Probabilistic Databases: Diamonds in the Dirt*

Nilesh Dalvi
Yahoo!Research
USA
ndalvi@yahoo-inc.com

Christopher Ré
University of Washington
USA
chrisre@cs.washington.edu

Dan Suciu
University of Washington
USA
suciu@cs.washington.edu

1. INTRODUCTION

A wide range of applications have recently emerged that need to manage large, imprecise data sets. The reasons for imprecision in data are as diverse as the applications themselves: in sensor and RFID data, imprecision is due to measurement errors [15, 34]; in information extraction, imprecision comes from the inherent ambiguity in natural-language text [20, 26]; and in business intelligence, imprecision is tolerated because of the high cost of data cleaning [5]. In some applications, such as privacy, it is a requirement that the data be less precise. For example, imprecision is purposely inserted to hide sensitive attributes of individuals so that the data may be published [30]. Imprecise data has no place in traditional, precise database applications like payroll and inventory, and so, current database management systems are not prepared to deal with it. In contrast, the newly emerging applications offer value precisely because they query, search, and aggregate large volumes of imprecise data to find the "diamonds in the dirt". This wide-variety of new applications points to the need for generic tools to manage imprecise data. In this paper, we survey the state of the art of techniques that handle imprecise data, by modeling it as probabilistic data [2–4, 7, 12, 15, 23, 27, 36].

A *probabilistic database management system*, or ProbDMS, is a system that stores large volumes of probabilistic data and supports complex queries. A ProbDMS may also need to perform some additional tasks, such as updates or recovery, but these do not differ from those in conventional database management systems and will not be discussed here. The major challenge in a ProbDMS is that it needs both to *scale* to large data volumes, a core competence of database management systems, and to do *probabilistic inference*, which is a problem studied in AI. While many scalable data management systems exists, probabilistic inference is a hard problem [35], and current systems do not scale to the same extent as data management systems do. To address this challenge, researchers have focused on the specific nature of relational probabilistic data, and exploited the special form of probabilistic inference that occurs during query evaluation. A number of such results have emerged recently: lineage-based representations [4], safe plans [11], algorithms for top-k queries [31, 37], and representations of views over probabilistic data [33]. What is common to all these results is that they apply and extend well known concepts that are fundamental to data management, such as the separation of query and data when analyzing complexity [38], incomplete databases [22], the threshold algorithm [16], and the use of materialized views to answer queries [21]. In this paper, we briefly survey the key concepts in probabilistic database systems, and explain the intellectual roots of these concepts in data management.

1.1 An Example: The Purple Sox System

We illustrate using an example from an information extraction system. The Purple Sox[1] system at Yahoo! Research focuses on technologies to extract and manage structured information from the Web related to a specific community. An example is the DbLife system [14] that aggregates structured information about the database community from data on the Web. The system extracts lists of database researchers together with structured, related information such as publications that they have authored, their co-author relationships, talks that they have given, their current affiliations, and their professional services. Figure 1(a) illustrates the researchers' affiliations, and Figure 1(b) illustrates their professional activities. Although most researchers have a single affiliation, in the data in Figure 1(a), the extracted affiliations are not unique. This occurs because outdated/erroneous information is often present on the Web, and even if the extractor is operating on an up-to-date Webpage, the difficulty of the extraction problem forces the extractors to produce many alternative extractions or risk missing valuable data. Thus, each `Name` contains several possible affiliations. One can think of `Affiliation` as being an attribute with uncertain values. Equivalently, one can think of each row as being a separate uncertain tuple. There are two constraints on this data: tuples with the same `Name` but different `Affiliation` are mutually exclusive; and tuples with different values of `Name` are independent. The professional services shown in Figure 1 (b) are extracted from conference Webpages, and are also imprecise: in our example, each record in this table is an independent extraction and assumed to be independent.

In both examples, the uncertainty in the data is repre-

*This work was partially supported by NSF Grants IIS-0454425, IIS-0513877, IIS-0713576, and a Gift from Microsoft. An extended version of this paper with additional references is available at `http://www.cs.washington.edu/homes/suciu/`.

[1] `http://research.yahoo.com/node/498`

Researchers:

	Name	Affiliation	P	
t_1^1	Fred	U. Washington	$p_1^1 = 0.3$	$X_1 = 1$
t_1^2		U. Wisconsin	$p_1^2 = 0.2$	$X_1 = 2$
t_1^3		Y! Research	$p_1^3 = 0.5$	$X_1 = 3$
t_2^1	Sue	U. Washington	$p_2^1 = 1.0$	$X_2 = 1$
t_3^1	John	U. Wisconsin	$p_3^1 = 0.7$	$X_3 = 1$
t_3^2		U. Washington	$p_3^2 = 0.3$	$X_3 = 2$
t_4^1	Frank	Y! Research	$p_4^1 = 0.9$	$X_4 = 1$
t_4^2		M. Research	$p_4^2 = 0.1$	$X_4 = 2$

(a)

Services:

