MAYBMS: A SYSTEM FOR MANAGING LARGE AMOUNTS OF UNCERTAIN DATA

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Lyublena R. Antova

February 2010

MAYBMS: A SYSTEM FOR MANAGING LARGE AMOUNTS OF UNCERTAIN DATA

Lyublena R. Antova, Ph.D.

Cornell University 2010

This dissertation presents the foundations for building a scalable database management system for managing uncertain data, as it appears in different data management scenarios such as data integration, data cleaning, scientific data and web data management. The result of this work is MayBMS - a scalable open-source database management system for managing large amounts of uncertain data. MayBMS uses the so-called U-relational databases to represent uncertainty. U-relational databases store uncertainty and correlations in a purely relational way, and are a complete representation system for finite world sets. Other benefits achieved by our representation model include compact storage and efficient query evaluation. The results of our experimental evaluation clearly show that query evaluation in MayBMS scales up to large data sizes and uncertainty ratios, and that MayBMS consistently outperforms other current systems for managing uncertain data. The dissertation also discusses optimization of queries on vertically partitioned data, efficient confidence computation algorithms, and challenges and solutions when designing an application programming interface for uncertain databases.

BIOGRAPHICAL SKETCH

I obtained my Bachelor degree in Computer Science at Sofia University, Bulgaria in 2003. After spending one year as exchange student at the Catholic University Eichstaett-Ingolstadt in Germany, I joined the Master program at Saarland University, Germany, supported by a scholarship by the Max Planck Institute for Computer Science. I obtained a Master degree in 2006 and moved on the PhD program there, again supported by Max Planck. In 2007 I joined Cornell where I continued working with Professor Christoph Koch towards my dissertation.

To my father Rosen who passed away just before this dissertation was ready.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help and support of a number of people. My advisor Christoph Koch was the one who made me discover my passion for databases and inspired me to work in the field. Throughout the years he has provided valuable support for my research, making sure I do not get lost along the way and always keep my focus on producing important and valuable results. Christoph was extremely approachable and did not hesitate to find time for meetings at the most unusual times and days whenever I needed help. Dan Olteanu, with whom I was working in the early days of my PhD was another highly motivated and inspiring person who has helped me with valuable advice throughout my research.

I am grateful to Alon Halevy who hosted me twice as a summer intern at Google and gave me the invaluable option to work with large scale systems and technologies in a unique environment.

Last but not least I would like to thank my family and friends who were there for me in good and bad.

TABLE OF CONTENTS

LIST OF FIGURES

Part I

Prolog

CHAPTER 1

INTRODUCTION

Uncertain data is ubiquitous in many real life application. Scientific data, web data extraction, data cleaning and warehousing, census and other forms of surveys are just a few examples of scenarios where data volumes are high and uncertainty is the rule rather than an exception. Despite this fact, until recently researchers and companies in these areas had one of the two choices. Either they rely on the extremely limited support for uncertainty in existing database management systems (which typically does not go beyond null values built in the DBMS), or had to build their own custom solutions for representing and managing uncertainty. This includes designing a custom format for representing uncertainty and implementing their own query language and access methods which are often ad-hoc with no clean semantics, and in addition were not transferable between companies or organizations. This very much resembles the situation of 40 years ago before the appearance of the first database management systems. In those years data was stored in custom file formats and processed using propriety programs. Then the relational databases of Codd came along and saved the day. They included well defined model of representing data, which allowed abstracting the logical level of representing data from the physical storage details, and made possible to specify queries in a declarative query language that allowed developers to define the result they want to obtain from the database and leave the execution and access path selection to the query optimizer. Benefits in terms of code reuse and query optimization were tremendous.

