The VLDB Journal (2008) 17:243–264
DOI 10.1007/s00778-007-0080-z

SPECIAL ISSUE PAPER

Databases with uncertainty and lineage

Omar Benjelloun · Anish Das Sarma · Alon Halevy · Martin Theobald · Jennifer Widom

Received: 20 February 2007 / Revised: 16 July 2007 / Accepted: 16 August 2007 / Published online: 16 January 2008

Abstract This paper introduces ULDBs, an extension of relational databases with simple yet expressive constructs for representing and manipulating both *lineage* and *uncertainty*. Uncertain data and data lineage are two important areas of data management that have been considered extensively in isolation, however many applications require the features in tandem. Fundamentally, lineage enables simple and consistent representation of uncertain data, it correlates uncertainty in query results with uncertainty in the input data, and query processing with lineage and uncertainty together presents computational benefits over treating them separately. We show that the ULDB representation is *complete*, and that it permits straightforward implementation of many relational operations. We define two notions of ULDB minimality—*data-minimal* and *lineage-minimal*—and study minimization of ULDB representations under both notions. With lineage, derived relations are no longer self-contained: their uncertainty depends on uncertainty in the base data. We provide an algorithm for the new operation of extracting a database subset in the presence of interconnected uncertainty. We also show how ULDBs enable a new approach to query processing in probabilistic databases. Finally, we describe the current state of the *Trio* system, our implementation of ULDBs under development at Stanford.

Keywords Uncertainty in Databases · Lineage · Provenance · Probabilistic data management

This work was supported by the National Science Foundation under grants IIS-0324431, IIS-1098447, and IIS-9985114, by DARPA Contract #03-000225, and by a grant from the Boeing Corporation.

O. Benjelloun (✉) · A. Halevy
Google Inc., Mountain View, USA
e-mail: benjello@google.com

A. Halevy
e-mail: halevy@google.com

A. Das Sarma · M. Theobald · J. Widom
Stanford University, Palo Alto, USA
e-mail: anish@cs.stanford.edu

M. Theobald
e-mail: theobald@stanford.edu

J. Widom
e-mail: widom@cs.stanford.edu

1 Introduction

The problems faced when managing *uncertain data*, and those associated with tracking *data lineage*, have been addressed in isolation in the past (e.g., [2,4,21,27,30,31, 34,41,45] for uncertain data and [11,17–19,39,40] for data lineage). Motivated by a diverse set of applications including data integration, deduplication, scientific data management, information extraction, and others, we became interested in the combination of uncertainty and lineage as the basis for a new type of data management system [46].

Intuitively, an uncertain database is one that represents multiple *possible instances*, each corresponding to a single possible state of the database. Lineage identifies a data item's *derivation*, in terms of other data in the database, or outside data sources. One relationship between uncertainty and lineage is that lineage can be used for understanding and resolving uncertainty. To draw a loose analogy with web search, answers returned by a search engine are uncertain, reflected by their ranking. Search engines typically provide lineage information including at least a URL and text snippet, and users tend to consider both ranking and lineage to determine which links to follow. More generally, any application that integrates information from multiple sources may be uncertain about which data is correct, and the original source and derivation of data may offer helpful additional information.

Lineage is also important for uncertainty within a single database. When users pose queries against uncertain data, the results are uncertain too. Lineage facilitates the correlation and coordination of uncertainty in query results with uncertainty in the input data. For example, suppose we know that either one set of base data is correct or another one is, but not both. Then we don't want to produce any query results that are derived by mixing data from the two sets, directly or indirectly, now or later. Lineage is a particularly convenient and intuitive mechanism for encoding the complex uncertainty relationships that can arise among base and derived data.

Beyond the conceptual relationships between uncertainty and lineage, this paper presents several tangible representational and computational benefits derived from their combination. We begin by describing a representation for uncertainty and lineage that extends the relational model with *tuple alternatives* (a set of possible values for each tuple), *maybe tuples* (tuples that may be present or absent), and a *lineage function* mapping tuple alternatives to the data from which they were derived. We call databases in this scheme ULDBs, for *Uncertainty-Lineage Databases*. We show that because we represent lineage along with uncertainty, ULDBs are *complete*, i.e., they can represent all finite sets of possible instances. In contrast, complete models for uncertainty without lineage are more complex, e.g., [23,31].

Next, we study problems related to querying ULDBs. First, we show that ULDBs permit straightforward and efficient implementation of many relational operations. We then consider the problem of extracting one or more relations from a ULDB: creating a "projection" of a ULDB onto a subset of its relations, without changing the possible instances of the relations. Extracting relations is tricky because when data in a relation R is derived from data in R', then the possible instances of R may correlate with the possible instances of R' (even when R' is not included in the projection), which may in turn correlate with possible instances of other relations. Finally, for both querying and extraction we are interested in operating on ULDBs that satisfy some notions of minimality. We define *data minimality* and *lineage minimality* of ULDBs, and we present results on minimizing ULDBs.

ULDBs also open up an interesting alternative approach to query processing in *probabilistic databases*, which are captured by a simple extension of basic ULDBs to include *confidence values*. Previous work [21] suggests special techniques for constructing query plans that ensure correctness for probabilistic data. It turns out that when lineage is tracked, special considerations are no longer needed: query execution initially proceeds without computing probabilities, so any query plan may be used. Probabilities are then computed from lineage as needed in a separate step.

Uncertainty, lineage and data integration

As discussed in [46], uncertainty and lineage are important for many classes of applications. In this paper we highlight the need for modeling uncertainty and lineage in the context of data integration systems, which are systems that offer a uniform interface to a multitude of data sources.

In data integration applications, uncertainty arises in different ways, but they can all be traced to the "subjectivity" effect. Data sources designed by different people or organizations will, by nature, describe the same domain in different, sometimes inconsistent ways. Differences may arise in the aspects of the domain they choose to model, the schemas they design for the domain (table structure and attribute names), and in the naming conventions they use for data objects. As data integration applications strive to offer a single "objective" and coherent integrated view of data sources, uncertainty is bound to appear. Furthermore, since many data integration applications are based on structure that is automatically extracted from unstructured data, there may even be uncertainty about the data itself, since the extraction techniques are approximate at best. As a result, we have at least three kinds of uncertainty in data integration: the data, the mappings between the schemas and the mappings between data objects in different sources. We emphasize that these types of uncertainty are pervasive especially in data integration applications whose goal is to offer access to a large number of sources on the WWW (e.g., MetaQuerier [13], and PayGo [37]).

The explicit modeling of data lineage essentially acknowledges that data comes from somewhere, and that we are not sure about the transformations (such as those outlined above) that brought data into the database nor about its intrinsic meaning. As uncertainty-generating data integration operations are performed, lineage keeps track of the origins of data, thereby giving a powerful tool to manage, explain and potentially correct the uncertain truth resulting from the integration process.

Throughout the paper, we will illustrate how the key aspects of ULDBs are useful for data integration applications, and also point out a few directions for future research in the application of uncertainty and lineage to data integration.

In summary, this paper makes the following contributions:

- We define ULDBs—uncertain databases with lineage—and show that they are complete (Sect. 3).
- We give algorithms for relational operations in ULDBs (Sect. 4.2).
- We define data-minimality and lineage-minimality for ULDBs, and discuss both types of minimization (Sect. 4.3).
- We define the new problem of extracting data from a ULDB, and we present an algorithm for it (Sect. 4.4).

- We describe how ULDBs can be extended with confidence values, and we show how they offer an alternative solution to query processing in probabilistic databases (Sect. 5).
- We describe our Trio system for ULDBs, and give a detailed account of how the above features are implemented on top of a conventional relational DBMS (Sect. 6).

We discuss related work in Sect. 7 and conclude with future directions in Sect. 8.

This paper is an extension of an earlier conference paper [6]. This paper provides proofs for numerous theorems that did not appear in [6]. It also describes the implementation of ULDBs in a system development effort underway, called *Trio*. Specifically, we describe how Trio encodes ULDBs in a relational database, how queries over ULDBs are translated into sets of SQL queries, and how computation of confidences is optimized in our system. Some of these aspects of Trio were also a topic of a system demonstration [38].

2 Preliminaries

We begin by describing databases with lineage, which we call LDBs, and then we describe uncertain databases. In Sect. 3 we present ULDBs, which combine the two formalisms.

LDBs and ULDBs extend the relational model. A database D is comprised of a set of relations $\bar{R} = R_1, \ldots, R_n$, where each R_i is a multiset of tuples. We attach a unique identifier to each tuple in the database, and $I(\bar{R})$ denotes all identifiers in relations $R_1, \ldots, R_n$.

2.1 Databases with lineage

In the terminology of [11], in LDBs we focus on "where lineage": the lineage of a tuple identifies the data from which it was derived. Some tuples in an LDB are derived from other LDB tuples, e.g., as a result of queries. The lineage of derived tuples consists of references to other tuples in the LDB, via their unique identifiers. Base tuples in some cases are derived from entities outside the LDB, such as an external data set or a sensor feed. For the latter case we introduce *external* lineage, which is formalized in this section along with *internal* lineage, but not discussed in any detail until Sect. 4. External lineage refers to a set of external symbols we denote by E. Thus, the set of symbols known by an LDB is $S = I(\bar{R}) \cup E$.

Definition 1 (Database with lineage): An LDB D is a triple $(\bar{R}, S, \lambda)$, where $\bar{R}$ is a set of relations, S is a set of symbols containing $I(\bar{R})$, and λ is a *lineage function* from S to 2^S.

Example 1 We introduce as a running example a highly simplified "crime-solver" database. Consider LDB relations `Drives(person,car)` and `Saw(witness,car)` representing driver information and crime-vehicle sightings respectively. Consider also a relation `Accuses(witness,person)` produced by the query $\pi_{\texttt{witness,person}}(\texttt{Saw} \bowtie \texttt{Drives})$. Here is some sample data:

Saw

ID	witness	car
21	Amy	Mazda
22	Amy	Toyota
23	Betty	Honda

Drives

ID	person	car
31	Jimmy	Mazda
32	Jimmy	Toyota
33	Billy	Mazda
34	Billy	Honda

Accuses

ID	witness	person	
41	Amy	Jimmy	$\lambda(41) = \{21, 31\}$
42	Amy	Jimmy	$\lambda(42) = \{22, 32\}$
43	Amy	Billy	$\lambda(43) = \{21, 33\}$
44	Betty	Billy	$\lambda(44) = \{23, 34\}$

The ID column denotes the tuple identifiers, and empty lineage is omitted.

Our basic formalism places no restrictions on the lineage function λ. However, when operations are performed there is often an obvious lineage function for the tuples in the result. The above example demonstrates a natural lineage function for joins: lineage of a tuple t in the result of a join is the set of tuples, one from each of the joined relations, that were combined to form t, e.g., `(Amy, Billy)` is obtained from `(Amy, Mazda)` and `(Billy, Mazda)`. Some operations, such as negation, duplicate-elimination, and aggregation, have less obvious lineage functions. For discussion of lineage functions see, e.g., [8,11,18,19,35]. The operations we consider in this paper all have simple lineage functions, and furthermore they preserve a notion of *well-behaved* lineage that we formalize later in the paper.

In an LDB, query results include lineage that refers to other tuples in the database. Hence, in our formalism the result of applying a query Q to database D includes the original relations $\bar{R}$ and a new relation for Q's answer with the appropriate lineage function. Thus, an important aspect of LDBs is that we cannot consider each relation in the database in isolation. We explore this point further in Sect. 4.4.

Note that even though a relation may contain duplicates, each tuple has its own lineage. For instance, tuples 41 and 42 in Example 1 have the same data. However, each one of them was derived differently, therefore they have a different lineage. Extending our model to set semantics requires more complex lineage functions than those we consider in this paper, and is a subject of follow-on work.

As discussed in the introduction, lineage is particularly important in data integration settings. If base relations are derived from different data sources, external lineage may be used to convey that information, e.g., by encoding the URI of the source as part of external symbol identifiers. Since LDBs keep track of lineage, data integration applications can easily query the lineage to examine how tuples potentially derived through multiple layers of complex queries relate to the original sources.

