[15] A. Das Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. Technical report, Stanford InfoLab, March 2007. Available on http://dbpubs.stanford.edu.

[16] N. Fuhr. A Probabilistic Framework for Vague Queries and Imprecise Information in Databases. In *Proc. of VLDB*, 1990.

[17] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM TOIS*, 14(1), 1997.

[18] G. Grahne. Dependency Satisfaction in Databases with Incomplete Information. In *Proc. of VLDB*, 1984.

[19] G. Grahne. Horn Tables - An Efficient Tool for Handling Incomplete Information in Databases. In *Proc. of ACM PODS*, 1989.

[20] T. J. Green and V. Tannen. Models for incomplete and probabilistic information. In *Proc. of IIDB Workshop*, 2006.

[21] T. Imielinski and W. Lipski Jr. Incomplete Information in Relational Databases. *Journal of the ACM*, 31(4), 1984.

[22] R. M. Karp and M. Luby. Monte-Carlo algorithms for enumeration and reliability problems. In *Proc. of FOCS*, 1983.

[23] E. J. McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 1956.

[24] M. Mutsuzaki, M. Theobald, A. Keijer, 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.

[25] W. Tan P. Buneman, S. Khanna. Data provenance: Some basic issues. In *Proc. of FSTTCS*, 2000.

[26] W. Tan P. Buneman, S. Khanna. On propagation of deletions and annotations through views. In *Proc. of ACM PODS*, 2002.

[27] J. Pearl. *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., 1988.

[28] W. Quine. The problem of simplifying truth functions. *American Math Monthly*, 59(1), 1952.

[29] C. Re, N. Dalvi, and D. Suciu. Query evaluation on probabilistic databases. *IEEE Data Eng. Bull.*, 29(1), 2006.

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

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

[32] Transaction Processing Council (TPC). TPC Benchmark H: Standard Specification, 2006. `http://www.tpc.org/tpch`.

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

[34] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. of ICDE*, pages 91–102, 1997.

APPENDIX

A. RELATIONAL OPERATORS AND QUERY PLANS

We consider ULDBs with tuple alternatives as presented in Section 6, specifying algorithms for all relational operators. Tuples without alternatives, as considered in Sections 2–5, are captured as a special case. Algorithms for a considerably restricted set of operators—the *DL-monotonic* ones—were presented in [4]. In our specifications, the emphasis is on readability, particularly for understanding how lineage is generated. Implementations as real physical operators would instead focus on efficiency.

Our algorithms operate on multisets, with set semantics obtained through an explicit duplicate-elimination operator. For readability, we present intersection and difference algorithms here that are neither the pure set nor multiset versions—a set version is obtained by adding duplicate-elimination, while a multiset version requires relatively intricate (but not inherently expensive) counting.

Selection

Consider operator σ applied to relation S with lineage λ_S, producing result R with lineage λ.

1: For each tuple t in S with at least one alternative satisfying the selection predicate, add a tuple to R containing all alternatives in t satisfying the selection predicate.
2: For each alternative a in R corresponding to alternative a_S in S, set $\lambda(a) = \lambda_S(a_S)$

Projection

Consider operator π applied to relation S with lineage λ_S, producing result R with lineage λ.

1: For each tuple in S, add a corresponding tuple to R with each alternative projected onto the specified attribute-list.
2: For each alternative a in R corresponding to alternative a_S in S, set $\lambda(a) = \lambda_S(a_S)$

Join

Consider operator $\bowtie$ applied to S_1 and S_2 with lineage λ_1 and λ_2 respectively, producing result R with lineage λ. An alternative is chosen in R for every pair of alternatives in S_1 and S_2 matching the join condition. For each pair of tuples from S_1 and S_2 with at least one pair of alternatives satisfying the join condition, an alternative is added to R.

1: For each pair of tuples s_1 in S_1 and s_2 in S_2, if at least one pair of alternatives matches the join condition:

1. Construct a tuple t in R, with an alternative a for every pair of alternatives $a_1 \in s_1$ and $a_2 \in s_2$ satisfying the join condition.
2. For each alternative a in t, set $\lambda(a) = \lambda_1(a_1) \wedge \lambda_2(a_2)$

Union

Consider operator $\cup$ applied to S_1 and S_2 with lineage λ_1 and λ_2 respectively, producing result R with lineage λ. All tuples of both relations are copied into R with the appropriate lineage.