The goal of this dissertation is to provide the foundations for a general-

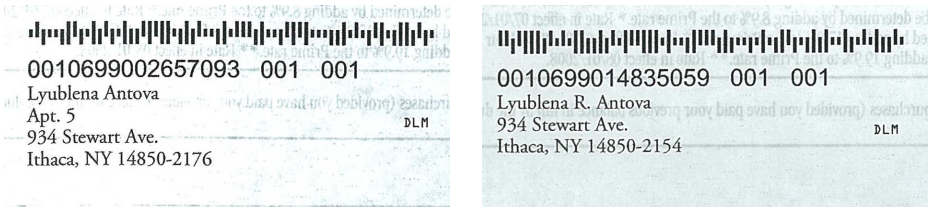


Figure 1.1: Credit card offers mailed to the same person.

purpose uncertainty database management system that will allow users to store data containing uncertainty independent of its domain, much like relational databases allowed for storing and processing structured data in a uniform way. This is a multi-faceted problem requiring solutions for different subproblems including:

- Representation model
- Well defined query language and application programming interface
- Efficient query evaluation and optimization techniques

This dissertation touches on all of these problems and provides in-depth solutions for all of them.

Before we go into the particular details of the techniques proposed in this work, let us look at some examples that motivate the need to manage uncertainty in practice.

Example 1.1. Consider Figure 1.1. It shows parts of two scanned letters offering credit card deals from the same bank, which the author of this dissertation received on the very same day. Except for the address box, the two letters were identical. What seems to have happened is that the credit card company received information about this person from two different sources that disagreed on the exact name of the person, details of the address line and the precise zip

Cust	FName	MI	LName	AddrId
	Lyublena	{R, NULL}	Antova	a1

Addr	Id	Street	Apt	City	State	Zip	Add_on
	a1	934 Stewart Ave.	{5,NULL}	Ithaca	NY	14850	{2176, 2154}

Figure 1.2: Or-set relations for the address entries of Figure 1.1

code. Because of the dirty data, the same message was mailed twice to the same receiver. The potential customers information can be cleaned and stored in a probabilistic database. For example, using or-set notation we can represent the deduplicated data of Figure 1.1 as the database of Figure 1.2. The or-sets contain the possible values for the respective fields, and are assumed to be independent of each other. For example the customer's middle initial is reported to be R, but can be missing as well, and the zip code add-on is either 2176 or 2154. Even though the data is uncertain, the credit company can still query the data and even get certain results in some cases. For example the query

```
select FName, LName, Street, City, Zip
from Cust c, Addr a
where c.AddId = Id;
```

will return the same result in each possible world, independent of what the values for MI, Apt and Add_on are. Alternatively, a query can be written which asks for the most likely address line for each person. □

Example 1.2. Figure 1.3 demonstrates some challenges that arise when extracting information from the web. The Figure shows search results for used cars obtained on the website thecarconnection.com. Notice that the first two results are very likely to refer to the same car. They come from different sources and