2.2 Uncertain databases

An uncertain database represents a set of *possible instances*, each of which is one possible state of the database. A number of different formalisms have been proposed for representing sets of possible instances, e.g., [1,4,23,29,31,34]. One difference among these formalisms is in their expressive power: which sets of possible instances can be represented in the formalism. In what follows we introduce *x-relations*, a specific formalism for uncertain databases. Conceivably, we could have considered the combination of any uncertainty formalism with lineage, but we found x-relations to be a good starting point. They provide a good balance of simplicity and expressiveness, and are orthogonal to the capabilities brought in by lineage.

Definition 2 An *x-tuple* is a multiset of tuples, called *alternatives*. An x-tuple may be annotated with a '?', in which case it is called a *maybe x-tuple*. An *x-relation* is a multiset of x-tuples.

Alternatives of an x-tuple represent mutually exclusive values for the tuple, leading to the following definition of possible instances.

Definition 3 An x-relation R represents the set of possible instances P that can be constructed as follows: choose exactly one alternative from each x-tuple in R that is not a maybe x-tuple, and choose zero or one alternative from each x-tuple in R that is a maybe x-tuple.

Example 2 The following x-relation represents an uncertain version of relation `Saw` from Example 1:

ID	Saw(witness, car)	
21	`(Amy,Mazda) || (Amy,Toyota)`	?
23	`(Betty,Honda)`	

Here, Amy may have seen a Mazda, a Toyota, or no car at all, and the relation has three possible instances.

The uncertainty about the car seen by Amy may come from the "source" herself, who may be uncertain about her memory of the facts. Uncertainty may also come from the fact that this information was extracted from a written police report, and the process of extracting information from plain text generated uncertainty. Alternatively, data may come from contradictory sources, e.g., two inspectors who interrogated Amy at different times, and got different answers from her. These uncertainty causes are very common in data integration applications.

A formalism for representing uncertainty is said to be *complete* if it can represent any finite set of possible instances. *c-tables* [31] is the prototypical complete formalism for uncertainty. x-relations are not a complete formalism. For example, the join `Accuses` of the x-relation `Saw` above with `Drives` from Example 1 cannot be represented as an x-relation: x-tuples are independent, so they cannot express the fact that if Amy accuses Jimmy (due to the Mazda), then she must accuse Billy as well.

Studies of completeness in various models for uncertainty can be found in, e.g., [2,23,29,31,34]. We will soon see (Sect. 3.1) that although x-relations alone are incomplete as shown above, adding lineage makes them complete.

3 Combining lineage and uncertainty

We now present ULDBs, a representation that captures both lineage and uncertainty. ULDBs extend the LDBs of Sect. 2.1 with the x-relations of Sect. 2.2.

Definition 4 A ULDB D is a triple $(\bar{R}, S, \lambda)$, where $\bar{R}$ is a set of x-relations, S is a set of symbols containing $I(\bar{R})$, and λ is a lineage function from S to 2^S.

Identifiers in $I(\bar{R})$ now correspond to tuple alternatives. $I(\bar{R})$ thus contains pairs (i, j), where i identifies the x-tuple and j is an index for one of its alternatives. When we refer to an arbitrary symbol in the set S, we use $s_{(i,j)}$, denoting either $(i, j) \in I(\bar{R})$ or an external symbol.

External symbols behave just like identifiers of x-tuple alternatives: $s_{(i,j)}$ and $s_{(i,j')}$ are mutually exclusive, as they are part of the same "logical" x-tuple t_i, which may also be annotated with a '?', just like an x-tuple of $\bar{R}$.

Example 3 We combine the uncertain `Saw` x-relation from Example 2 with the earlier `Drives` relation to create a new version of `Accuses` that has both uncertainty and lineage:

ID	Saw(witness, car)	
21	`(Amy,Mazda) || (Amy,Toyota)`	?
23	`(Betty,Honda)`	

ID	Drives(person, car)
31	`(Jimmy,Mazda)`
32	`(Jimmy,Toyota)`
33	`(Billy,Mazda)`
34	`(Billy,Honda)`

ID	Accuses(witness, person)		
41	`(Amy,Jimmy)`	?	$\lambda(41,1)=\{(21,1),(31,1)\}$
42	`(Amy,Jimmy)`	?	$\lambda(42,1)=\{(21,2),(32,1)\}$
43	`(Amy,Billy)`	?	$\lambda(43,1)=\{(21,1),(33,1)\}$
44	`(Betty,Billy)`	?	$\lambda(44,1)=\{(23,1),(34,1)\}$

We now define the semantics of a ULDB as a set of possible instances, where each instance is an LDB. The main technical challenge in the definition is to ensure that each possible LDB is based on consistent lineage. Recall that alternatives of an x-tuple are mutually exclusive in a given instance (0 or 1 of them are chosen), so we need to ensure that a possible LDB does not have two tuples whose lineages are from distinct alternatives of the same x-tuple. Recall $s_{(i,j)}$ denotes both internal identifiers $(i, j) \in I(\bar{R})$ and external symbols.

Definition 5 Let $D = (\bar{R}, S, \lambda)$ be a ULDB. A possible LDB D_k of D is obtained as follows. Pick a set of symbols $S_k \subseteq S$ such that:

1. If $s_{(i,j)} \in S_k$, then for every $j' \neq j$, $s_{(i,j')} \notin S_k$.
2. $\forall s_{(i,j)} \in S_k, \lambda(s_{(i,j)}) \subseteq S_k$.
3. For any t_i such that there does not exist a $s_{(i,j)} \in S_k$, the following hold: (i) t_i is a maybe x-tuple, and (ii) $\forall s_{(i,j)} \in t_i$, either $\lambda(s_{(i,j)}) = \emptyset$ or $\lambda(s_{(i,j)}) \nsubseteq S_k$.

The possible LDB D_k is the triple $(\bar{R}_k, S_k, \lambda_k)$ where $\bar{R}_k$ includes exactly the alternatives of x-tuples in $\bar{R}$ such that $s_{(i,j)} \in S_k$, and λ_k is the restriction of λ to S_k.

Intuitively, the first condition in Definition 5 says that alternatives of the same x-tuple are mutually exclusive, i.e., at most one of them may appear in each possible instance. The second condition enforces the semantics of lineage: if an alternative is present in a possible instance, so must be the alternatives it was derived from. Observe that this implication is in one direction only. The third condition says that an x-tuple must yield a tuple in a possible instance unless: (i) it is a maybe x-tuple, and (ii) none of its alternatives has a nonempty lineage that would have been consistent with condition 2.

Example 4 We explain the possible instances of the ULDB in Example 3. Consider the choices for x-tuple 21 of `Saw`, which has two alternatives and is a maybe x-tuple. The possible instance that picks (21,1) must also have (41,1) and (43,1) to satisfy condition 3 in Definition 5, and it cannot have (42,1) or condition 2 would be violated. Similarly, the possible instance that picks (21,2) must have (42,1) but not (41,1) or (43,1). The possible instance that doesn't pick any alternative for x-tuple 21 has neither of (41,1) or (42,1), nor (43,1) by condition 2. Note that since (23,1) and (34,1) are always present, all possible instances have tuple (44,1) to satisfy condition 3. This gives us the three possible instances we expect. Note in particular that not all combinations of the maybe x-tuples in `Accuses` are included in the possible instances.

3.1 Completeness

As discussed earlier, *completeness* is one of the important measures for the expressive power of a formalism for uncertainty. In general, a formalism is complete if it is possible to represent any set of possible instances within the formalism. Extending the traditional notion of completeness for uncertain databases, we consider a stronger definition that includes both uncertainty and lineage. The following theorem shows that ULDBs are indeed complete.

Theorem 1 *Given any set of possible* LDB*s* $P = \{P_1, P_2, \ldots, P_m\}$ *over relations* $\overline{R} = \{R_1, R_2, \ldots, R_n\}$*, there exists a* ULDB $D = (\overline{R}, S, \lambda)$ *whose possible* LDB*s are* P.

Proof First, we construct $\overline{R}$ with x-relations $S_1, \ldots, S_n$, corresponding to $R_1, \ldots, R_n$, and an extra relation PW that encodes the possible instances. PW contains exactly one x-tuple: `(1)||(2)||...||(m)`. Intuitively, the possible instance of D in which this tuple takes the value `(j)` encodes P_j.

Each S_i is constructed as follows. For every P_j, each tuple t in R_i forms a maybe x-tuple with just one alternative with value t. Duplicates within and across possible instances are preserved in S_i. We add `(j)` in PW to the lineage of alternatives in tuples copied from P_j. This now exactly encodes the data in each of the possible instances. The correct lineage is obtained as follows. We look at the lineage λ_j in P_j and mimic it in the x-tuples it contributes in S_1 through S_n. For example, if $\lambda_j(t_1) = \{t_2\}$ in P_j, where $t_1 \in R_1$ and $t_2 \in R_2$, the x-tuple that t_2 gave in S_2 is added to the lineage of the x-tuple from t_1 in S_1.

As a final step, we remove the extra relation PW but retain its symbols as external lineage. Therefore, each possible LDB of D now has the same schema as each P_j, and represents exactly the same data and internal lineage. □

3.2 Well-behaved lineage

Although the formal definition of a ULDB allows an arbitrary lineage function λ, in practice tuples are derived as results of queries, data imports, and other activities. Therefore, we expect λ to have a restricted structure and not be an arbitrary function. As a simple example, we don't expect to have a tuple t_1 derived from t_2 and also t_2 derived from t_1.

We define an interesting restricted class of lineage that we call *well-behaved* lineage. We will see that this class is closed under many relational operations, and its properties yield efficient algorithms for them. Let λ^* denote the transitive closure of lineage function λ.

Definition 6 (*well-behaved lineage*) The lineage of an x-tuple t_i is *well-behaved* if it satisfies the following three conditions:

1. **Acyclic:** $\forall s_{(i,j)}, s_{(i,j)} \notin \lambda^*(s_{(i,j)})$
2. **Deterministic:** $\forall s_{(i,j)}, \forall s_{(i,j')}$, if $j \neq j'$ then either $\lambda(s_{(i,j)}) \neq \lambda(s_{(i,j')})$ or $\lambda(s_{(i,j)}) = \emptyset$

3. **Uniform:** $\forall s_{(i,j)}, \forall s_{(i,j')}$, $B(s_{(i,j)}) = B(s_{(i,j')})$, where $B(s_{(i,j)}) = \{t_k \mid \exists s_{(k,l)} \mid s_{(k,l)} \in \lambda(s_{(i,j)})\}$

We say that a ULDB $D = (\overline{R}, S, \lambda)$ is *well-behaved* if all its x-tuples have well-behaved lineage.

Informally, Definition 6 says lineage is well-behaved when: (1) there are no cycles; (2) all alternatives of an x-tuple have distinct lineage; and (3) their lineage points to alternatives of the exact same set of x-tuples. Observe that, because of uniformity, acyclicity holds not only for alternatives, but also for x-tuples, i.e., no x-tuple appears in its own lineage graph.

We refer to an x-tuple with empty lineage as a *base x-tuple*. An interesting and useful property of well-behaved lineage is that the possible instances of a well-behaved ULDB are determined entirely by the base x-tuples. That is, selecting a set of alternatives for base x-tuples determines which alternatives are selected for all x-tuples derived from them.

Theorem 2 (well-behaved ULDB) *Let D_1 and D_2 be two possible instances of a well-behaved* ULDB *$D = (\bar{R}, S, \lambda)$. Then, $D_1 = D_2$ if and only if D_1 and D_2 pick the exact same alternative or '?' for every base x-tuple.*

Proof Define the distance of each alternative from base data as the maximum number of lineage links that can be traversed before reaching base data. The acyclicity property of well-formed lineage ensures that this distance of each alternative is finite.

The forward direction is immediate: $D_1 = D_2$ means by definition that D_1 and D_2 pick the same set of symbols. In particular, they pick the same alternatives or '?' for all base x-tuples.

We prove the reverse direction by contradiction. Suppose D_1 and D_2 pick the same alternatives or '?' for all base x-tuples, yet $D_1 \neq D_2$. There exists an alternative (from a derived tuple) in D_1 that is not present in D_2. Consider one such alternative $s_{(i,j)}$ in D_1 whose distance to base data is minimum. D_2 must have some other alternative $s_{(i,j')}$ chosen. Note that because of the uniformity condition of well-formed lineage, the distance of two alternatives of the same x-tuple, here $s_{(i,j)}$ and $s_{(i,j')}$, to base data is the same.