1: For each tuple in S_1 (S_2 respectively), add an indentical tuple to R.
2: For each alternative a in R corresponding to alternative a_1 in S_1 (a_2 in S_2 respectively), set $\lambda(a) = \lambda_1(a_1)$ ($= \lambda_2(a_2)$ respectively)

Intersection

Consider operator $\cap$ applied to S_1 and S_2 with lineage λ_1 and λ_2, respectively, producing result R with lineage λ. An alternative can be chosen in R only if alternatives with the same value are chosen in both S_1 and S_2. Note that our algorithm is not symmetric, however the possible instances in $R = S_1 \cap S_2$ and $R = S_2 \cap S_1$ are identical.

1: For each tuple s_1 in S_1:

1. Construct a tuple t in R containing all alternatives of s_1 that appear in S_2. Drop t if no alternatives of r appears in S_2.
2. For each alternative a in t, set
$\lambda(a) = \lambda_1(a_1) \wedge (\lambda_2(a_{2_1}) \vee \lambda_2(a_{2_2}) \vee \cdots \vee \lambda_2(a_{2_m}))$
where a_1 is the alternative in r giving a, and $a_{2_1}, a_{2_2}, \ldots, a_{2_m}$ are all alternatives in S_2 with the same value.

Lineage ensures that an alternative is present in R if the corresponding alternative is present in S_1 and at least one alternative of the same value is present in S_2.

Difference

Consider operator $-$ applied to S_1 and S_2 with lineage λ_1 and λ_2 respectively, producing result $R = S_1 - S_2$ with lineage λ. We use negation in λ to encode the condition that an alternative in R can be chosen only if the corresponding alternative in S_1 is chosen, and no alternative in S_2 with the same value is also chosen. (In the special case where whenever a certain value is chosen in R that same value is also chosen in S, we will get some alternatives in our result with unsatisfiable lineage. These *extraneous* alternatives can be detected and removed as discussed in Section 6.)

1: For each tuple s_1 in S_1:

1. Construct an identical tuple t in R.
2. For each alternative a in t, set
$\lambda(a) = \lambda_1(a_1) \wedge \neg\lambda_2(a_{2_1}) \wedge \neg\lambda_2(a_{2_2}) \wedge \cdots \wedge \neg\lambda_2(a_{2_m})$
where a_1 is the alternative in s_1 giving a, and $a_{2_1}, a_{2_2}, \ldots, a_{2_m}$ are all the alternatives in S_2 that have the same value.

Duplicate Elimination

Consider operator δ applied to relation S with lineage λ_S, producing result R with lineage λ. Disjunctive lineage encodes that a tuple is present in R if any tuple with the same value is present in S.

1: For each alternative a_s in S whose value appears more than once in S, add a single tuple to R with a single alternative a having the same value as a_s.
2: Add to R all tuples in S but without any of the alternatives added in step 1.
3: For each alternative a in R set
$\lambda(a) = \lambda_S(s_1) \vee \lambda_S(s_2) \vee \cdots \vee \lambda_S(s_m)$
where $s_1, s_2, \ldots, s_m$ are all the alternatives in S having the same value as a.

Grouping and Aggregation

Performing aggregations in an uncertain database is known to be a computationally hard problem [4]. For the purposes of this paper, we present a simple algorithm that is correct but clearly inefficient. We are separately exploring the general problem of exact and approximate aggregations in ULDBs.

Consider first aggregation without grouping of a ULDB relation S with lineage λ_S, to produce R with lineage λ. For every combination of alternatives chosen in S, R contains one alternative in its single-tuple result. The lineage of the alternative is the conjunct of the lineage of all the alternatives that were chosen in the combination. We generalize this idea to grouping by introducing one tuple for each possible group in the result.