	Name	Conference	Role	P	
s_1	Fred	VLDB	Session Chair	$q_1 = 0.2$	$Y_1 = 1$
s_2	Fred	VLDB	PC Member	$q_2 = 0.8$	$Y_2 = 1$
s_3	John	SIGMOD	PC Member	$q_3 = 0.7$	$Y_3 = 1$
s_4	John	VLDB	PC Member	$q_4 = 0.7$	$Y_4 = 1$
s_5	Sue	SIGMOD	Chair	$q_5 = 0.5$	$Y_5 = 1$

(b)

Figure 1: Example of a probabilistic database. This is a *block-independent-disjoint* database: the 8 tuples in Researchers are grouped in four groups of disjoint events, e.g., t_1^1, t_1^2, t_1^3 are disjoint, and so are t_4^1, t_4^2, while tuples from different blocks are independent, e.g., t_1^2, t_2^2, t_4^1 are independent; the five tuples in Services are independent probabilistic events. This database can be represented as a c-table using the hidden variables X_1, X_2, X_3, X_4 for Researchers and Y_1, Y_2, Y_3, Y_4, Y_5 for Services.

sented as a probabilistic confidence score, which is computed by the extractor. For example, Conditional Random Fields produce extractions with semantically meaningful confidence scores [20]. Other sources of uncertainty can also be converted to confidence scores, for example probabilities produced by *entity matching algorithms* (does the mention *Fred* in one Webpage refer to the same entity as *Fred* in another Webpage?). The example in Figure 1 presents a very simplified view of a general principle: uncertain data is annotated with a confidence score, which is interpreted as a probability. In this paper we use "probabilistic data" and "uncertain data" as synonyms.

1.2 Facets of a ProbDMS

There are three important, related facets of any ProbDMS: (1) How do we store (or represent) a probabilistic database? (2) How do we answer queries using our chosen representation? (3) How do we present the result of queries to the user?

There is a tension between the power of a representation system, i.e., as the system more faithfully models correlations, it becomes increasingly difficult to scale the system. A simple representation where each tuple is an independent probabilistic event is easier to process, but it cannot faithfully model the correlations important to all applications. In contrast, a more complicated representation, e.g., a large Markov Network [9], can capture the semantics of the data very faithfully, but it may be impossible to compute even simple SQL queries using this representation. An extra challenge is to ensure that the representation system maps smoothly to relational data, so that the non-probabilistic part of the data can be processed by a conventional database system.

A ProbDMS needs to support complex, decision-support style SQL, with aggregates. While some applications can benefit from point queries, the real value comes from queries that search many tuples, or aggregate over many data values. For example the answer to *find the affiliation of PC Chair of SIGMOD'2008* is inherently imprecise (and can be answered more effectively by consulting the SIGMOD'2008 home page), but a query like *find all institutions (affiliations) with more than 20 SIGMOD and VLDB PC Members* returns more interesting answers. There are two logical steps in computing a SQL query on probabilistic data: first, fetch and transform the data, and second, perform probabilistic inference. A straightforward but naïve approach is to separate the two steps: use a database engine for the first step, and a general-purpose probabilistic inference technique [9, 13] for the second. But on large data the probabilistic inference quickly dominates the total running time. A better approach is to integrate the two steps, which allows us to leverage some database specific techniques, such as query optimization, using materialized views, and schema information, to speedup the probabilistic inference.

Designing a good user interface raises new challenges. The answer to a SQL query is a set of tuples, and it is critical to find some way to rank these tuples, because there are typically lots of false positives when the input data is imprecise. Alternatively, aggregation queries can extract value from imprecise data, because errors tend to cancel each other out (the Law of the Large Numbers). A separate and difficult task is how to indicate to the user the correlations between the output tuples. For example, the two highest ranked tuples may be mutually exclusive, but they could also be positively correlated. As a result, their ranks alone convey insufficient information to the user. Finally, a major challenge of this facet is how to obtain feedback from the users and how to employ this feedback to "clean" the underlying database. This is a difficult problem, which to date has not yet been solved.

1.3 Key Applications

Probabilistic databases have found usage in a wide class of applications. *Sensor data* is obtained from battery-powered sensors that acquire temperature, pressure, or humidity readings from the surrounding environment. The BBQ system [15] showed that a probabilistic data model could represent well this kind of sensor data. A key insight was that the probabilistic model could answer many queries with sufficient confidence without needing to acquire additional readings. This is an important optimization since acquiring fewer sensor readings allows longer battery life, and so more longer lasting sensor deployments. *Information Extraction* is a process that extracts data items of a given type from large corpora of text [26]. The extraction is always noisy, and the system often produces several alternatives.

Gupta and Sarawagi [20] have argued that such data is best stored and processed by a probabilistic database. In *Data Cleaning*, deduplication is one of the key components and is also a noisy and imperfect process. Andritsos, Fuxman, and Miller [1] have shown that a probabilistic database can simplify the deduplication task, by allowing multiple conflicting tuples to coexist in the database. Many other applications have looked at probabilistic databases for their data management needs: RFID data management [34], management of *anonymized data* [30] and scientific data management [28].

2. KEY CONCEPTS IN A PROBDMS

We present a number of key concepts for managing probabilistic data that have emerged in recent years. We group these concepts by the three facets, although some concepts may be relevant to more than one facet.