We show that the conditions on well-behaved ULDBs entail that there exists another pair of alternatives that is closer to base data than $s_{(i,j)}$, $s_{(i,j')}$, such that one alternative is present in D_1 (and not D_2) and the other in D_2 (and not D_1).

Since $s_{(i,j)}$ is not a base alternative, $\lambda(s_{(i,j)}) \neq \emptyset$. By the determinism property of well-formed lineage, $\lambda(s_{(i,j)}) \neq \lambda(s_{(i,j')})$. Consider some $s_{(k,l)} \in \lambda(s_{(i,j)})$ but not in $\lambda(s_{(i,j')})$, therefore $s_{(k,l)}$ is not from a base tuple. By uniformity, there exists some $s_{(k,l')} \in \lambda(s_{(i,j')})$. We now have $s_{(k,l)}$ and $s_{(k,l')}$ closer to base data, thus violating our assumption that $s_{(i,j)}$ and $s_{(i,j')}$ were the closest pair of alternatives to base data chosen in D_1 and D_2 respectively, a contradiction. □

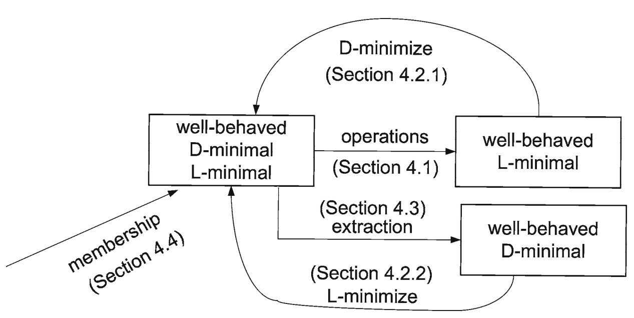


Fig. 1 ULDB states and queries

The theorem above shows that well-behaved lineage essentially considers a class of uncertainty that can be captured by (1) a finite set of base facts that are either mutually exclusive or independent, and (2) possibly correlated data derived from these base facts in a way that propagates but does not affect uncertainty. Well-behaved lineage is therefore well-suited for database queries. We will soon see that if we start from a well-behaved ULDB and perform a standard set of relational operations creating the natural lineage for the results, the ULDB remains well-behaved. Unless otherwise specified, we assume well-behaved ULDBs for the rest of the paper.

4 Querying ULDBs

In this section we consider querying and minimizing ULDBs. We begin in Sect. 4.1 by introducing the class of queries we consider. In particular, we identify queries that are monotonic w.r.t. to data and lineage. Section 4.2 describes how we answer such queries over ULDBs.

ULDBs do not necessarily have a unique representation. Section 4.3 considers two notions of minimality for ULDBs: (1) *D-minimality*, guaranteeing that a ULDB does not contain extraneous data, and (2) *L-minimality*, guaranteeing that a ULDB does not contain extraneous lineage. We describe how to minimize ULDBs w.r.t. both notions. We show that when ULDBs are minimized, we can efficiently answer *membership queries*, where the goal is to determine whether a particular tuple (or set of tuples) is guaranteed to be in some (or all) possible instances of a ULDB.

Finally, Sect. 4.4 discusses the *relation extraction* problem. Since ULDBs track the lineage of its tuples, we cannot look at an x-relation in a ULDB in isolation of others. We show how we can appropriately extract lineage along with the result x-relation, so that the correct set of possible instances is preserved.

Figure 1 summarizes the different operations (querying, extraction, and minimization) we consider for ULDBs, and the possible transitions between states of the ULDB.

4.1 DL-monotonic queries

We will restrict our discussion to queries that are *monotonic* with respect to data and lineage. Monotonicity is defined in terms of LDBs, i.e., "certain" databases with lineage. Extending these definitions to ULDBs is not necessary, since the semantics of monotonic operations are also defined on LDBs.

To define monotonicity, we must first define containment of LDBs. Intuitively, for an LDB D to be contained in D', every data element and its transitive "lineage graph" in D should also be in D'.

Definition 7 Let $D = (\bar{R}, S, \lambda)$ and $D' = (\bar{R}', S', \lambda')$ be two LDBs, where $\bar{R}$ and $\bar{R}'$ have the same schemas. We say that D is *contained* in D', denoted $D \subseteq D'$, if:

1. $S \subseteq S'$
2. $\bar{R}$ is contained in $\bar{R}'$, i.e., if $t \in R_i$ then $t \in R'_i$, with the same tuple identifier
3. For every symbol $s_1 \in S$, if $s_2 \in \lambda(s_1)$, then $s_2 \in \lambda'^*(s_1)$.

Note that $\subseteq$ is not exactly a partial order on LDBs because it is not antisymmetric. Specifically, $D \subseteq D'$ and $D' \subseteq D$ only implies that $\lambda^* = \lambda'^*$, not necessarily that $\lambda = \lambda'$.

Based on Definition 7, we define the class of *DL-monotonic queries*. In the definition, given a query Q and an LDB D, $Q(D)$ is an LDB that extends D with one x-relation R_q and with lineage λ_{R_q} from R_q to $I(\bar{R})$. We write $Q(D) = D + (R_q, I(R_q), \lambda_{R_q})$.

Definition 8 Let D be an LDB. Let $D|_I$ denote the restriction of D to the tuples identified in set I, and the lineage among them. A DL-monotonic query is a function Q from LDBs to LDBs that satisfies the following conditions:

1. $\forall D, \forall D'$ such that $D \subseteq D'$, then $Q(D) \subseteq Q(D')$.
2. $\forall t \in R_q$, $Q(D|_{\lambda(t)}) = D|_{\lambda(t)} + (t, I(t), \lambda(t))$, and no strict subset of $D|_{\lambda(t)}$ produces t.

The first condition enforces monotonicity on both data and lineage. The second condition constrains the lineage of a result tuple to be a minimal subset of the database that produces exactly that tuple.

Example 5 In Example 1, the query `Accuses` $=$ $\pi_{\texttt{witness,person}}$(`Saw` $\bowtie$ `Drives`) is DL-monotonic. In particular the reader may verify that the lineage associated with the four x-tuples of `Accuses` satisfies Definition 8 above. Note that the lineage of each of the two `(Amy,Jimmy)` tuples must have a distinct combination of base tuples so that condition 2 of Definition 8 is satisfied.

Intuitively, any operation that can produce its results in a "tuple-by-tuple" fashion is DL-monotonic. Considering the standard relational operations with bag semantics: selection, projection, join, and union are all DL-monotonic, and so are any queries composed from them. Clearly, these operations are data monotonic, and the lineage of each resulting tuple points exactly to the combination of tuples that produces it, and it alone, hence they are DL-monotonic. Aggregation, duplicate-elimination, and some set operators are not DL-monotonic. In the remainder of this section, we assume all queries Q to be DL-monotonic. In follow-on work we are extending our approach to other operations.

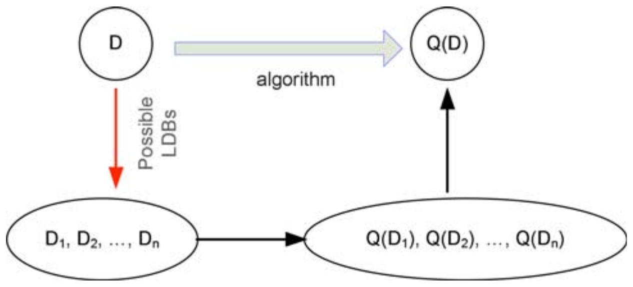


Fig. 2 Semantics of queries on ULDBs

4.2 Applying a query to a ULDB

We consider the problem of applying a query Q to a ULDB D. In our formulation of the problem, the result $Q(D)$ is defined to include the original database *and* the answer relation. We consider extracting the answer relation from the resulting ULDB in Sect. 4.4.

The semantics of query answers are determined via the instances of the ULDB, as depicted in Fig. 2: $Q(D)$'s possible LDBs are logically obtained by applying Q to each of the $D_1, \ldots, D_n$ possible instances of D. Hence, the goal of our algorithm it to implement the top (broad) arrow in the diagram, going directly from the representation of D to the representation of $Q(D)$.

Algorithm 1 (see figure) proceeds in two phases. First (lines 4–5), it performs a "standard" evaluation of the query Q on an LDB $\overline{D}$ that contains all the alternatives of the base x-relations. The resulting relation $\overline{R_q}$ and its lineage λ_{R_q} are then used to: (a) construct one x-tuple t_l in R_q for each combination $t_1, \ldots, t_n$ of x-tuples in D that produced tuples through Q (lines 6–8); and (b) generate lineage for t_i's alternatives (line 9). Note that although t_l is defined as a maybe x-tuple, it may still contribute a tuple in every possible LDB of $Q(D)$. We discuss elimination of extraneous '?'s in Sect. 4.3.1.

Theorem 3 *Given a* ULDB *D and a query Q:*

1. *Algorithm 1 returns $Q(D)$.*
2. *If D is a well-behaved* ULDB, *then so is $Q(D)$.*

Algorithm 1 Query evaluation

```
input: a ULDB $D$ with $x$-relations $\{R_1, \ldots, R_n\}$,
       and a DL-monotonic query $Q$ on $D$
output: a ULDB $D' = Q(D)$
1: $R_q \leftarrow \emptyset$ ; $\lambda_{R_q} \leftarrow$ undefined function
2: Let $\overline{D} = \overline{R_1}, \ldots, \overline{R_n}$ be the LDB such that $\forall k, k \in [1, n]$
   $\bar{R}_k = \{$ tuples $s_{(i,j)} \mid s_{(i,j)}$ is an alternative in $R_k\}$
   Remember with each tuple $s_{(i,j)}$ that it represents
   alternative $j$ of $x$-tuple $t_i$.
3: Compute $Q(\bar{D}) = \bar{D} + (\bar{R}_q, I(\bar{R}_q), \lambda_{\bar{R}_q})$
4: Group the tuples in $\bar{R}_q$ by the $x$-tuple identifiers
   corresponding to the tuples in their lineage
5: for each group of tuples with the same set of $x$-tuple
   identifiers $t_1, \ldots, t_n$ do
6:     create a maybe $x$-tuple $t_l$ in $R_q$ with all the tuples
       of the group as alternatives
7:     $\forall s_{(l,k)}$ alternative of $t_l$, set $\lambda_{R_q}(s_{(l,k)})$ as in $\lambda_{\bar{R}_q}$
8: end for
9: return $D' = D + (R_q, I(R_q), \lambda_{R_q})$
```

To prove this theorem, we need the following lemma, which establishes that a ULDB can be grown monotonically by adding new x-tuples and well-behaved lineages: □

Lemma 1 *Given a* ULDB $D = (\bar{R}, S, \lambda)$ *with possible* LDBs $D_1, \ldots, D_n$, *and a maybe x-tuple t, such that* (1) $t \notin D$ *and* (2) *t has well-behaved non-empty lineage* $\lambda(t) \subseteq I(\bar{R})$, *then the* ULDB $D' = D + (\{t\}, I(t), \lambda(t))$ *has possible instances* $D'_1, \ldots D'_n$ *such that* $\forall i,\ D_i \subseteq D'_i$.

In this case, we say that D' preserves the possible instances *of D.*

Proof Any possible LDB P'_i of D' makes the same alternative choices for all x-tuples of D as one and only one possible LDB P_j of D, or it wouldn't be consistent. Clearly, $P_j \subseteq P'_i$.

Conversely, for any possible LDB P_j of D, one and only one choice is possible for t:

- either the alternatives chosen by P_j for the x-tuples in the lineage of t (all the same, by uniformity) form the lineage of one of the alternatives of t, and this alternative (unique by determinism) must be chosen to get a consistent LDB, or
- the alternatives chosen by P_j for the x-tuples in the lineage of t do not correspond to the lineage of any alternative. In this case the only choice for t is not picking an alternative, which is possible because t is a maybe x-tuple and none of the alternative's lineage is satisfied.