1: Create a temporarily-empty tuple in R for each possible value of the grouping attributes in S.
2: For each combination of alternatives $s_1, s_2, \ldots, s_m$ for the tuples in S:

   1. Group and aggregate $s_1, s_2, \ldots, s_m$ to produce a temporary (conventional) relation T.
   2. Add each tuple in T as an alternative in the tuple of R corresponding to the values of its grouping attributes.
   3. For each added alternative a, set $\lambda(a) = \lambda_S(s_1) \wedge \lambda_S(s_2) \wedge \cdots \wedge \lambda_S(s_m)$

3: Delete any tuples in R created in step 1 that are still empty.

B. QUERY PLAN EQUIVALENCE

Our proof of Theorem 3.1 is based on the correctness of the operators used in query plans, captured in the following Lemma. We do not provide an exhaustive proof for this Lemma–the correctness of the operators as specified in Appendix A is fairly evident.

LEMMA B.1. Consider a ULDB database D with possible instances $\{D_1, \ldots, D_n\}$, and any operator *op* from Appendix A. Let ULDB relation R be the result of applying *op* to D, and let regular relation R_i be the result of applying *op* to D_i, $i = 1..n$. Then $PI(D + R) = \{D_1 + R_1, \ldots, D_n + R_n\}$. □

The following Lemma says that any plan composed from correct operators is itself correct. We use $P(D_i)$ to denote the result of running a query plan P on a regular database D_i.

LEMMA B.2. Consider a ULDB database D with possible instances $\{D_1, \ldots, D_n\}$, and any query plan P over D producing result relation R. $PI(D+R) = \{D_1+P(D_1), \ldots, D_n+P(D_n)\}$. □

PROOF. We prove the result by induction on the height of the tree of operators comprising P. The base case of a single operator follows from Lemma B.1. Without loss of generality, assume that *op* is a binary operator. (The proof for unary operators is a simpler special-case.) Thus, P consists of an operator *op* applied to two expression trees (subplans) P^1 and P^2.

1. Suppose P has height h. The induction hypothesis (I.H.) says that for any query plan P' with height $< h$, applied on any database E producing result relation S, $PI(E + S) = \{E_1 + P'(E_1), \ldots, E_n + P'(E_m)\}$, where $E_1, \ldots, E_m$ are the possible instances of E.
2. Let us suppose the subplans P^1 and P^2 produce result relations R^1 and R^2 respectively.
3. Applying the I.H. on P^1 and D yielding $D + R^1$, we have: $PI(D + R^1) = \{D_1 + P^1(D_1), \ldots, D_n + P^1(D_n)\}$.
4. Now applying the I.H. on P^2 and $D + R^1$ yielding $D + R^1 + R^2$, we have: $PI(D+R^1+R^2) = \{D_1+P^1(D_1)+P^2(D_1+P^1(D_1)), \ldots, D_n + P^1(D_n) + P^2(D_n + P^1(D_n))\}$.
5. In Step 4, since P^2 operates only on relations in D_i, we have $PI(D + R^1 + R^2) = \{D_1 + P^1(D_1) + P^2(D_1), \ldots, D_n + P^1(D_n) + P^2(D_n)\}$.
6. Now applying Lemma B.1 for *op* on $D + R^1 + R^2$ yielding $D + R^1 + R^2 + R$, we have $PI(D + R^1 + R^2 + R) = \{D_1+P^1(D_1)+P^2(D_1)+op(D_1+P^1(D_1)+P^2(D_1)), \ldots, D_n+P^1(D_n)+P^2(D_n)+op(D_n+P^1(D_n)+P^2(D_n))\}$.
7. In $D_i + P^1(D_i) + P^2(D_i)$ of Step 6, *op* operates only on the result of P^1 and P^2. Therefore, we have: $PI(D + R^1 + R^2 + R) = \{D_1 + P^1(D_1) + P^2(D_1) + op(P^1(D_1) + P^2(D_1)), \ldots, D_n + P^1(D_n) + P^2(D_n) + op(P^1(D_n) + P^2(D_n))\}$.
8. By definition P consists of *op* applied to expression trees P^1 and P^2. Therefore, for any regular database D_i, $op(P^1(D_i)+P^2(D_i)) = P(D_i)$. Using this substitution in Step 7, we have: $PI(D+R^1+R^2+R) = \{D_1+P^1(D_1)+P^2(D_1)+P(D_1), \ldots, D_n + P^1(D_n) + P^2(D_n) + P(D_n)\}$.
9. We now use the *extraction algorithm* from [4] to restrict a ULDB database to a subset of its relations. Here, we restrict the database $D + R^1 + R^2 + R$ to $D + R$.
10. The following Lemma directly follows from the extraction algorithm: For set of relations S and T with $PI(S + T) = \{S_1 + T_1, \ldots, S_n + T_n\}$, if S's lineage doesn't involve T, then we have $PI(S) = \{S_1, \ldots, S_n\}$.
11. Since the lineage of all of R^1, R^2, and R refer only to data in D, using the lemma from Step 10, we have: $PI(D + R) = \{D_1 + P(D_1), \ldots, D_n + P(D_n)\}$