2.1 Facet 1: Semantics and Representation

The *de facto* formal semantics of a probabilistic database is the *possible worlds model* [12]. By contrast, there is no agreement on a representation system, instead there are several approaches covering a spectrum between expressive power and usability [4]. A key concept in most representation systems is that of *lineage*, which is derived from early work on incomplete databases by Immelinski and Lipski [22].

2.1.1 Possible Worlds Semantics

In its most general form, a probabilistic database is a probability space over the possible contents of the database. It is customary to denote a (conventional) relational database instance with the letter I. Assuming there is a single table in our database, I is simply a set of tuples (records) representing that table; this is a conventional database. A *probabilistic database* is a discrete probability space $PDB = (W, \mathbf{P})$, where $W = \{I_1, I_2, \ldots, I_n\}$ is a set of possible instances, called *possible worlds*, and $\mathbf{P} : W \rightarrow [0, 1]$ is such that $\sum_{j=1,n} \mathbf{P}(I_j) = 1$. In the terminology of networks of belief, there is one random variable for each possible tuple whose values are 0 (meaning that the tuple is not present) or 1 (meaning that the tuple is present), and a probabilistic database is a joint probability distribution over the values of these random variables.

This is a very powerful definition that encompasses all the concrete data models over discrete domains that have been studied. In practice, however, one has to step back from this generality and impose some workable restrictions, but it is always helpful to keep the general model in mind. Note that in our discussion we restrict ourselves to discrete domains: although probabilistic databases with continuous attributes are needed in some applications [7, 15], no formal semantics in terms of possible worlds has been proposed so far.

Consider some tuple t (we use interchangeably the terms *tuple* and *record* in this paper). The probability that the tuple belongs to a randomly chosen world is $\mathbf{P}(t) = \sum_{j:t \in I_j} \mathbf{P}(I_j)$, and is also called the marginal probability of the tuple t. Similarly, if we have two tuples t_1, t_2, we can examine the probability that *both* are present in a randomly chosen world, denoted $\mathbf{P}(t_1 t_2)$. When the latter is $\mathbf{P}(t_1)\mathbf{P}(t_2)$, we say that t_1, t_2 are independent tuples; if it is 0 then we say that t_1, t_2 are disjoint tuples or exclusive tuples. If none of these hold, then the tuples are correlated in a non-obvious way. Consider a query Q, expressed in some relational query language like SQL, and a possible tuple t in the query's answer. $\mathbf{P}(t \in Q)$ denotes the probability that, in a randomly chosen world, t is an answer to Q. The job of a probabilistic database system is to return all possible tuples $t_1, t_2, \ldots$ together with their probabilities $\mathbf{P}(t_1 \in Q), \mathbf{P}(t_2 \in Q), \ldots$

2.1.2 Representation Formalisms

In practice, one can never enumerate all possible worlds, and instead we need to use some more concise representation formalism. One way to achieve that is to restrict the class of probabilistic databases that one may represent. A popular approach is to restrict the possible tuples to be either independent or disjoint. Call a probabilistic database *block independent-disjoint*, or BID, if the set of all possible tuples can be partitioned into blocks such that tuples from the same block are disjoint events, and tuples from distinct blocks are independent. A BID database is specified by defining the partition into blocks, and by listing the tuples' marginal probabilities. This is illustrated in Figure 1. The blocks are obtained by grouping `Researchers` by `Name`, and grouping `Services` by `(Name,Conference,Role)`. The probabilities are given by the `P` attribute. Thus, the tuples t_1^2 and t_1^3 are disjoint (they are in the same block), while the tuples t_1^1, t_5^2, s_1, s_2 are independent (they are from different blocks). An intuitive BID model was introduced by Trio [4] and consists of *maybe-tuples*, which may or may not be in the database, and *x-tuples*, which are sets of mutually exclusive tuples.

Several applications require a richer representation formalism, one that can express complex correlations between tuples, and several such formalisms have been described in the literature: lineage-based [4, 18], U-relations [2], or the closure of BID tables under conjunctive queries [12]. Others are the Probabilistic Relational Model of Friedman et al. [17] that separates the data from the probabilistic network, and Markov Chains [25,34]. Expressive formalisms, however, are often hard to understand by users, and increase the complexity of query evaluation, which lead researchers to search for simpler, *workable* models for probabilistic data [4].

All representation formalisms are, at their core, an instance of database normalization: they decompose a probabilistic database with correlated tuples into several BID tables. This is similar to the factor decomposition in graphical models [9], and also similar to database normalization based on multivalued dependencies [39]. A first question is how to design the normal representation given a probabilistic database. This requires a combination of techniques from graphical models and database normalization, but, while the connection between these two theories was described by Verma and Pearl [39] in the early 1990s, to date there exists no comprehensive theory of normalization for probabilistic databases. A second question is how to recover the complex probabilistic database from its normalized representation as BID tables. This can be done through SQL views [12] or through lineage.

2.1.3 Lineage

The *lineage* of a tuple is an annotation that defines its derivation. Lineage is used both to represent probabilistic data, and to represent query results. The Trio system [4] recognized the importance of lineage in managing data with uncertainty, and called itself a ULDB, for *uncertainty-lineage database*. In Trio, when new data is produced by queries over uncertain data, the lineage is computed automatically and

captures all correlations needed for computing subsequent queries over the derived data.