Acyclicity ensures that the choice made for t does not affect the rest of the LDB. The obtained LDB is the unique possible LDB P'_i of D' s.t. $P_j \subseteq P'_i$. □

Proof (Theorem 3) The algorithm returns a ULDB $D' = (R_q, I(R_q), \lambda_{R_q})$ which adds to D the x-relation R_q containing exclusively maybe x-tuples with well-behaved lineage. Therefore, by the above lemma, D' preserves the possible instances of D. We now show that if D_i is a possible possible LDB of D, then the corresponding LDB D'_i of D' is precisely $Q(D_i)$.

As $D_i \subseteq \bar{D}$, by monotonicity $Q(D_i) \subseteq Q(\bar{D})$. Every tuple in $Q(D_i)$ becomes an alternative in R_q with lineage pointing to alternatives of D that are picked by D_i. Since $D_i \subseteq D'_i$, D'_i can (and must) pick those alternatives. Hence, $Q(D_i) \subseteq D'_i$.

To show that $D'_i \subseteq Q(D_i)$, suppose D'_i picks some $s_{(i,j)} \notin Q(D_i)$. Clearly, $\lambda(s_{(i,j)}) \subseteq D_i$, and by definition of lineage for positive queries $Q(D_i|_{\lambda}(s_{(i,j)}))$ produces $s_{(i,j)}$, which implies by monotonicity that $s_{(i,j)} \in Q(D_i)$, a contradiction.

Finally, if D is well-behaved, the addition of x-tuples with well-behaved lineage clearly preserves acyclicity, determinism and uniformity, hence D' is well-behaved. □

Observe that our algorithm is based on evaluating Q over a conventional database $\overline{D}$. Since the size of $\overline{D}$ is the same as the size of x-relations $R_1, \ldots, R_n$, complexity does not increase due to uncertainty. More importantly, we can implement Algorithm 1 readily using a standard relational DBMS, without having to build a special-purpose query engine for ULDBs. In fact, as we describe in Sect. 6, our implementation of the *Trio* prototype has taken exactly this approach. Of course special-purpose techniques also may be interesting in order to maximize performance of query processing on ULDBs.

4.3 ULDB minimality

An interesting aspect of ULDBs is that they do not have a unique representation. That is, we can have two different x-relations that have exactly the same set of possible instances. Therefore, it is interesting to ask whether a ULDB has a *minimal* representation. We now define two notions of minimality for ULDBs: *data minimality* and *lineage minimality*, and then show how we can answer membership queries on minimal ULDBs efficiently.

4.3.1 Data minimality

As the following example illustrates, a ULDB may contain extraneous data, including "impossible" alternatives in an x-tuple, or x-tuples unnecessarily marked with '?'. As a special case, an entire x-tuple is extraneous if all its alternatives are extraneous.

Example 6 In Example 3, the '?' on x-tuple 44 is extraneous because the alternative (24,1) is present in every possible LDB. As an example of an extraneous alternative (entire x-tuple in this case), consider the following x-relations, where `Car1` and `Car2` represent separate lists of possible crime vehicles.

ID	Saw(witness, car)
1	(Carol,Acura) || (Carol,Lexus)

ID	Car1(car)
2	Acura

ID	Car2(car)
3	Lexus

Suppose we perform `Saw1 = (Car1 ⋈ Saw)` and `Saw2 = (Car2 ⋈ Saw)` to get sightings related to the two car lists:

ID	Saw1(witness,car)
4	(Carol,Acura)

$\lambda(4,1)=\{(1,1),(2,1)\}$

ID	Saw2(witness,car)
5	(Carol,Lexus)

$\lambda(5,1)=\{(1,2),(3,1)\}$

Finally, suppose we compute `(Saw1 ⋈`$_{\texttt{witness}}$` Saw2)` to find pairs of car sightings in `Car1` and `Car2` by the same witness:

ID	(witness,car1,car2)	
6	(Carol,Acura,Lexus)	? $\lambda(6,1)=\{(4,1),(5,1)\}$

There is no possible instance of the database with alternative (6,1). Intuitively, Carol saw either an Acura or a Lexus, while both sightings would be necessary to derive x-tuple `(Carol,Acura,Lexus)`. Thus, `(Carol,Acura,Lexus)` is extraneous.

We now define data minimality formally.

Definition 9 (*D-minimality*) An alternative (i, j) of an x-tuple t_i in a ULDB D is said to be *extraneous* if removing it from the x-relation does not change the possible instances of D. Similarly, a '?' on an x-tuple in D is said to be extraneous if removing it does not change the possible instances of D. A ULDB D is *D-minimal* if it does not include any extraneous alternatives or '?'s.

The following theorems provide conditions on ULDBs that enable us to detect extraneous data.

Theorem 4 (extraneous alternative) *Let D be a well-behaved* ULDB. *An alternative with identifier (k, l) (in x-tuple t_k) in D is extraneous if and only if there exist $s_{(i,j_1)}$ and $s_{(i,j_2)}$, both in $\lambda^*(s_{(k,l)})$, with $j_1 \neq j_2$.*

Proof Clearly, if $s_{(i,j_1)}, s_{(i,j_2)} \in \overline{\lambda}(s_{(k,l)})$, the alternative is extraneous. It now suffices to show that if (k, l) is extraneous, there exist $s_{(i,j_1)}$ and $s_{(i,j_2)}$, both in $\overline{\lambda}(s_{(k,l)})$. We prove the contrapositive in two steps:

1. We show that the tuple t, corresponding to this alternative, appears in some possible instance if the base tuples in $\overline{\lambda}(s_{(k,l)})$ are picked. That is, $t \in Q(D)$ where D is the LDB constituting the base tuples $\overline{\lambda}(s_{(k,l)})$, and Q is the query performed to obtain the x-relation of $s_{(k,l)}$ from the base relations.
2. There exists a possible instance LDB of the base relations D' that satisfies $D \subseteq D'$. Now $t \in Q(D)$ and $D \subseteq D'$ and so by Definition 8, $t \in Q(D')$. Therefore, t appears in a possible instance with D' and is not extraneous. □

In other words, an alternative is extraneous if and only if it has contradictory lineage. Let us now consider extraneous '?'. Let $\eta(t_i)$ denote the number of alternatives in x-tuple t_i that are not extraneous. Let $h(t_i)$ denote the set of base x-tuples from which t_i is derived, i.e., $t_j \in h(t_i)$ if $\exists s_{(i,k)}, \exists s_{(j,l)}$ such that $s_{(j,l)} \in \lambda^*(s_{(i,k)})$ and $\forall m, \lambda(s_{(j,m)}) = \emptyset$.

Theorem 5 (extraneous '?') *Let D be a well-behaved* ULDB. *A '?' on an x-tuple $t \in D$ is extraneous if and only if:*

1. *No x-tuple in $h(t)$ has a '?'*
2. $\eta(t) = \prod_{t' \in h(t)} \eta(t')$

Proof First note that even if one of the tuples in $h(t)$ has a ?, the ? in t is not extraneous because choosing the ? in the base tuple results in t not having any tuple. Now, we claim that even if one of the combinations of alternatives in $h(t)$ does not give an alternative in t, the ? in it is not extraneous. This follows from an argument similar to the proof of Theorem 4, i.e., choosing a possible instance containing the combination in $h(t)$ not having an alternative result in a possible instance of the database with no alternative being picked from t. Finally, the only way ? can be picked for t is if no alternative of t is satisfied; and, this can only happen if some combination of alternatives in $h(t)$ does not result in an alternative in t. □

We can now use Theorems 4 and 5 to D-minimize ULDB representations. Since minimization needs to work on the transitive closure λ^* of the lineage, we can follow two approaches to D-minimization: (1) a *lazy* approach in which λ^* is computed during minimization, and (2) an *eager* approach in which the algorithm for operations maintains λ^* and also the D-minimal form. Algorithm 2 presents the lazy approach for D-minimizing a ULDB D; the eager approach uses the same idea but performs the computation incrementally with operations. It is easy to see that the algorithm returns the D-minimal representation.

4.3.2 Lineage minimality

A second notion of minimality has to do with lineage. For ULDB $D = (\bar{R}, S, \lambda)$, let its *internal lineage* be the restriction of λ to only symbols in $I(\bar{R})$. (Recall the domain of symbols $S = I(\bar{R}) \cup E$ also includes external symbols E.)

Algorithm 2 Lazy Algorithm for D-minimization

```
1: input: A well-behaved ULDB $D$
2: output: An equivalent D-minimized version of $D$
3: for each $x$-relation $R$ in $D$ do
4:    Skip if $R$ has been D-minimized
5:    Recursively perform Steps 3-8 to D-minimize all $x$-relations
      $\{R_1, R_2, \ldots, R_n\}$ that contain lineage of some $x$-tuple in $R$.
6:    Compute $\lambda^*$ for each alternative of $R$ using the already computed
      $\lambda^*$ for each $R_i$
7:    Delete all extraneous alternatives using the condition of Theorem 4
8:    Compute $\eta(t)$ for all $x$-tuples $t$ in $R$ and for all $x$-tuples in $h(t)$
9:    Use the condition in Theorem 5 to delete any extraneous '?'s
10:   Mark $R$ as D-minimized
11: end for
12: return $D$
```

Definition 10 (*L-minimal* ULDB) A ULDB $D = (\bar{R}, S, \lambda)$ is *L-minimal* if for any $D' = (\bar{R}, S', \lambda')$ over the same x-relations $\bar{R}$ such that:

1. $S' \subseteq S, \lambda'^* \subseteq \lambda^*$
2. D and D' have the same internal lineage

D' has the same possible instances as D only if $S' = S$ and $\lambda'^* = \lambda^*$.

We have the following main theorem about L-minimality.

Theorem 6 (L-minimality of Algorithm 1) *Given a well-behaved L-minimal* ULDB *D and a query Q, the result $Q(D)$ of Algorithm 1 is an L-minimal* ULDB.

Proof (sketch) Let the x-relation added by Q be R_q. We first show that the newly generated lineage is minimal, i.e., removing any symbol from $\lambda(x)$ where x is a symbol of R generates a possible instance not in $Q(D)$. Next, if $Q(D)$ is not L-minimal and some $\lambda(y)$ can be made smaller without changing the possible instances where y is in some relation of D, then $\lambda(y)$ can be reduced in the same way in D too.

The above theorem guarantees that query processing preserves L-minimality. Algorithms for "L-minimizing" a ULDB D, i.e., finding an L-minimal D' that coincides with D on data and internal lineage, are the topic of ongoing work. Finding one L-minimal ULDB equivalent to D can be done efficiently, by iteratively pruning the external lineage of D. However, the result of L-minimization is not unique. It is still open whether we can efficiently find a "global minimum" among all possible L-minimizations, with respect to the size of their representation. □

4.3.3 Membership queries

One useful side-effect of minimization is that it helps us answer *membership queries* [2,23,29–31]: determining whether a particular tuple or relation is present in some (or every) possible instance of an uncertain database. In the context of ULDBs, these problems are defined as follows.

Definition 11 (*membership queries*)

- *Tuple membership* (*resp. certainty*): Given a ULDB D containing an x-relation R, and given a tuple alternative $t \in R$, determine whether $t \in R$ in some (resp. all) possible instance(s) of D.
- *Instance membership* (*resp. certainty*): Given a ULDB D containing an x-relation R, and a multiset T of tuple alternatives from R, determine whether R contains exactly the tuples of T in some (resp. all) possible instance(s) of D.

Note that, in the context of ULDBs, membership and certainty problems are defined in term of specific tuple alternatives of the database, identified by their $s_{(i,j)}$ symbols. This is a slight variation from the traditional definition of these problems, which is based on tuple *values*. This difference reflects the fact that, in our model, tuples are defined not only by their values, but also by their lineage.

The following theorem shows that it is tractable to answer the tuple-membership and tuple-certainty problems. The algorithms to do so (outlined in the proof) build directly on D-minimization. However, as is true of all complete uncertainty models [23] including ULDBs, the instance-membership and instance certainty problems are intractable.

Theorem 7 *Let D be a well-behaved* ULDB.

1. *The tuple-membership and tuple-certainty problems are solvable in polynomial time in the size of D.*
2. *The instance-membership and instance-certainty problems are NP-hard.*

Proof By Definition 9, a D-minimal ULDB does not have extraneous tuples or '?'s, i.e., all alternatives do appear in some possible instance. The tuple-membership problem for t and R thus returns "yes" if t is an alternative of the x-relation for R, and otherwise returns "no". Similarly, the tuple-certainty problem returns "yes" if there is an x-tuple with a single alternative t and no '?'. Therefore, tuple-membership and tuple-certainty can be answered in linear time in the size of D, if D is D-minimal. Finally, note that the D-minimization algorithm takes linear time in the size of the lineage of D. Therefore, these problems are also answered in linear time if D is not D-minimal.