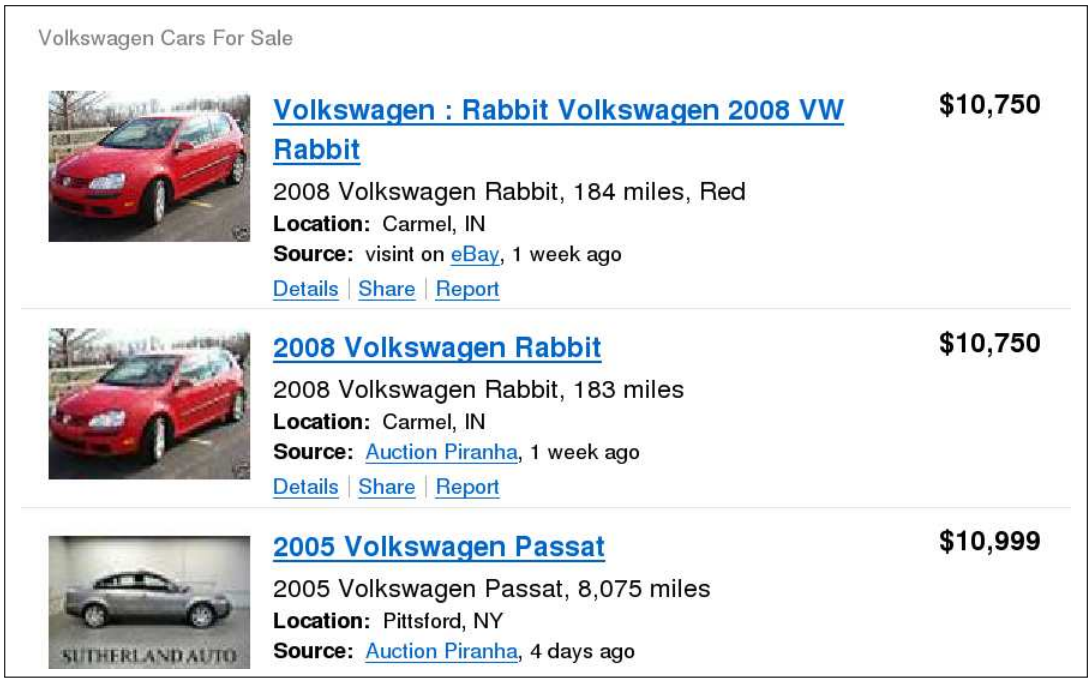


Figure 1.3: Search results for used cars

differ in the car mileage and color properties, but apart from that the entries are identical. A duplicate identification algorithm that uses similarity measures and other technique can detect that the entities are the same, and an uncertain database management system can be used to store the possible values and provide querying capabilities until the true values are resolved. □

Example 1.3. Figure 1.4 shows a protein-protein interaction network. Each node in the network is a protein, and a link between two nodes indicates the two proteins interact. Typically interactions are not certain but are quantified with a probability, and there can be constraints such as two proteins interact only when a third one is present. This network can be seen as representing a set of possible instances, where in each one of them all interactions are resolved and two proteins either interact with 100% certainty or do not interact at all. In such scenarios researchers typically want to find and predict patterns of interactions,

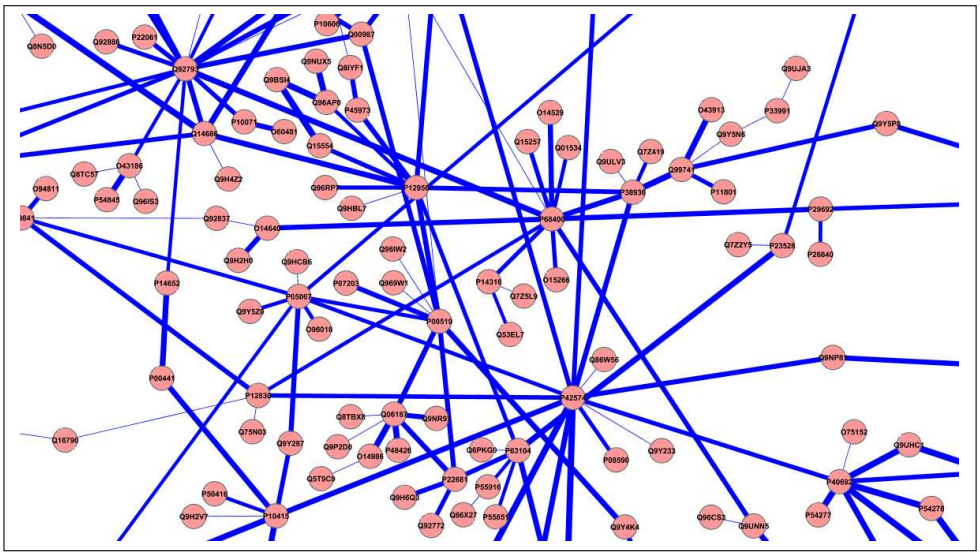


Figure 1.4: Protein-protein interaction network

such as what is the probability that two proteins interact when certain conditions hold or what are the most likely groups of proteins that interact at a given point in time. Using probabilistic database techniques, we can answer those questions by evaluating the respective pattern query in each possible world, and then aggregating over all worlds to compute the likelihood of the desired pattern. □