Lineage also provides a powerful mechanism for understanding and resolving uncertainty. With lineage, user feedback on correctness of results can be traced back to the sources of the relevant data, allowing unreliable sources to be identified. Users can provide much detailed feedback if data lineage is made visible to them. For example, in information extraction applications where data items are generated by pipelines of AI operators, users can not only indicate if a data item is correct, but can look at the lineage of data items to locate the exact operator making the error.

The notion of lineage is derived from a landmark paper by Imielinski and Lipski [22] from 1984, who introduced *c-tables*, as a formalism for representing incomplete databases. We describe c-tables and lineage by using the example in Figure 2. In a c-table, each tuple is annotated with a Boolean expression over some hidden variables; today, we call that expression *lineage*. In our example there are three tuples, `U. of Washington`, `U. of Wisconsin`, and `Y! Research`, each annotated with a lineage expression over variables X_1, X_3, Y_1, Y_2, Y_4. The semantics of a c-table is a set of possible worlds. An assignment of the variables defines the world consisting of precisely those tuples whose lineage is `true` under that assignment, and the c-table "means" the set of possible worlds defined by all possible assignments. For an illustration, in our example any assignment containing $X_1 = 3, Y_2 = 1, X_3 = 2, Y_4 = 1$ (and any values for the other variables) defines the world {`Y! Research, U. of Washington`}, while any assignment with $Y_1 = Y_2 = Y_3 = 0$ defines the empty world.

Lineage is a powerful tool in ProbDMS because of the following important property: the answer to a query over a c-table can always be represented as another c-table, using the same hidden variables. In other words, it is always possible to compute the lineage of the output tuples from those of the input tuples. This is called a *closure property* and was first shown by Imielinski and Lipski [22]. We illustrate this property on the database in Fig. 1, where each tuple has a very simple lineage. Consider now the SQL query in Figure 3(a), which finds the affiliations of all people who performed some service for the `VLDB` conference. The answer to this query is precisely the c-table in Figure 2.

2.2 Facet 2: Query Evaluation

Query evaluation is the hardest technical challenge in a ProbDMS. One approach is to separate the query and lineage evaluation from the probabilistic inference on the lineage expression. Various algorithms have been used for the latter, such as Monte Carlo approximation algorithms [11, 31]. Recently, a much more general Monte Carlo framework has been proposed by Jampani et al. [23]. Variable Elimination [9] was used by Sen and Deshpande [36].

Another approach is to integrate the probabilistic inference with the query computation step. With this approach, one can leverage standard data management techniques to speed up the probabilistic inference, such as static analysis on the query or using materialized views. This has led to safe queries and to partial representations of materialized views, which we discuss next.

2.2.1 Safety

Two of the current authors showed that certain SQL queries can be evaluated on a probabilistic database by pushing the probabilistic inference completely inside the query plan [11]. Thus, for these queries there is no need for a separate probabilistic inference step: the output probabilities are computed inside the database engine, during normal query processing. The performance improvements can be large, e.g., Ré et al. [31] observed two orders of magnitude improvements over a Monte Carlo simulation. Queries for which this is possible are called *safe queries*, and the relational plan that computes the output probabilities correctly is called a *safe plan*. To understand the context of this result we review a fundamental principle in relational query processing: the separation between *what* and *how*.

In a relational query the user specifies *what* she wants: relational query languages like SQL are declarative. The system translates the query into relational algebra, using operators like join $\bowtie$, selection σ, projection-with-duplicate-elimination Π. The resulting expression is called a *relational plan* and represents *how* the query will be evaluated. The separation between *what* and *how* was first articulated by Codd when he introduced the relational data model [8], and is at the core of any modern relational database system. A *safe plan* allows probabilities to be computed in the relational algebra, by extending its operators to manipulate probabilities [12]. There are multiple ways to extend them, the simplest is to assume all tuples to be independent: a join $\bowtie$ that combines two tuples computes the new probability as $p_1 p_2$, and a duplicate elimination that replaces n tuples with one tuple computes the output probability as $1 - (1 - p_1) \cdots (1 - p_n)$. A *safe plan* is by definition a plan in which all these operations are provably correct. The correctness proof (or safety property) needs to be done by the query optimizer, through a static analysis on the plan. Importantly, safety does not depend on the actual instance of the database, instead, once a plan has been proven to be safe, it can be executed on any database instance.

We illustrate with the query in Figure3(a). Any modern relational database engine will translate it into the logical plan shown in (b). However, this plan is not safe, because the operation $\Pi_{\texttt{Affiliation}}$ (projection-with-duplicate elimination) combines tuples that are not independent, and therefore the output probabilities are incorrect. The figure illustrates this for the output value `Y! Research`, by tracing its computation through the query plan: the output probability is $1 - (1 - p_3^1 q_1)(1 - p_3^1 q_2)$. However, the lineage of `Y! Research` is $(X_1 = 3 \wedge Y_1 = 1) \vee (X_1 = 3 \wedge Y_2 = 1)$, hence the correct probability is $p_3^1(1 - (1 - q_1)(1 - q_2))$.