The proof for instance-membership is by a reduction from the NP-complete graph 3-colorability problem, and for instance-certainty by a reduction from the Co-NP-complete graph non 3-colorability problem. The hardness of both these problems for c-tables was first shown in [2]. The colorability problems were reduced to instance-membership (or certainty) of the result of positive existential queries on base c-tables. These base c-tables, which represented the graph structure and possible colorings, can be represented as well-behaved base x-relations. Therefore, the instance membership and certainty problems are hard for ULDBs too. □

The minimization algorithms we described in this section are currently being implemented and their performance is being measured. We are also experimenting with various strategies of eager versus lazy minimization.

4.4 Extraction

Typically, after issuing a query to a database, users are interested in seeing only the result relation, not the entire database. More generally, given a ULDB, we may want to extract a subset of its relations, but in a way that preserves the possible instances of the extracted subset. In principle, whenever a database includes constraints across relations, extracting a subset of the database is an interesting question; otherwise, the meaning of every relation is independent of the others, and therefore extraction is trivial.

Extraction is also important in the context of data integration. If the data in a ULDB comes from multiple external sources, a flexible way to bring into the ULDB just the data that is needed from these sources is to create x-relations defined as queries on them. Performing extraction on these x-relations preserves their information, while discarding irrelevant data and lineage. Note that this idea is similar to how queries are used to define source mappings in traditional data integration systems such as mediators or data warehouses.

Definition 12 (*extraction*) Let D be a well-behaved ULDB with x-relations $\overline{R}$ and possible instances P, and let $\bar{X}$ be a subset of $\overline{R}$. The problem of extracting $\bar{X}$ from $\overline{R}$ is to return a well-behaved ULDB D' with $\overline{R'} = \bar{X}$ and possible instances P', such that the restriction of P to $\bar{X}$ equals P' with respect to data and internal lineage.

Simply removing the relations in $\overline{R} - \bar{X}$ and their symbols does not give a correct extracted result. For instance, if the x-relation `Accuses` from Example 3 is extracted without any lineage, x-tuple 43 may now occur without x-tuple 41, which is not allowed by any of the possible instances of the original ULDB.

The following algorithm produces the correct extraction.

Algorithm 3 Algorithm **relationExtract**

```
1: input: ULDB $D = (\bar{R}, S, \lambda)$, and $\bar{X} \subseteq \bar{R}$
2: output: a ULDB $D' = (\bar{X}, S', \lambda')$
3: $S' = I(\bar{X}) \cup (\bigcup_{x \in I(\bar{X})} \lambda^*(x))$
4: $\lambda' = \lambda|_{S'}$, the restriction of $\lambda$ to $S'$
5: return $D'$
```

Effectively, the algorithm works by identifying all lineage that is necessary to ensure that the possible instances of the extracted relations are preserved. Lineage that is not within the extracted relations is converted from internal (identifiers (i, j) in $I(\bar{R})$) to external (the corresponding symbols $s_{(i,j)}$). Note that by our definitions, the mutual exclusion of x-tuple alternatives carries over to what are now external symbols. One subtlety is that we must associate a logical '?' with each set of external symbols that were created from an x-tuple having a '?'.

Consider again the `Accuses` example discussed above. If we extract `Accuses` from the database shown in Example 3, we retain the lineage on the x-tuples of `Accuses`, except it now refers to external symbols. By doing so, Definition 5 of possible instances correctly prohibits a possible instance containing one but not the other of x-tuples 41 and 43.

We have the following theorem about our extraction algorithm.

Theorem 8 *Let $D = (\bar{R}, S, \lambda)$ be a well-behaved D-minimal* ULDB*, and consider any $\bar{X} \subseteq \bar{R}$.*

1. *Algorithm* **relationExtract** *returns a correct extraction D'.*
2. *Algorithm* **relationExtract** *runs in polynomial time in the size of D.*
3. *The result D' is D-minimal.*

Proof (sketch) The correctness of the algorithm follows from the fact that all lineage that constrains the possible instances for x-tuples in $\bar{X}$ is retained in D'. Since the algorithm needs only one traversal of the lineage of all x-tuples in $\bar{X}$, the running time is polynomial in the size of D. Finally, if D is D-minimal, there are no extraneous alternatives or '?'s. Now if we restrict the possible instances to $\bar{X}$, all alternatives in $\bar{X}$ still appear in some possible instance. Similarly, all x-tuples with '?' continue to give the empty instance in some possible instance in D'. Therefore, D' is D-minimal. □

5 Confidences and probabilistic data

We now show how ULDBs can be extended to include *confidence values* and probabilistic query processing. With confidences, ULDBs subsume the typical notion of *probabilistic databases*, which assign a confidence value to tuples, without alternatives or lineage [4,12,21,34]. A noteworthy feature of probabilistic query processing using ULDBs is that we can decouple the computation of data in query results from the computation of the data's probability (confidence) values. This decoupling enables more freedom with query plan selection than is typically available for probabilistic query processing [22], and it allows confidence values to be computed selectively as needed.

In some applications, confidence values come directly from the data. This is mostly the case for sensor data, where a confidence can be determined from the strength of the signal received by an individual sensor. However, in most cases, confidences are generated by the layers of processing traversed by the data (which advocates for keeping track of the lineage). For instance, in data integration applications, operations such as extraction from text, reference reconciliation, or schema matching rely on a variety of techniques (e.g., distance metrics, probabilistic models) that naturally generate confidence values along with their results.

Note that the confidence computations presented here are *exact*. This makes the system consistent with respect to query composition: A query will produce the same results and probabilities, regardless of the intermediary x-relations that may be materialized. Exact computation also avoids loosing precision on probabilities as layers of x-relations are built on top of each other. For well-behaved ULDBs, the cost of this exact computation is manageable, since it is polynomial in the size of the x-tuples for which confidences are computed and their lineage, and intermediary results may be cached and reused.

5.1 Confidence values

In the remainder of this section we assume ULDBs to be well-behaved and D-minimized. If we consider the semantics of x-relations probabilistically, then without lineage different alternatives of the same x-tuple represent *disjoint* events, while different x-tuples represent *independent* events. Recall from Sect. 3.2 that in well-behaved ULDBs, the possible instances are determined entirely by the choices for the base x-tuples; the choices for derived x-tuples are determined by their lineage.

We preserve this intuition when extending ULDBs with confidences. Now, each base alternative a has an associated *confidence value* $c(a)$. For each base x-tuple t, the sum $\sigma(t)$ of the confidence values of its alternatives must be at most 1, and exactly 1 if t has no '?'. The confidence of '?' for any x-tuple is $(1 - \sigma(t))$.

For flexibility, we support confidence values that are not necessarily modeled as probabilities. In Sect. 6 we give an example of where an application may want to use a different interpretation. However, in the rest of this section, we assume that the confidence values are interpreted as probabilities. In particular, when we discuss possible instances, we say that each instance has a *probability* of being the "correct" instance, and this probability is based on confidences in the data comprising the instance. Specifically, the probability of a possible instance is the product of the confidences of the base alternatives and '?' chosen in it.

Example 7 Suppose Amy sighted an Acura with confidence 0.8, while Betty is sure she saw either an Acura or a Mazda with confidences 0.4 and 0.6 respectively. Furthermore, Hank drives an Acura with confidence 0.6. We have:

ID	**Saw(witness, car)**	
11	`(Amy,Acura):0.8`	**?**
12	`(Betty,Acura):0.4 || (Betty,Mazda):0.6`	

ID	**Drives(person, car)**	
51	`(Hank,Acura):0.6`	**?**

This database has eight possible instances, since each of the three x-tuples has two possible choices. For example, the possible instance where Amy saw an Acura, Betty saw a Mazda, and Hank does not drive an Acura has confidence $0.8 * 0.6 * (1 - 0.6) \approx 0.20$.

It can be shown that for any well-behaved D-minimal ULDB with confidences, the following desirable properties hold.

1. The sum of probabilities of its possible instances is 1.
2. The confidence of a base alternative a (resp. '?' on an x-tuple t) equals the sum of the confidences of the possible instances where a (resp. no alternative of t) is picked.

5.2 Query processing

The presence of lineage allows us to decouple ULDB query processing with confidences into two steps:

1. *Data computation*, in which we compute the data and lineage in query results, just as in ULDBs without confidences.
2. *Confidence computation*, in which we compute confidence values for query results based on their lineage (and confidence values on base data).

We first motivate why this decoupling works. Then we discuss data and confidence computation in Sect. 5.3.

Suppose we have a derived x-tuple t, and consider one of its alternatives a. With well-behaved lineage, a appears in a possible instance if and only if all of the base x-tuple alternatives in the transitive closure of a's lineage appear in the instance. Furthermore, these base x-tuple alternatives are *independent*, since they have no lineage of their own and cannot be alternatives of the same x-tuple. Thus, the confidence of a is computed as the product of the confidences of the base-tuple alternatives in the transitive closure of its lineage. For an x-tuple t with a '?', confidence for the '?' is $(1-\sigma(t))$, where $\sigma(t)$ is the sum of the confidences of t's alternatives.

Thus, the confidence value for every result alternative a is a function of the confidence values for the base alternatives reachable by a's transitive lineage. Hence we need not compute confidence values during query processing—we can compute them afterwards using the lineage on query results together with the original base data confidences.

Next, we show how decoupling data and confidence computation overcomes a previously identified shortcoming of query processing in probabilistic databases, and we briefly discuss efficient confidence computation in the decoupled scenario.

5.3 Confidence computation

Dalvi and Suciu [21] show that naive propagation of confidences during query processing—essentially assuming independence of tuples in intermediate results—may lead to incorrect confidences in the result. We illustrate the problem with an example, and also show how our decoupled technique operates (correctly) on the same example.

In [25], we study the problem of exploiting lineage to efficiently compute confidences. The reader is referred to [25] for details on model extensions, and query processing techniques.

Example 8 Let us simplify the data in Example 7 to:

ID	**Saw(witness,car)**	
11	`(Amy,Acura):0.8`	**?**
12	`(Betty,Acura):0.4`	**?**

ID	**Drives(person,car)**	
51	`(Hank,Acura):0.6`	**?**

Suppose we want the list of accused persons with confidences: `Accused` $= \Pi_{\texttt{person}}$ `(Saw` $\bowtie$ `Drives)`. Here we are using a duplicate-eliminating projection. We consider three ways of executing this query: two query plans that compute confidences as part of operator execution, and a third method showing our decoupled approach.

Query Plan 1 (correct): Evaluating the query using the following plan gives the correct confidences in the result:

$$\Pi_{\texttt{person}}(\Pi_{\texttt{car}}(\texttt{Saw}) \bowtie \texttt{Drives})$$

In $\Pi_{\texttt{car}}(\texttt{Saw})$, there is just one tuple `(Acura)` whose confidence is given by:

$$\Pr((11,1) \vee (12,1)) = \Pr((11,1)) + \Pr((12,1)) - \Pr((11,1) \wedge (12,1))$$

Since alternatives (11, 1) and (12, 1) are independent, $\Pr(\texttt{(Acura)})$ evaluates to $0.8+0.4-(0.8*0.4) = 0.88$. Now joining `(Acura)` with x-tuple 51, we get the confidence of the result `(Hank,Acura)` to be $0.88 * 0.6 = 0.528$. In the final step, projecting onto `person`, the confidence remains 0.528.