Scenarios like the ones above require the support of uncertainty in databases. A widely accepted approach to managing in certainty is designing a representation system for storing sets of possible worlds, where one of them is the true world but it is not clear which one. In addition, some systems support defining probability distributions on the world-set which quantifies the likelihood that a world is the true one. For a representation system to be useful in practice the following properties are desirable:

- **Compactness**. It should be able to store compactly large sets of possible worlds.
- **Expressiveness**. It should allow for expressing different correlations that exist in the data or arise as result of querying. In particular, the representation system should be closed under querying: query results should be representable in the same formalism for at least standard query languages such as relational algebra.
- **Efficient query evaluation**. It should support efficient techniques for evaluating queries directly on the representation. Typically, there is a conflict between the succinctness and expressiveness of the representation system, and the efficiency of evaluating interesting queries on top of it.
- **Ease of use**. The representation system should be easy to implement and deploy by developers and researchers.

This work introduces U-relational databases, a space efficient formalism for managing uncertain information. We will introduce U-relations with the following example.

Example 1.4. Let us consider a modification of the example of Figure 1.1. Suppose the credit card company stores information about three individuals, Lilly An, Mimi Pan, and Ron Wang where again some details about the three persons are conflicting. The following information is known:

- It is known that Lilly's middle initial is either 'R' or non-existent.
- Mimi's middle initial is either 'D' or 'O' (perhaps due to errors in the OCR process).
- Ron's middle initial is either 'P' or 'Q'.

U_1	T_R	FN	LN
	t_1	Lilly	An
	t_2	Mimi	Pan
	t_3	Ron	Wang

U_2	T_R	Street	City	State	D
	t_1	Stewart	Ithaca	NY	
	t_2	Stewart	Ithaca	NY	
	t_3	Aurora	Ithaca	NY	$w \mapsto 1$
	t_3	El Camino	Mountain View	CA	$w \mapsto 2$

U_3	T_R	MI	D
	t_1	R	$x \mapsto 1$
	t_1	NULL	$x \mapsto 2$
	t_2	D	$y \mapsto 1$
	t_2	O	$y \mapsto 2$
	t_3	P	$v \mapsto 1$
	t_3	Q	$v \mapsto 2$

U_4	T_R	Apt	D
	t_1	5	$z \mapsto 1$
	t_1	6	$z \mapsto 2$
	t_2	5	$z \mapsto 1$
	t_2	6	$z \mapsto 2$
	t_3	10	

U_5	T_R	Zip	D
	t_1	14850-2176	$z \mapsto 1$
	t_1	14850-2154	$z \mapsto 2$
	t_2	14850-2176	$z \mapsto 1$
	t_2	14850-2154	$z \mapsto 2$
	t_3	14850-1234	$w \mapsto 1$
	t_3	94040-0178	$w \mapsto 2$

W	Var	Rng	Pr
	x	1	0.9
	x	2	0.1
	y	1	0.5
	y	2	0.5
	z	1	0.6
	z	2	0.4
	v	1	0.8
	v	2	0.2
	w	1	0.3
	w	2	0.7

Figure 1.5: Representing deduplicated customer data using U-relational databases.

- Ron lives in either Mountain View, CA, or Ithaca, NY.
- The apartment number for both Lilly and Mimi is 5 or 6.
- Lilly and Mimi are roommates, so they share the same address line.
- There is a functional dependency $\{Street, Apt, City, State\} \rightarrow Zip$.