Alternatively, consider the plan shown in (c). This plan performs an early projection and duplicate elimination on `Services`. It is logically equivalent to the plan in (b), i.e., the extra duplicate elimination is harmless. However, the new plan computes the output probability correctly: the figure illustrates this for the same output value, `Y! Research`. Note that although plans (b) and (c) are logically equivalent over conventional databases, they are no longer equivalent in their treatment of probabilities: one is safe, the other not.

Safety is a central concept in query processing on probabilistic databases. A query optimizer needs to search not just for a plan with lowest cost, but for one that is safe as well, and this may affect the search strategy and the outcome: in a conventional database there is no reason to favor the plan in (c) over that in (b) (and in fact modern optimizers will not choose plan (c) because the extra duplication elimina-

Location	
U. Washington	$(X_1 = 1) \wedge (Y_1 = 1) \vee (X_1 = 1) \wedge (Y_2 = 1) \vee (X_3 = 2) \wedge (Y_4 = 1)$
U. Wisconsin	$(X_1 = 2) \wedge (Y_1 = 1) \vee (X_1 = 2) \wedge (Y_2 = 1) \vee (X_3 = 1) \wedge (X_4 = 1)$
Y! Research	$(X_1 = 3) \wedge (Y_1 = 1) \vee (X_1 = 3) \wedge (Y_2 = 1)$

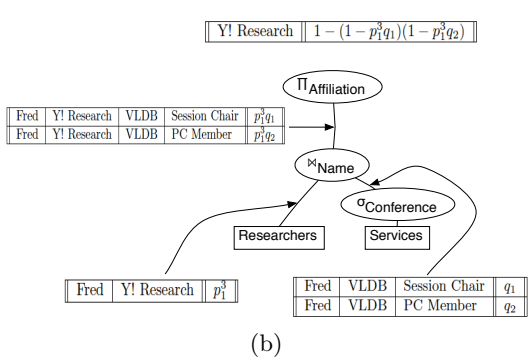
Figure 2: An example of a c-table.

```
SELECT x.Affiliation, confidence( )
FROM Researchers x, Services y
WHERE x.Name = y.Name
 and  y.Conference = 'VLDB'
GROUP BY x.Affiliation
```

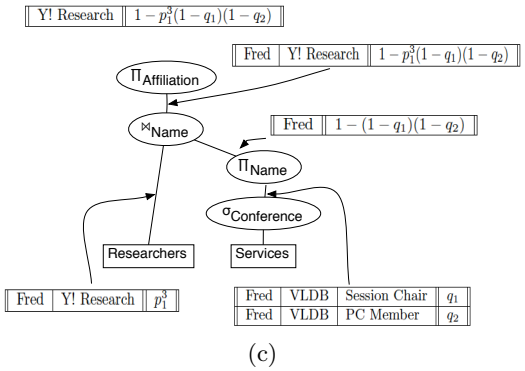
(a)

```
SELECT x.Affiliation, 1-prod(1-x.P*y.P)
FROM Researchers x, (SELECT Name, 1-(1-prod(P))
                     FROM Services
                     WHERE Conference = 'VLDB'
                     GROUP BY Name) y
WHERE x.Name = y.Name
GROUP BY x.Affiliation
```

(d)



(b)



(c)

Figure 3: A SQL query on the data in Figure 1(a) returning the affiliations of all researchers who performed some service for VLDB. The query follows the syntax of MayBMS, where `confidence()` is an aggregate operator returning the output probability. The figure shows an unsafe plan in (b) and a safe plan in (c), and also traces the computation of the output probability of `Y! Research`: it assumes there is a single researcher `Fred` with that affiliation, and that `Fred` performed two services for VLDB. The safe plan re-written in SQL is shown in (d): the aggregate function `prod` is not supported by most relational engines, and needs to be rewritten in terms of `sum`, logarithms, and exponentiation.

tion increases the cost), but in a probabilistic database plan (c) is safe while (b) is unsafe. A safe plan can be executed directly by a database engine with only small changes to the implementation of its relational operators. Alternatively, a safe plan can be executed by expressing it in regular SQL and executing it on a conventional database engine, without any changes: Figure 3(d) illustrates how the safe plan can be converted back into SQL.

Safe plans have been described for databases with independent tuples [11], for BID databases [12], for queries whose predicates have aggregate operators [32], and for Markov Chain databases [34]. While conceptually a safe plan ties the probabilistic inference to the query plan, Olteanu et al. [29] have shown that is it possible to separate them at runtime: the optimizer is free to choose any query plan (not necessarily safe), then the probabilistic inference is guided by the information collected from the safe plan. This results in significant execution speedup for typical SQL queries.

2.2.2 Dichotomy of Query Evaluation

Unfortunately, not all queries admit safe plans. In general, query evaluation on a probabilistic database is no easier than general probabilistic inference. The latter is known to be a hard problem [35]. In databases, however, one can approach the query evaluation problem differently, in a way that is best explained by recalling an important distinction made by Vardi in a landmark paper in 1982 [38]. He proposed that the query expression (which is small) and the database (which is large) be treated as two different inputs to the query evaluation problem, leading to three different complexity measures: the data complexity (when the query is fixed), the expression complexity (when the database is fixed), and the combined complexity (when both are part of the input). For example, in conventional databases, all queries have data complexity in PTIME, while the combined complexity is PSPACE complete.