Query Plan 2 (incorrect): Suppose instead we use plan:

$$\Pi_{\texttt{person}}\ (\texttt{Saw} \bowtie \texttt{Drives})$$

Now we get an incorrect result, because the intermediate x-tuples `(Amy,Acura,Hank)` and `(Betty,Acura,Hank)` from `(Saw` $\bowtie$ `Drives)` are not independent. Let these tuples have IDs (61, 1) and (62, 1) respectively. The confidence of `(Amy,Acura,Hank)` is:

$$\Pr((61,1)) = \Pr((11,1) \wedge (51,1))$$

giving $0.8 * 0.6 = 0.48$. Similarly, the confidence of `(Betty,Acura,Hank)` is $0.6 * 0.4 = 0.24$. Now the x-tuple `Hank` after projecting onto `person` has confidence given by

$$\Pr((61,1) \vee (62,1)) = \Pr((61,1)) + \Pr((62,1)) - \Pr((61,1) \wedge (62,1))$$

Assuming independence of tuples (61, 1) and (62, 1), the confidence evaluates to $0.48 + 0.24 - 0.48 * 24 = 0.6048$, which is incorrect. See [21] for further discussion of these issues.

Query Plan 3 (decoupled approach): In our approach, we first compute the query result using any execution plan. We get the one x-tuple `(Hank)`; let its identifier be (71, 1). Because of the duplicate-elimination operator, which is not DL-monotonic, $\lambda((71,1))$ is no longer a set of tuple alternatives (indicating conjunction), but rather a Boolean formula over alternatives. (Disjunctive and negative lineage is required once we go beyond the DL-monotonic operations; details are the subject of ongoing work.) Specifically, $\lambda((71,1)) = ((51,1) \wedge ((11,1) \vee (12,1)))$.

Now, we compute the confidence of the `(Hank)` tuple based on its lineage formula and confidence values for the (independent) base alternatives:

$$\Pr((71,1)) = \Pr(((51,1) \wedge ((11,1) \vee (12,1)))$$

With $\Pr((51,1)) = 0.6$, $\Pr((11,1)) = 0.8$, and $\Pr((12,1)) = 0.4$, we obtain the correct result $\Pr((71,1)) = 0.528$.

Our decoupled approach has two important advantages: First, the data computation step has the flexibility to use the most efficient execution plan, without worrying about plans that produce incorrect confidences as illustrated above. Second, in the case where confidence values may not be required for all data in all query results, the values can be computed selectively and on-demand.

To avoid the erroneous confidence calculations as exhibited in Example 8, reference [21] characterizes logical query plans that are guaranteed to propagate confidences correctly, and restricts considered evaluation strategies to such plans. In our decoupled approach, we have the luxury of a wider space of plans, which can be shown to result in arbitrarily large performance improvements (confidence computation included) in extreme cases. Consider a query Q that produces an empty result. Our approach does not need to perform any confidence computation for Q since there are no result x-tuples. The alternative approach computes confidences during query execution until finally the result is discovered to be empty. Furthermore, an expensive plan may need to be used in order to correctly compute confidence values that are eventually thrown away.

More concretely, suppose we have $2n$ large relations, $R_1(X), \ldots, R_n(X)$ and $S_1(Z), \ldots, S_n(Z)$, and two small relations $A(X, Y)$ and $B(Y, Z)$. Consider a query $Q(Y)$ that computes the natural join of all the relations and projects onto Y, and suppose $A \bowtie B$ is empty. With simple statistics any standard optimizer will choose to perform $A \bowtie B$ first. However, in the plans permitted by [21] (or any other plans that require independence of tuples for confidence propagation), $A \bowtie B$ must be performed last. In these plans, we can make the cost of computing $R_1 \bowtie \cdots \bowtie R_n$ and $S_1 \bowtie \cdots \bowtie S_n$ arbitrarily large.

Of course this example was contrived, and reference [21] shows that for some queries, computing results with confidences has #P-hard data complexity, regardless. In such situations, our decoupled approach offers a practical solution: Answers without confidence values give an approximation of the result, and their lineage can be used to selectively compute confidence values for tuples of interest. If the latter is still too expensive, we can use approximate techniques like the Monte Carlo simulations proposed in [33] to estimate the confidences.

We do incur some overhead when confidences are finally computed, particularly if we follow the most naive approach of tracing the entire lineage of each result x-tuple alternative to obtain the base data confidences. The following are a few methods for optimizing the computation, two of which are supported in our implementation:

- The confidence value for a derived alternative can be computed from confidence values for a set of "closest independent descendants" (CIDs) for the alternative, rather than from confidence values on base data. Roughly, the CID of an alternative a is a minimal set S of alternatives in a's transitive lineage such that the alternatives in S do not share a common base alternative in their transitive lineage. It can be shown that CIDs are unique, and for more complex types of lineage, recursive computation of confidence values based on CIDs can be much cheaper than not using CIDs.
- CIDs also enable *memoization*, which avoids performing redundant confidence computations. Memoization can be useful within the computation for a single alternative, as well as across confidence computations, as long as intervening updates don't alter the relevant lineage or confidences.
- If transitive lineage λ^* is already being maintained for eager D-minimization (Sect. 4.3.1), it can then also be applied to considerably speed up confidence computations.
- So far we have discussed computing the confidence value for a single alternative. In the case where we wish to compute confidences for an x-tuple or an entire x-relation, batch techniques can be used based on the structure guaranteed by well-behaved lineage.

6 The Trio system

We now describe the Trio system, our implementation of a relational DBMS that supports uncertainty and lineage. The Trio system is based on the ULDB data model, and accepts queries in the *TriQL* language [44], our extension of SQL with uncertainty and lineage-specific features. The Trio system already supports richer lineages and wider classes of queries than those studied in this paper. Namely, lineages can be propositional and queries are not necessarily DL-monotonic. We focus here on the parts of the system that support the features of ULDBs discussed in previous sections. Further extensions are described in [38,44].

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

We first present the general architecture of Trio-One (Sect. 6.1) and the encoding of ULDB data (Sect. 6.2), then present TriQL queries, their evaluation, and the features they add to SQL (Sect. 6.3). Finally, we discuss the implementation of the additional operations supported by the system, such as confidence computation, and extraneous data removal (Sect. 6.4).

6.1 General architecture

Figure 3 shows the basic three-layer Trio-One architecture. The core system is implemented in Python and mediates between the underlying relational DBMS (currently the *PostgreSQL* open-source DBMS) and Trio interfaces and applications. The Python layer presents a simple Trio API that extends the standard Python DB 2.0 API for database access

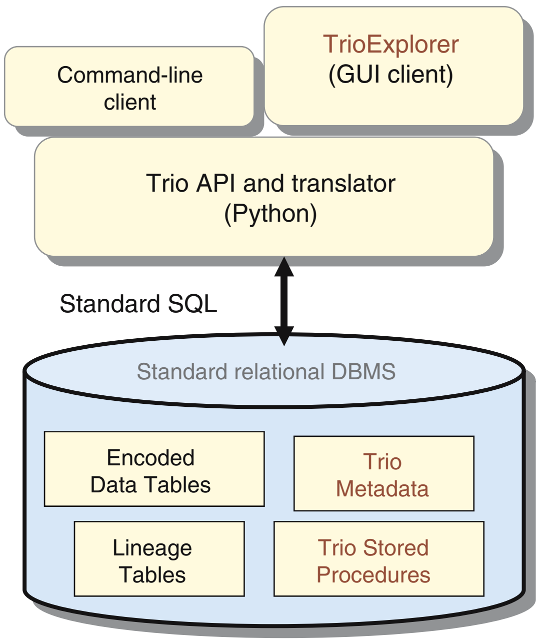


Fig. 3 System Architecture

(Python's analog of JDBC). The Trio API accepts TriQL queries (see Sect. 6.3) in addition to regular SQL, and query results may be x-tuples as well as regular tuples. The API also exposes *lineage tracing*, along with the other ULDB-specific operations discussed in Sects. 4 and 5. Using the Trio API, we built a generic command-line interactive client similar to that provided by most DBMS's, and a full-featured graphical user interface called *TrioExplorer*.

TrioExplorer offers a rich interface for interacting with the Trio system. It implements a Python-generated, multi-threaded web server using the *CherryPy* framework [15], and it supports multiple users logged into private and/or shared databases. It accepts Trio DDL and DML commands and provides numerous features for browsing and exploring schema, data, uncertainty, and lineage. It also enables on-demand confidence computation, coexistence checks, and extraneous data removal. Finally, it supports loading of scripts, command recall, and other user conveniences. Figure 4 shows a snapshot of TrioExplorer's schema visualizer, which displays schema-level lineage relationships among tables.

Trio DDL commands are translated via Python to SQL DDL commands based on the encoding of data described in Sect. 6.2. The translation is fairly straightforward, as is the corresponding translation of `insert` statements and bulk load.

TriQL query processing is discussed in Sect. 6.3.1. TriQL query results can either be *stored* or *transient*. Stored query results are placed in a new persistent table, and lineage relationships from the query's result data to data in the query's input tables also is stored persistently. Transient query results are accessed through the Trio API in a typical cursor-oriented fashion, with an additional method that can be invoked to explore the lineage of each returned tuple. For transient queries, query result processing and lineage creation occurs in response to cursor *fetch* calls, and neither the result data nor its lineage are persistent.

6.2 Encoding ULDB data

We now describe how ULDB databases are encoded in regular relational tables. Hereafter we use *x-tuple* to refer to a tuple in the ULDB model, and *tuple* to denote a regular relational tuple.

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

- `aid` is a unique alternative identifier (across the table)
- `xid` identifies the x-tuple that this alternative belongs to (also across the table)
- `conf` stores the confidence of the alternative, or `NULL` if there are no confidence values or if this confidence value has not yet been computed. (Each table either permits confidence values on all alternatives or on none of them; this *table type* is part of the schema information.)
- `num` is a nonnegative integer that tracks whether the alternative's x-tuple has a "?". Our scheme essentially maintains the invariant that an alternative's x-tuple has a "?" if and only if its `num` field exceeds the x-tuple's number of alternatives. (Sect. 6.3.1 explains how this field is used to propagate "?" annotations during query processing, and to avoid unnecessary grouping for certain data.)

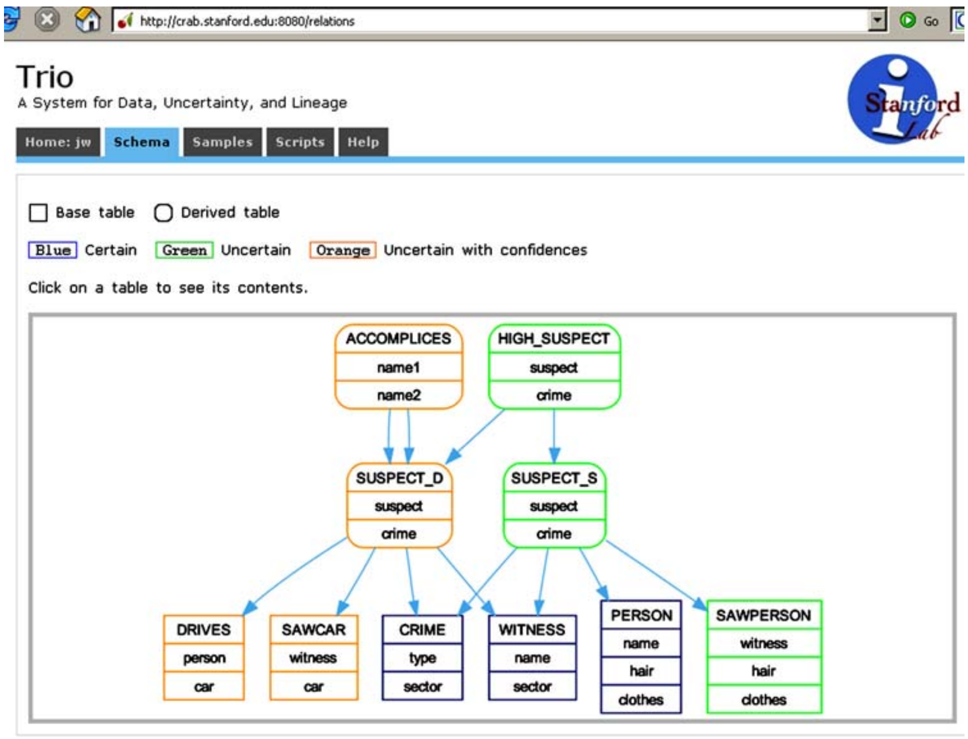
The system always creates indexes on `aid` and `xid`. In addition, Trio users may create indexes on any of the original data attributes $A_1, \ldots, A_n$ using standard `CREATE INDEX` commands that are simply passed through Trio to the underlying DBMS.