Suppose the database schema contains a single relation R(FN, MI, LN, Street, Apt, City, State, Zip). We model this scenario using the U-relational database of Figure 1.5. We use vertical partitioning (cf. e.g. [13, 64]) to achieve attribute-level uncertainty, and have partitioned R's attributes into five partitions. Partitions U_2, U_3, U_4 and U_5 contain the attributes that have multiple possible values. To encode uncertainty we use five variables x, y, z, v, w, each of which takes

R	FN	MI	LN	Street	Apt	City	State	Zip
t_1	Lilly	R	An	Stewart	5	Ithaca	NY	14850-2176
t_2	Mimi	D	Pan	Stewart	5	Ithaca	NY	14850-2176
t_3	Ron	Q	Wang	El Camino	5	Mountain View	CA	94040-0178

Figure 1.6: Possible world for the valuation $\theta = \{x \mapsto 1, y \mapsto 2, z \mapsto 1, v \mapsto 2, w \mapsto 2\}$ in the U-relational database of Figure 1.5.

two possible values independently of the other four. The domain values for each variable and the corresponding probability distribution are given in the so called world-table W in Figure 1.5. All U-relations except for U_1 have an additional condition column D, the so-called *world-set descriptor* which encodes in which worlds the tuples exists in the database. To obtain a possible world, we choose a valuation θ mapping each variable to a value from its domain, and select the tuples from the U-relations whose world-set descriptors are consistent with θ. For example for the valuation $\theta = \{x \mapsto 1, y \mapsto 2, z \mapsto 1, v \mapsto 2, w \mapsto 2\}$ we obtain the possible world in Figure 1.6. The probability of the possible world is $0.9 \cdot 0.5 \cdot 0.6 \cdot 0.2 \cdot 0.7 = 0.0378$, corresponding to the product of the probabilities of the chosen variable values. Note that while the variables are independent, we can correlate values of different fields, even across tuples and relations by reusing the same variable in different world-set descriptors. For example, we have used the same variable z to represent the fact that Lilly and Mimi live together, and their apartment numbers must match, and that the address determines the zip code. As we shall see later in Chapter 4, we can build world-set descriptors of multiple (var, rng) pairs, which is crucial for the succinctness of the formalism. □

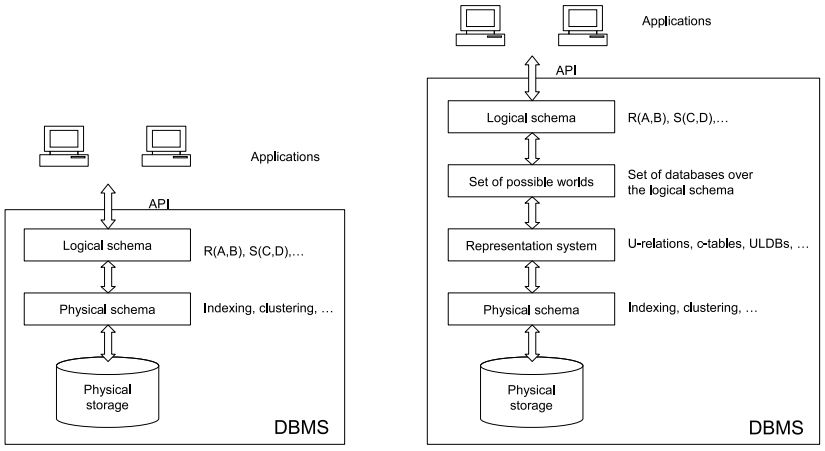
An important strength of U-relational databases is the use of vertical partitioning. Vertical partitioning is shown to bring performance advantages in managing data in different domains such as data warehousing applications [64],

RDF graph data [1], etc. Some of the benefits include opportunities for compressing representation, and optimized query performance due to increased read throughput and late materialization.

Most of these advantages carry over to probabilistic databases as well. In addition, there are new opportunities for storage compression by exploiting independence between attribute values. As a simple example, consider a relation with one tuple over m attributes where each attribute has 2 possible values, and attributes are independent of each other (that is, an or-set relation). A tuple-level representation will require storing 2^m possible tuples, while by vertically decomposing the relation one can represent the same set of possible worlds using only $O(2 * m)$ space.

An important component when designing systems for managing uncertain data is the ability to write database application programs that access and update data through an application programming interface (API). Most current systems are either limited to providing only batch execution of SQL-like queries on the uncertain database with no user interaction, or have a representation-dependent programming model that requires knowledge of system-specific implementation details.