We apply the same distinction to query evaluation on probabilistic databases. Here the data complexity offers a more striking picture: some queries are in PTIME (e.g., all safe queries), while others have #P-hard data complexity. In fact, for certain query languages or under certain assumptions it is possible to prove a complete dichotomy, i.e. each query belongs to one of these two classes [10,12,32,34]. Figure 4 describes the simplest dichotomy theorem, for con-

junctive queries without self-joins over databases with independent tuples, first proven in [11]. Safe queries are by definition in the first class; under the dichotomy property, any unsafe query has #P-hard data complexity. For unsafe queries we really have no choice but to resort to a probabilistic inference algorithm that solves, or approximates a #P-hard problem. The abrupt change in complexity from PTIME to #P-hard is unique to probabilistic databases, and it means that query optimizers need to make special efforts to identify and use safe queries.

An active line of research develops query evaluation techniques that soften the transition from safe to unsafe queries. One approach extends the reach of safe plans: for example safe sub-plans can be used to speed up processing unsafe queries [33], functional dependencies on the data, or knowing that some relations are deterministic can be used to find more safe plans [11, 29], and safe plans have been described for query languages for streams of events [34].

Another approach is to optimize the general-purpose probabilistic inference on the lineage expressions [36]. A new direction is taken by a recent project at IBM Almaden [23], which builds a database system where Monte Carlo simulations are pushed deep inside the engine, thus being able to evaluate any query, safe or unsafe. What is particularly promising about this approach is that through clever query optimization techniques, such as *tuple bundles*, the cost of sampling operations can be drastically reduced. A complementary approach, explored by Olteanu et al. [29] is to rewrite queries into ordered binary decision diagrams (OBDD). They have observed that safe plans lead to linear-sized OBDD's. This raises the possibility that other tractable cases of OBDDs can be inferred, perhaps by analyzing both the query expression and the database statistics.

2.2.3 Materialized Views

The use of materialized views to answer queries is a very powerful tool in data management [21]. In its most simple formulation, there are a number of materialized views, e.g., answers to previous queries, and the query is rewritten in terms of these views, to improve performance.

In the case of probabilistic databases, materialized views have been studied in [33]. Because of the dichotomy of the query complexity, materialized views can have a dramatic impact on query evaluation: a query may be unsafe, hence #P-hard, but after rewriting it in terms of views it may become a safe query, and thus is in PTIME. There is no magic here, we don't avoid the #P-hard problem, we simply take advantage of the fact that the main cost has already been paid when the view was materialized.

The major challenge in using materialized views over probabilistic data is that we need to represent the view's output. We can always compute the lineage of all the tuples in the view, and this provides a complete representation of the view, but it also defeats our purpose, since using these lineage expressions during query evaluation does not simplify the probabilistic inference problem. Instead, we would like to use only the marginal tuple probabilities that have been computed for the materialized view, not their lineage. For example, it may happen that all tuples are independent probabilistic events, and in this case we only need the marginal probabilities; we say in this case the view is fully representable. In general, not all tuples in the view are independent, but it is always possible to partition the tuples into blocks such that tuples from different blocks are independent, and, moreover, there exists a "best" such partition [33]; within a block, the correlations between the tuples remain unspecified. The blocks are described at the schema level, by specific a set of attributes: grouping by those attributes gives the blocks. This is called a *partial representation*, and can be used to evaluate some queries over the views. Note that the problem of finding a good partial representation of the view is done by a static analysis that is orthogonal to the analysis whether the view is safe or unsafe: there are examples for all four combinations of safe/unsafe representable/unrepresentable views.

2.3 Facet 3: User Interface

The semantics of query Q on a probabilistic database with possible worlds W is, in theory, quite simple, and is given by the *image probability space* over the set of possible answers, $\{Q(I) \mid I \in W\}$. In practice, it is impossible, and perhaps useless, to return all possible *sets* of answers. An important problem in probabilistic databases is how to best present the set of possible query answers to the user. To date, two practical approaches have been considered: ranking tuples, and aggregation over imprecise values.

2.3.1 Ranking and Top-k Query Answering

In this approach the system returns all possible answer tuples and their probabilities: $\mathbf{P}(t_1 \in Q)$, $\mathbf{P}(t_2 \in Q)$, ... in Sec. 2.1.1; the correlations between the tuples are thus lost. The emphasis in this approach is to rank these tuples, and restrict them to the top k.

One way to rank tuples is in decreasing order of their output probabilities [31]: $\mathbf{P}(t_1 \in Q) \geq \mathbf{P}(t_2 \in Q) \geq \ldots$. Often, however, there may be a user-specified order criteria, and then the system needs to combine the user's ranking scores with the output probability [37]. A separate question is whether we can use ranking to our advantage to speed up query performance by returning only the k highest ranked tuples: this problem is called *top-k query answering*. One can go a step further and drop the output probabilities altogether: Ré et al. [31] argue that ranking the output tuples is the only meaningful semantics for the user, and proposes to focus the query processor on computing the ranking, instead of the output probabilities.