The lineage information for each table T is stored in a separate table $lin\_T($`aid`, `src_aid`, `src_table`$)$, indexed on `aid` and `src_aid`. A tuple (a_1, a_2, T_2) in lin_T denotes that T's alternative a_1 has alternative a_2 from table T_2 in its lineage.

To illustrate, the ULDB of Example 3 can be encoded as follows:

Example 9 Relational encoding of the ULDB of Example 3. The `num` and `conf` attributes are placed at the end of the tables for readability.

Fig. 4 TrioExplorer Screenshot



Saw

aid	xid	witness	car	num	conf
211	21	Amy	Mazda	3	NULL
212	21	Amy	Toyota	3	NULL
221	22	Betty	Honda	1	NULL

Drives

aid	xid	person	car	num	conf
311	31	Jimmy	Mazda	1	NULL
321	32	Jimmy	Toyota	1	NULL
331	33	Billy	Mazda	1	NULL
341	34	Billy	Honda	1	NULL

Accuses

aid	xid	witness	person	num	conf
411	41	Amy	Jimmy	3	NULL
421	42	Amy	Jimmy	3	NULL
431	43	Amy	Billy	3	NULL
441	44	Betty	Billy	1	NULL

Lin_Accuses

aid	src_aid	src_table
411	211	Saw
411	311	Drives
421	212	Saw
421	321	Drives
431	211	Saw
431	331	Drives
441	231	Saw
431	341	Drives

Note that the first three x-tuples in `Accuses` have a "?" (their single alternative has `num` > 1), while x-tuple 44 has no "?". We saw in Example 6, that the "?" on x-tuple 44 was extraneous. In the next section we see how the `num` value is computed.

The same ULDB with confidences would only vary by the presence of numerical values in the `conf` column, adding up to less than or equal to 1 for alternatives of the same x-tuple, and consistent with the num annotation.

6.3 Trio queries

TriQL [7,44], Trio's query language for ULDBs, is an extension of SQL. TriQL queries return uncertain relations in the ULDB model, with lineage that connects query result data to the queried data. As mentioned in Sect. 6.1, a TriQL query result may be *transient*, offering a cursor interface and a special method for retrieving lineage, or the query result and its lineage may be stored in persistent tables according to the encoding scheme described in Sect. 6.2. As a first example, the join query from Sect. 3 with its result stored in table `Accuses` would be written in TriQL simply as:

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

In Example 9, tables `Accuses` and `Lin_Accuses` are the (stored) result of the above query.

In addition to modifying SQL semantics for ULDBs, TriQL adds a number of new constructs for querying and manipulating both uncertainty and lineage. A comprehensive specification for TriQL's query and update language appears in [44]. In the remainder of this section, we use examples to illustrate TriQL semantics and functionality, and how TriQL queries are rewritten automatically into standard SQL over the relationally-encoded ULDB data.

6.3.1 Basic rewriting scheme

Trio essentially evaluates queries on ULDBs using the algorithm we presented in Sect. 4.2. The system leverages the encoding of ULDBs presented above to (1) make the bookkeeping of lineages, "?" annotations and confidences efficient, and (2) delegate as much of the processing to the query engine of the underlying DBMS.

TriQL query processing proceeds in two phases. In the *translation* phase, a TriQL parse tree is created and progressively transformed into a tree representing one or more standard SQL statements, based on the data encoding scheme. In the *execution* phase, the SQL statements are executed against the relational database encoding. Depending on the original TriQL query, Trio stored procedures may be invoked and some post-processing may occur.

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

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

Let *Tfetch* denote a cursor call to the Trio API for the original TriQL query, and let *Sfetch* denote a cursor call to the underlying DBMS for the translated SQL query. Each call to *Tfetch* must return a complete x-tuple, which may entail several calls to *Sfetch*: Each tuple returned from *Sfetch* on the SQL query corresponds to one alternative in the TriQL query result, and the set of alternatives with the same returned `Saw.xid` and `Drives.xid` pair comprise a single result x-tuple. (The TriQL operational join semantics presented in [7] makes this property very clear.) Thus, on *Tfetch*, Trio collects all SQL result tuples for a single `Saw.xid`/`Drives.xid` pair (enabled by the `ORDER BY` clause in the SQL query), generates a new `xid` and new `aid`'s, and constructs and returns the result x-tuple.

Note that the underlying SQL query also returns the `aid`'s from `Saw` and `Drives`. These values (together with the table names) comprise the lineage for the alternatives in the result x-tuple.

To propagate the "?" annotations (encoded in the `num` field), the query simply multiplies the `num` values of the underlying base tuples. Note that this propagation avoids generating extraneous "?" annotations for the obvious case where the underlying x-tuples are not maybe x-tuples. This is the case in Example 9 of tuple 441 (x-tuple 44) in `Accuses`. However, extraneous "?" annotations (and alternatives) may still exist because of intricate lineage dependencies, as discussed in Sect. 4.3.1.

The `num` field is a also useful when processing certain data. The generation of a new `xid` and `aid` for alternatives in the result is trivial if we can detect that they are certain. This is done by checking whether `num` $= 1$, which is the case for result tuples produced exclusively from base tuples having `num` $= 1$. Therefore, when processing data that is certain from the start, the overhead induced by ULDBs is very small.

Finally, since result confidence values for joins are not computed until they are explicitly requested (see Sect. 6.4), *Tfetch* initially returns `NULL` confidence for all alternatives, whether or not the query result logically contains confidence values.

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

6.3.2 Built-in predicates and functions

TriQL goes beyond SQL by offering constructs to query the lineage, maybe annotations and confidences of ULDB data. Because these additions keep queries DL-monotonic (see Sect. 4.1), the basic rewriting scheme presented above can be easily extended to support them.

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

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

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

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

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

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

Function `Conf()` is implemented as an SPI stored procedure. If it has just one argument `T`, the procedure first examines the current `T.conf` field to see if a value is present. If so, that value is returned. If `T.conf` is `NULL`, on-demand confidence computation is invoked (see Sect. 6.4.2), and the resulting confidence value is stored permanently and returned. `Conf(*)` always activates confidence computation, and includes the resulting confidence value in the query result (instead of `NULL`) as well as returning it from the function.

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

Predicate `Lineage(X,Y)` is translated into one or more SQL subqueries that check if the lineage relationship holds: Schema-level lineage information is used to determine the possible table-level "paths" from `X` to `Y`. Each path produces a subquery that joins lineage tables along that path, with `X` and `Y` at the endpoints. Suppose for the sake of illustration that a table `Saw2` was derived from `Saw`, and then `Accuses` was derived from `Saw2`. Then `Lineage(Accuses,Saw)` would be translated as follows, recalling the lineage encoding described in Sect. 6.2.

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

6.3.3 Querying uncertainty

"Horizontal" subqueries in TriQL enable querying across the alternatives that comprise individual x-tuples, i.e., *across possible worlds*. While horizontal queries depend on the particular ULDB at hand, they are a first foray into the (so far mainly unexplored) realm of queries on the uncertainty of data. As an (admittedly contrived) example, we can select from table `Saw` all vehicles sighted that are not Mazdas, but a Mazda sighting appears as another alternative of the same x-tuple:

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

On the ULDB of Example 3, this query would return a maybe x-tuple with `Toyota`, the car from the second alternative of tuple 21.

In general, enclosing a subquery in `[]` instead of `()` causes the subquery to be evaluated over the "current" x-tuple, treating its alternatives as if they are a table. Syntactic shortcuts are provided for common cases, such as simple filtering predicates as in the example above. Full details of horizontal subqueries and numerous examples can be found in [44].

Horizontal subqueries are very powerful, but surprisingly easy to implement based on our data encoding. First, syntactic shortcuts are expanded. In our example above, `[car = 'Mazda']` is a shortcut for `[SELECT * FROM Saw WHERE car='Mazda']`. Here, `Saw` within the horizontal subquery refers to the `Saw` alternatives in the current x-tuple being evaluated [44].) Second, the horizontal subquery is replaced with a standard SQL subquery that adds aliases for inner tables and a condition correlating `xid`'s with the outer query:

```
SQL> ... AND EXISTS (SELECT * FROM Saw S
SQL>                 WHERE car = 'Maz\-da'
SQL>                 AND S.xid = Saw.xid)
```

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

6.3.4 Query-defined result confidences

By default, confidence values on query results respect a probabilistic interpretation, and they are computed by the system on-demand. (A "COMPUTE CONFIDENCES" clause can be added to a query force confidence computation as part of query execution.) Algorithms for probabilistic confidence computation are discussed in Sect. 6.4.2.

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

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

Recall from Sect. 6.2 that our data encoding scheme adds a column `conf` to each underlying table to store confidence values. Consequently, "`AS conf`" clauses simply pass through the query translation phase unmodified.

6.4 Additional Trio features

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

6.4.1 Lineage

As TriQL queries are executed and their results are stored, and additional queries are posed over previous results, complex lineage relationships can arise. As we have seen, data-level lineage is used for confidence computation and `Lineage()` predicates; it is also used for coexistence checks (Sect. 6.4.3) and extraneous data removal (Sect. 6.4.4). Trio also maintains a *schema-level lineage graph* that is used for `Lineage()` predicate translation (Sect. 6.3.2) and for some confidence-computation optimizations. This graph can also be a useful tool for the user; it is depicted in the TrioExplorer screenshot of Fig. 4.

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

6.4.2 Confidence computation

Recall that each possible instance of a ULDB has a probability based on the confidences of the data in that instance. In query results, lineage ties the possible result instances to the possible instances of the queried data. Thus, using lineage, each result alternative has a confidence value that captures the fraction of possible instances in which its lineage appears. This confidence value is correctly computed by constructing an alternative's lineage formula in terms of base data (i.e., the result of the `BaseLineage()` method described above) and then evaluating the probability of the formula using the confidence values on the base alternatives, as explained in Sect. 5.

Thus, when confidence computation is invoked for an alternative a, the system effectively invokes `BaseLineage(`a`)` and then evaluates the probability of the resulting formula using base-data confidences.

As discussed in Sect. 5.3, a number of optimizations to this simple scheme are possible. CIDs are currently being investigated and implemented. For now the system supports the following optimizations:

- Whenever confidence values are computed, they are *memoized* for future use.
- We have developed algorithms for *batch* confidence computation that are implemented through SQL queries. These algorithms are appropriate and efficient when confidence values are desired for a significant portion of a result table.

6.4.3 Coexistence checks

A user may wish to select a set of alternatives from one or more tables and ask whether those alternatives can all coexist. Two alternatives from the same x-tuple clearly cannot coexist, but the general case must take into account arbitrarily complex lineage relationships as well as tuple alternatives. For example, if we asked about alternatives (42,1) and (43,1) in Example 3, the system would tell us these alternatives cannot coexist. Coexistence checking can be performed by generating base-lineage formulas for the set of alternatives, augmenting them with formulas capturing mutual exclusion of tuple-alternatives, and then checking satisfiability.

6.4.4 Extraneous data removal

As seen in Sect. 4.3, the natural execution of TriQL queries can generate *extraneous data*: a tuple alternative is extraneous if it can never be chosen (i.e., its lineage includes the conjunction of data that cannot coexist); a "?" annotation is extraneous if its tuple is always present. It is possible to check for extraneous alternatives and ?'s immediately after query execution (and, sometimes, as part of query execution). However, like confidence computation and coexistence checks, extraneous data detection may require tracing lineage to the base data. Because we expect extraneous data and ?'s to be relatively uncommon, and users may not be concerned about their presence, we have chosen to implement extraneous data removal as a separate operation, roughly akin to garbage collection.

The astute reader may note that all of the features discussed in this section are interconnected. In fact they share code in the system, and they can share some of the optimizations discussed in Sects. 6.4.2 and 5.3 as well. For example, we can determine if an alternative is extraneous by computing its confidence and checking if it is equal to 0, while conversely a "?" is extraneous if the confidence values for its tuple sum to 1. Similarly, a set of alternatives can coexist iff, when treated as conjunctive lineage for a dummy alternative a, the confidence of a is >0.

7 Related work

In [46], we described the original motivation that led to the work in this paper: development of a general-purpose database management system that incorporates data, lineage, and uncertainty. In [23], we explored the space of incomplete and complete models for uncertainty, without considering lineage. In [24], we posed and solved a number of new theoretical problems with respect to representation schemes for uncertainty, again without lineage.