□

Finally, we restate and prove Theorem 3.1.

Theorem 3.1: Consider two relationally-equivalent query plans P_1 and P_2 evaluated over a set of input relations in a ULDB D, yielding ULDB result relations R_1 and R_2. $D + R_1$ and $D + R_2$ have identical possible instances. □

PROOF. By Lemma B.2, P_1 and P_2 satisfy $PI(D + R_1) = \{D_1 + P_1(D_1), \ldots, D_n + P_1(D_n)\}$, and $PI(D + R_2) = \{D_1 + P_2(D_1), \ldots, D_n + P_2(D_n)\}$. Since P_1 and P_2 are relationally equivalent, $P_1(D) = P_2(D)$ for any D. Thus, $PI(D + R_1) = PI(D + R_2)$. □

C. NON-ISOMORPHIC RESULTS FOR EQUIVALENT QUERY PLANS

PROPOSITION C.1. Relationally-equivalent query plans may yield equivalent results that have non-isomorphic ULDB representations. □

PROOF. Consider query plans $P_1(A) = \delta(\delta(R_1) \cup \delta(R_2))$ and $P_2(A) = (((\delta(R_1) - \delta(R_2)) \cup (\delta(R_2) - \delta(R_1))) \cup (\delta(R_1) \cap \delta(R_2)))$. P_1 and P_2 are relationally equivalent. Suppose $R_1(A)$ and $R_2(A)$ both contain a single tuple `[1]` with some confidence $c < 1$, and identifiers `11` and `21` respectively. Then the result of P_1 and P_2 are as shown below.

ID	P1(A)
31	1

λ(31) = 11 ∨ 21

ID	P2(A)
41	1
42	1
43	1

λ(41) = 11 ∧ ¬21
λ(42) = ¬11 ∧ 21
λ(43) = 11 ∧ 21

□

D. EXTENSION TO MULTIPLE ALTERNATIVES

Lemma 6.1: Let f be a formula in which variables $a_1, a_2, \ldots, a_n$ are mutually exclusive and have confidences $c_1, c_2, \ldots, c_n$ respectively. Introduce n new independent variables $v_1, v_2, \ldots, v_n$, with confidences given by:

$$c(v_i) = \frac{c_i}{(\sum_{j=i}^{j=n} c_j + \delta)}, \text{ where } \delta = 1 - \sum_{k=1}^{n} c_k$$

In f, replace a_1 with v_1, and replace each a_i, $i \geq 2$, with $(\neg v_1 \wedge \cdots \wedge \neg v_{i-1} \wedge v_i)$. The probability of the resulting boolean formula is equal to that of f. $\square$

PROOF. The sum of confidences of alternatives other than a_1, ..., a_n is given by δ. Since at most one of the alternatives a_i is true, choosing an alternative can be thought of as up to n independent tosses of a biased coin in the following fashion: The ith coin toss has probability c_i of landing heads. Coins are tossed until a head is obtained. The jth coin landing heads is equivalent to picking the alternative corresponding to a_j. If all coin tosses land tails, it is equivalent to not picking any alternative.

The probability of a head in the ith toss, c_i, is obtained by scaling up the confidence of alternative a_i based on the sum of confidences of the remaining alternatives. Note that if $\sum_{i=1}^{n} c_i = 1$, the nth toss has probability 1 of landing heads. $\square$