The power of top-k query answering in speeding up query processing has been illustrated in a seminal paper by Fagin, Lotem, and Naor [16]. When applied to probabilistic databases that principle leads to a technique called *multisimulation* [31]. It assumes that a tuple's probability $\mathbf{P}(t \in Q)$ is approximated by an iterative algorithm, like a Monte Carlo simulation: after some number steps n, the probability $\mathbf{P}(t \in Q)$ is known to be, with high probability, in an interval $(p-\varepsilon_n, p+\varepsilon_n)$, where ε_n decreases with n. The idea in the multisimulation algorithm is to control carefully how to allocate the simulation steps among all candidate tuples in the query's answer, in order to identify the top k tuples without wasting iterations on the other tuples. Multisimulation reduces the computation effort roughly by a factor of N/k, where N is the number of all possible answers, and k is the number of top tuples returned to the user.

2.3.2 Aggregates over Imprecise Data

In SQL, aggregates come in two forms: value aggregates, as in *for each company return the* `sum` *of the profits in all its*

Hierarchical Queries

In the case of tuple-independent databases (where all tuples are independent) safe queries are precisely the *hierarchical queries*; we define hierarchical queries here.
A *conjunctive query* is:

$$q(\bar{z}) \quad :- \quad \texttt{body}$$

where `body` consists of a set of subgoals $g_1, g_2, \ldots, g_k$, and $\bar{z}$ are called the head variables. Denote $Vars(g_i)$ the set of variables occurring in g_i and $Vars(q) = \cup_{i=1,k} Vars(g_i)$. For each $x \in Vars(q)$ denote $sg(x) = \{g_i \mid x \in Vars(g_i)\}$.

DEFINITION 2.1. *Let q be a conjunctive query and $\bar{z}$ its head variables. q is called* hierarchical *if for all $x, y \in Vars(q) - \bar{z}$, one of the following holds: (a) $sg(x) \subseteq sg(y)$, or (b) $sg(x) \supseteq sg(y)$, or (c) $sg(x) \cap sg(y) = \emptyset$.*

A conjunctive query is without self-joins if any two distinct subgoals refer to distinct relation symbols.

THEOREM 2.2 (DICHOTOMY). *[11, 12] Let q be a conjunctive query without self-joins. (1) If q is hierarchical then its data complexity over tuple-independent databases is in PTIME. (2) If q is not hierarchical then its data complexity over tuple-independent databases is #P-hard.*

To illustrate the theorem, consider the two queries:

$$q_1(z) \quad :- \quad R(x,z), S(x,y), T(x,z)$$
$$q_2(z) \quad :- \quad R(x,z), S(x,y), T(y,z)$$

In q_1 we have $sg(x) = \{R, S, T\}, sg(y) = \{S\}$; hence it is hierarchical and can be evaluated in PTIME.
In q_2 we have $sg(x) = \{R, S\}, sg(y) = \{S, T\}$; hence it is non-hierarchical and is #P-hard.

Figure 4: The dichotomy of conjunctive queries without selfjoins on tuple-independent probabilistic databases is captured by *Hierarchical Queries*.

units, and predicate aggregates, as in *return those companies having the* `sum` *of profits greater than 1M*. Both types of aggregates are needed in probabilistic databases. The first type is interpreted as expected value, and most aggregate functions can be computed easily using the linearity of expectation. For instance, the complexities of computing `sum` and `count` aggregates over a column are same as the complexities of answering the same query without the aggregate, i.e., where all possible values of the column are returned along with their probabilities [11]. Complexities of computing `min` and `max` are same as those of computing the underlying queries with the aggregates replaced by projections removing the columns [11]. One aggregate whose expected value is more difficult to compute is `average`, which is an important aggregate function for OLAP over imprecise data. One can compute the expected values of `sum` and `count(*)`, but the expected value of `average` is not their ratio. A surprising result was shown by Jayram, Kale, and Vee [24] who proved that average can be computed efficiently. They give an exact algorithm to compute average on a single table in time $O(n \log^2 n)$. They also give efficient algorithms to compute various aggregates when the data is streaming.

The second type of aggregates, those occurring in the `HAVING` clause of a SQL query, have also been considered [32]. In this case, one needs to compute the entire density function of the random variable represented by the aggregate, and this is more difficult than computing the expected value. Similar to safe queries, the density function can sometimes be computed efficiently and exactly, but it is hard in general. Worse, in contrast to safe queries, which can always be efficiently approximated, there exists HAVING queries that do not admit efficient approximations.

3. A LITTLE HISTORY OF THE (POSSIBLE) WORLDS

There is a rich literature on probabilistic databases, and we do not aim here to be complete; rather, as in Gombrich's classic *A Little History of the World*, we aim to "catch a glimpse". Early extensions of databases with probabilities date back to Wong [40] and Cavallo and Pittarelli [6]. In an influential paper Barbara et al. [3] described a probabilistic data model that is quite close to the BID data model, and showed that SQL queries without duplicate elimination or other aggregations can be evaluated efficiently. ProbView [27] removed the restriction on queries, but returned confidence intervals instead of probabilities. At about the same time, Fuhr and Roelleke [18] started to use c-tables and lineage for probabilistic databases and showed that every query can be computed this way.