We are not aware of any previously proposed formal data representation that integrates both lineage and uncertainty. We briefly overview some of the work that addresses uncertainty and lineage independently.

Representation schemes and query answering for uncertain databases has been studied extensively, e.g., [2,4,5,10,27,29–31,34,45]. Much of this previous work is theoretical, but there has been recent interest in building systems, e.g., [9,14,46] for uncertainty, and [32,42] for integrating inconsistent data sources. Query answering in probabilistic databases has seen considerable progress and efficient solutions have been proposed [20–22]. We build on that work in this paper, showing how lineage can further improve query processing.

Approximate query answering has also received significant attention over the last decade [3,26,28,43], but we focus on exact queries over uncertain data rather than inexact queries over certain data. However, the simple representation of uncertainty in ULDBs is likely to facilitate approximate querying, and we plan to investigate this avenue of future work.

Integrating lineage (also known as *provenance*) has been proposed for relational databases, e.g., [11,39,40], and for data warehouses, e.g., [17–19]. It has been observed that there are various choices in defining lineage, and in this paper we use a definition similar to the *where lineage* of [11]. The main differences with [11] are that (1) our model is relational while theirs is tree-based and (2) we track and persist lineage at the granularity of tuples, while they track it at the granularity of individual values inside tuples. The finer granularity of lineage they consider is necessary to explain why and how individual values appear in query results. Our lineage model is simpler, but sufficient to track uncertainty. Moreover, we persist lineage as part of the database, which enables queries to be posed against the lineage itself.

Analysis of possible lineage information was also used for optimizing query evaluation and determining independence of queries from updates [36]. A recent system being developed around data provenance is described in [8,16].

8 Conclusions and future work

We introduce ULDBs as a representation for databases with both lineage and uncertainty. With simple extensions to the relational model (tuple alternatives, maybe tuples, and lineage functions), ULDBs can represent any finite set of possible instances containing data and lineage, and ULDBs are amenable to efficient query processing using standard relational techniques. ULDBs can be extended naturally to represent and query probabilistic data; moreover, because lineage enables query evaluation to be decoupled from the computation of confidences, substantial performance gains may be achieved over computing query operators and confidences in tandem.

In this paper, we focused on a specific class of DL-monotonic queries and their lineage. We are extending our techniques and results to a larger set of operations, e.g., duplicate-elimination, aggregation, and negation. Doing so primarily entails extending the types of lineage allowed, e.g., adding disjunctive and negative lineage, as briefly shown in Sect. 5.3.

As described in Sect. 6, Trio is currently implemented on top of a standard relational DBMS, and is able through simple rewriting techniques to evaluate DL-monotonic queries on ULDBs without altering any system internals. However, new techniques are required if we are to handle all aspects of ULDBs covered in this paper, e.g., keeping a ULDB D-minimized as query results are added, and efficiently L-minimizing the result of an extraction.

We highlighted throughout the paper how the main features of ULDBs are crucial for data integration applications: lineage is indispensable to track the origin of data and its journey through data integration operations. Uncertainty is inherent to almost every single one of these operations. We also stressed the importance of ULDB features such as extraction and confidence computations in a data integration context.

While ULDBs provide essential primitives for data integration, they are not expressive enough to fully represent the effects on data of some of its complex operations. Well-behaved lineage is suitable to represent traditional relational queries, but confines the uncertainty to base data, and falls short to capture operations which generate additional uncertainty. Whether the right approach is to extend ULDBs with richer primitives to support data integration (at the price of a greater complexity), or to keep data integration “outside the ULDB box” and transfer the lineage and uncertainty primitives to data integration systems remains unclear, and is an interesting question we plan to tackle.

Another challenging question in the context of data integration is the relationship between the data in a ULDB and the data in the data sources. We mentioned the possible use of extraction to define a flexible boundary between the ULDB and the external world. The open question of L-minimization is obviously also relevant to making this interface efficient.

Beyond those, the changing nature of external data sources needs to be accounted for. Revising the (uncertain) knowledge of a ULDB based on source changes, and propagating it (through lineage) to the derived x-relations of the ULDB is another important question on our future research agenda.

There are a number of other current and future directions of work in ULDBs:

- **Updates:** We are currently identifying a set of update primitives for ULDBs, and considering the design of efficient update algorithms.
- **Implementation:** ULDBs introduce several new physical design issues, such as data layout, indexing, partitioning, and materialized views, and their integration into query optimization. Fully exploring these topics is likely to entail modifying our prototype to operate inside (instead of on top of) a DBMS.
- **Theory:** There are numerous interesting theoretical problems to work on. We can reconsider nearly every topic in relational database theory in the context of ULDBs, e.g., dependency theory, query containment, and sampling and statistics.
- **Long-Term goals:** Our agenda for the overall Trio project [46] includes several features not yet present in ULDBs, such as uncertainty in the form of continuous distributions, incomplete relations, and versioning of data, uncertainty, and lineage.

Acknowledgments We are grateful to Dan Suciu and Jeff Ullman for useful discussions, and to Chris Hayworth, Ander de Keijzer, Michi Mutsuzaki, and Tomoe Sugihara for their extensive work on implementation.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, New York (1995)
2. Abiteboul, S., Kanellakis, P., Grahne, G.: On the representation and querying of sets of possible worlds. Theor. Comput. Sci. **78**(1), 137–158 (1991)
3. Agrawal, S., Chaudhuri, S., Das, G., Gionis, A.: Automated ranking of database query results. In: Proc. of CIDR (2003)
4. Barbará, D., Garcia-Molina, H., Porter, D.: The management of probabilistic data. IEEE Trans. Knowl. Data Eng. **4**(5), 487–502 (1992)
5. Barga, R.S., Pu, C.: Accessing imprecise data: an approach based on intervals. IEEE Data Eng. Bull. **16**(2), 12–15 (1993)
6. Benjelloun, O., Das Sarma, A., Halevy, A., Widom, J.: ULDBs: databases with uncertainty and lineage. In: VLDB, pp. 953–964 (2006)
7. Benjelloun, O., Das Sarma, A., Hayworth, C., Widom, J.: An introduction to ULDBs and the Trio system. IEEE Data Eng. Bull. **29**(1), 5–16 (2006)
8. Bhagwat, D., Chiticariu, L., Tan, W., Vijayvargiya, G.: An annotation management system for relational databases. In: Proc. of VLDB (2004)
9. Boulos, J., Dalvi, N., Mandhani, B., Mathur, S., Re, C., Suciu, D.: MYSTIQ: a system for finding more answers by using probabilities. In: Proc. of ACM SIGMOD (2005)
10. Buckles, B.P., Petry, F.E.: A fuzzy model for relational databases. Int. J. Fuzzy Sets Systems **7**, 213–226 (1982)
11. Buneman, P., Khanna, S., Tan, W.: Why and where: a charaterization of data provenance. In: Proc. of ICDT (2001)
12. Cavallo, R., Pittarelli, M.: The theory of probabilistic databases. In: Proc. of VLDB (1987)
13. Chang, K.C.C., He, B., Zhang, Z.: Toward large scale integration: building a metaquerier over databases on the web. In: Proc. of CIDR, pp. 44–55 (2005)
14. Cheng, R., Singh, S., Prabhakar, S.: U-DBMS: A database system for managing constantly-evolving data. In: Proc. of VLDB (2005)
15. The CherryPy web development framework. http://www.cherrypy.org
16. Chiticariu, L., Tan, W., Vijayvargiya, G.: DBNotes: a post-it system for relational databases based on provenance. In: Proc. of ACM SIGMOD (2005)
17. Cui, Y., Widom, J.: Practical lineage tracing in data warehouses. In: Proc. of ICDE (2000)
18. Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. VLDB J. **12**(1), 41–58 (2003)
19. Cui, Y., Widom, J., Wiener, J.L.: Tracing the lineage of view data in a warehousing environment. ACM TODS **25**(2), 179–227 (2000)
20. Dalvi, N., Miklau, G., Suciu, D.: Asymptotic conditional probabilities for conjunctive queries. In: Proc. of ICDT (2005)
21. Dalvi, N., Suciu, D.: Efficient query evaluation on probabilistic databases. In: Proc. of VLDB (2004)
22. Dalvi, N., Suciu, D.: Answering queries from statistics and probabilistic views. In: Proc. of VLDB (2005)
23. Das Sarma, A., Benjelloun, O., Halevy, A., Widom, J.: Working models for uncertain data. In: Proc. of ICDE (2006)
24. Das Sarma, A., Nabar, S., Widom, J.: Representing uncertain data: uniqueness, equivalence, minimization, and approximation. Tech. rep., Stanford InfoLab (2005). Available at http://dbpubs.stanford.edu/pub/2005-38
25. Das Sarma, A., Theobald, M., Widom, J.: Exploiting lineage for confidence computation in uncertain and probabilistic databases. Tech. rep., Stanford InfoLab (2007). Available on http://dbpubs.stanford.edu
26. Fuhr, N.: A probabilistic framework for vague queries and imprecise information in databases. In: Proc. of VLDB (1990)
27. Fuhr, N., Rölleke, T.: A probabilistic NF2 relational algebra for imprecision in databases. Unpublished Manuscript (1997)
28. Fuhr, N., Rölleke, T.: A probabilistic relational algebra for the integration of information retrieval and database systems. ACM TOIS **14**(1), 32–66 (1997)
29. Grahne, G.: Dependency satisfaction in databases with incomplete information. In: Proc. of VLDB (1984)
30. Grahne, G.: Horn tables—an efficient tool for handling incomplete information in databases. In: Proc. of ACM PODS (1989)
31. Imielinski, T., Lipski, W. Jr.: Incomplete information in relational databases. J. ACM **31**(4), 761–791 (1984)
32. Ives, Z.G., Khandelwal, N., Kapur, A., Cakir, M.: Orchestra: rapid, collaborative sharing of dynamic data. In: Proc. of CIDR (2005)
33. Karp, R.M., Luby, M.: Monte Carlo algorithms for enumeration and reliability problems. In: Proc. of FOCS (1983)
34. Lakshmanan, L.V.S., Leone, N., Ross, R., Subrahmanian, V.: ProbView: a flexible probabilistic database system. ACM TODS **22**(3), 419–469 (1997)
35. Levy, A.Y., Fikes, R.E., Sagiv, S.: Speeding up inferences using relevance reasoning: a formalism and algorithms. Artif. Intell. **97**(1–2), 83–136 (1997)
36. Levy, A.Y., Sagiv, Y.: Queries independent of updates. In: Proc. of VLDB (1993)

37. Madhavan, J., Cohen, S., Dong, X.L., Halevy, A.Y., Jeffery, S.R., Ko, D., Yu, C.: Web-scale data integration: you can afford to pay as you go. In: Proc. of CIDR, pp. 342–350 (2007)
38. Mutsuzaki, M., Theobald, M., de Keijzer, A., Widom, J., Agrawal, P., Benjelloun, O., Sarma, A.D., Murthy, R., Sugihara, T.: Trio-one: layering uncertainty and lineage on a conventional dbms (system demonstration). In: Proc. of CIDR, pp. 269–274 (2007)
39. Buneman, P., Khanna, S., Tan, W.: Data provenance: some basic issues. In: Proc. of FSTTCS (2000)
40. Buneman, P., Khanna, S., Tan, W.: On propagation of deletions and annotations through views. In: Proc. of ACM PODS (2002)
41. Tao, Y., Cheng, R., Xiao, X., Ngai, W.K., Kao, B., Prabhakar, S.: Indexing multi-dimensional uncertain data with arbitrary probability density functions. In: Proc. of VLDB (2005)
42. Taylor, N.E., Ives, Z.G.: Reconciling while tolerating disagreement in collaborative data sharing. In: Proc. of ACM SIGMOD (2006)
43. Theobald, A., Weikum, G.: The XXL search engine: ranked retrieval of xml data using indexes and ontologies. In: Proc. of ACM SIGMOD (2002)
44. TriQL: The Trio query language. Available from http://infolab.stanford.edu/trio
45. Vardi, M.Y.: Querying logical databases. In: Proc. of ACM PODS (1985)
46. Widom, J.: Trio: a system for integrated management of data, accuracy, and lineage. In: Proc. of CIDR (2005)