Traditional database management systems abstract away the physical details of how the data is stored. External applications interact with the database by formulating queries and updates on the *logical schema*, which are then translated into operations on the physical storage structures on disk. Figure 1.7 shows an architecture of a traditional DBMS, and one of a DBMS for uncertain data. Compared to traditional DBMSs, systems for managing uncertain data need to deal with two additional levels of abstraction, the representation system and the



(a) Traditional DBMS (b) DBMS for uncertain data

Figure 1.7: Levels of abstraction provided by a DBMS.

possible worlds model.

To demonstrate some of the challenges of designing an API for uncertain databases, consider the following example (essentially from [14]). Figure 1.8 shows an example of a police database containing reports on stolen cars. Due to conflicting or missing information, several instances are possible, as shown in the figure. An application that manages such data should provide users with the ability to update or insert new evidence regarding observed objects, execute queries, or apply expert knowledge to resolve inconsistency.

The program in Figure 1.9 allows to enter new evidence about a stolen car. In case the car already exists in the database, and the information about it matches the user's input, the program increments the number of witnesses; otherwise a new entry is inserted into the database.

The program makes sense in the case of certain databases and is straightforward to execute. What is different in the presence of uncertainty? Consider the

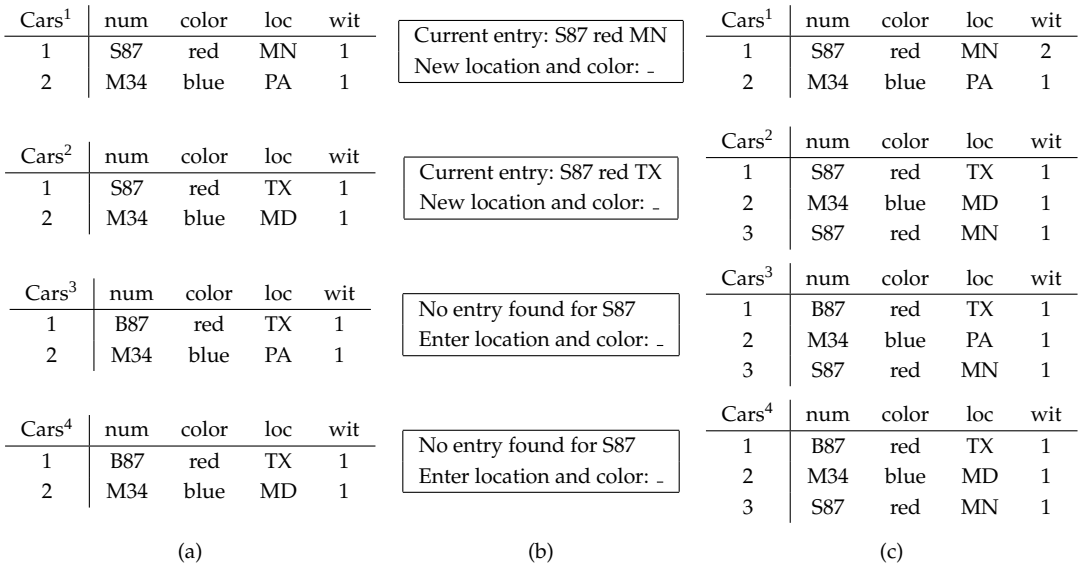


Figure 1.8: Executing programs on uncertain databases: (a) set of possible worlds reporting information on stolen cars; (b) output of the program in each of the worlds of (a); (c) result of running the program on the world-set of (a).

world-set in Figure 1.8 (a). The four worlds differ in the license plate number specified for car 1, and the reported location for cars 1 and 2. If the user enters S87 for the license plate number, in worlds 1 and 2 the car already exists in the database, so its entry will be shown to the user. In worlds 3 and 4 the car is not found and the user will receive a message reporting that. As shown in Figure 1.8 (b), there are three different outputs for the four worlds of Figure 1.8 (a), and the meaning of the expected input is also different. Figure 1.8 (c) shows the outcome of the program on each of the worlds if the user specifies S57, MN, red as input in each of the worlds. In world 1 the witness count for S57 is increased to 2, and in the remaining worlds a new tuple is added to the database.