Probabilities in databases have also been studied in the context of "reliability of queries", which quantifies the probability of a query being correct assuming that tuples in the database have some probability of being wrong. Grädel, Gurevich and Hirsch [19] were the first to prove that a simple query can have data complexity that is #P-hard.

Andritsos, Fuxman and Miller [1] have applied probabilistic databases to the problem of consistent query answering over inconsistent databases. They observed that the "certain tuples" [21] to a query over an inconsistent databases are precisely the tuples with probability 1 under probabilistic semantics.

The intense interest in probabilistic databases seen today is due to a number of influential projects: applications to sensor data [7, 15], data cleaning [1], and information extraction [20], the safe plans of [11], the Trio system [4] that

introduced ULDBs, and the advanced representation systems described in [2, 36].

4. CONCLUSIONS

Many applications benefit from finding valuable facts in imprecise data, the *diamonds in the dirt*, without having to clean the data first. The goal of probabilistic databases is to make uncertainty a first class citizen, and to reduce the cost of using such data, or (more likely) to enable applications that were otherwise prohibitively expensive. This paper described some of the recent advances for large scale query processing on probabilistic databases and their roots in classical data management concepts.

Acknowledgments We thank the anonymous reviewers for their helpful comments, and Abhay Jha for discussions on the paper.

5. REFERENCES

[1] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. In *ICDE*, 2006.

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

[3] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Trans. Knowl. Data Eng.*, 4(5):487–502, 1992.

[4] O. Benjelloun, A. D. Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDBJ*, 17(2):243–264, 2008.

[5] D. Burdick, P. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. Efficient allocation algorithms for olap over imprecise data. In *VLDB*, pages 391–402, 2006.

[6] R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In *Proceedings of VLDB*, pages 71–81, 1987.

[7] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, pages 551–562, 2003.

[8] E. F. Codd. Relational completeness of data base sublanguages. In *Database Systems*, pages 65–98. Prentice-Hall, 1972.

[9] R. Cowell, P. Dawid, S. Lauritzen, and D. Spiegelhalter, editors. *Probabilistic Networks and Expert Systems*. Springer, 1999.

[10] N. Dalvi and D. Suciu. The dichotomy of conjunctive queries on probabilistic structures. In *PODS*, pages 293–302, 2007.

[11] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.

[12] N. Dalvi and D. Suciu. Management of probabilistic data: Foundations and challenges. In *PODS*, pages 1–12, Beijing, China, 2007. (invited talk).

[13] A. Darwiche. A differential approach to inference in bayesian networks. *Journal of the ACM*, 50(3):280–305, 2003.

[14] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan. Dblife: A community information management platform for the database research community. In *CIDR*, pages 169–172, 2007.

[15] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, pages 588–599, 2004.

[16] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–113. ACM Press, 2001.

[17] N. Friedman, L. .Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *IJCAI*, pages 1300–1309, 1999.

[18] N. Fuhr and T. Roelleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[19] E. Grädel, Y. Gurevich, and C. Hirsch. The complexity of query reliability. In *PODS*, pages 227–234, 1998.

[20] R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, pages 965–976, 2006.

[21] A. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[22] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31:761–791, October 1984.

[23] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. Haas. MCDB: a Monte Carlo approach to managing uncertain data. In *SIGMOD*, pages 687–700, 2008.

[24] T. Jayram, S. Kale, and E. Vee. Efficient aggregation algorithms for probabilistic data. In *SODA*, 2007.

[25] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *ICDE*, pages 1160–1169, 2008.

[26] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 2001.

[27] L. Lakshmanan, N. Leone, R. Ross, and V. Subrahmanian. Probview: A flexible probabilistic database system. *ACM Trans. Database Syst.*, 22(3), 1997.

[28] A. Nierman and H. Jagadish. ProTDB: Probabilistic data in XML. In *VLDB*, pages 646–657, 2002.

[29] D. Olteanu, J. Huang, and C. Koch. SPROUT: Lazy vs. eager query plans for tuple independent probabilistic databases. In *ICDE*, 2009.

[30] V. Rastogi, D. Suciu, and S. Hong. The boundary between privacy and utility in data publishing. In *VLDB*, 2007.

[31] C. Ré, N. Dalvi, and D. Suciu. Efficient Top-k query evaluation on probabilistic data. In *ICDE*, 2007.

[32] C. Ré and D.Suciu. Efficient evaluation of having queries on a probabilistic database. In *Proceedings of DBPL*, 2007.

[33] C. Ré and D.Suciu. Materialized views in probabilistic databases for information exchange and query optimization. In *Proceedings of VLDB*, 2007.

[34] C. Ré, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD*, Vancouver, Canada, 2008.

[35] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.

[36] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.

[37] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Probabilistic top- and ranking-aggregate queries. *ACM Trans. Database Syst.*, 33(3), 2008.

[38] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of 14th ACM SIGACT Symposium on the Theory of Computing*, pages 137–146, San Francisco, California, 1982.

[39] T. Verma and J. Pearl. Causal networks: Semantics and expressiveness. *Uncertainty in Artificial Intelligence*, 4:69–76, 1990.

[40] E. Wong. A statistical approach to incomplete information in database systems. *ACM Trans. Database Syst.*, 7(3):470–488, 1982.