Clearly, we cannot expect users to supply input for each world individually, or to deal with different output produced in each of the worlds. Such programs are not only unintuitive to the user: they are also infeasible to implement as in

```
read("Enter license plate:", $x);
if (exists select * from cars where num=$x){
// modify existing entry
    for($t in select * from cars where num=$x){
      write("Current entry: $t");
      read("New location and color:", $loc,$color);
      if (exists select * from cars where num=$x
          and loc=$loc and color=$color)
        update cars set wit=wit+1
        where num=$x and loc=$loc and color=$color;
      else
        insert into cars values($x,$loc,$color,1);
    }
}
else { // entry does not exist
    write("No entry found for $x");
    read("Enter location and color:", $loc, $color);
    insert into cars values($x,$loc,$color,1);
}
```

Figure 1.9: Program for inserting new evidence into a stolen cars database.

practice the number of possible worlds can be prohibitively large. One solution to this problem is to introduce an intermediate level between the database and the user that verifies that the messages (output and input) that are passed between the database and the user are the same *in all worlds*, and collapses those into one. This has the advantage of hiding the uncertain nature of the data from the users, allowing them to work with the database as if it were complete. We say that in this case the program is observationally deterministic. A second approach is to only allow programs that are *guaranteed* to have the same behavior on all worlds, and to verify observational determinism in a *static manner*. For example, the above program can be modified to first request all evidence information, and then carry out the update in each of the worlds by either updating existing tuples or inserting new ones, without disclosing this different behavior to the user.

The second major challenge in API development for uncertain databases is to map programs that are conceptually executed on each world individually into programs on the representation. In this work we show how to do this efficiently for queries on the representation model we use, U-relational databases. Programs are more complex as they provide richer structure such as updates, looping and branching constructs and the flow of execution can be different in each world. We take the approach of first pushing as much of the program code as possible into (set-at-a-time) queries and updates, which we then map to queries and updates on representations.

1.1 Contributions and outline

This thesis defines and studies U-relational databases, an efficient formalism for managing probabilistic data. U-relations have the following properties:

- **Expressiveness**: U-relations are *complete* for finite sets of possible worlds, that is, they allow for the representation of any finite world-set. This also means that results of any relational algebra query on top of a U-relational database can be represented in U-relational form.
- **Succinctness**: U-relations represent uncertainty on the attribute level. Even though they allow for more efficient query evaluation, U-relations are, as we show, exponentially more succinct than other representation systems such as Trio's ULDBs [14] and WSDs [12], the representation system used in the first version of MayBMS. That is, there are (relevant) world-sets that necessarily take exponentially more space to represent by ULDBs or WSDs than by U-relations.

- **Leveraging RDBMS technology**: U-relations allow for a large class of queries (positive relational algebra extended by the operation "possible") to be processed *using relational algebra only*, and thus efficiently in the size of the data. Our approach is the first so far to achieve this for the above-named query language. Indeed, this not only settles that there is a succinct and complete *attribute-level* representation for which the so-called tuple Q-possibility problem for positive relational algebra is in polynomial time (previously open [11]) but puts a rich body of research results and technology at our disposal for building uncertain database systems.

  This makes U-relations the most efficient and scalable approach to managing uncertain databases to date.

- **Parsimonious translation**: The translation from relational algebra expressions on the logical schema level to query plans on the physical representations replaces a selection by a selection, a projection by a projection, a join by a join (however, with a more intricate join condition), and a "possible" operation by a projection. We have observed that state-of-the-art RDBMS do well at finding efficient query plans for such physical-level queries.

Ease of use: A main strength of U-relations is their simplicity and low "cost of ownership":

- The representation system is purely relational and in close analogy with relational representation schemes for vertically decomposed data. Apart from the column store relations that represent the actual data, there is only a single auxiliary relation W (which we need for computing certain answers, but not for possible answers).

- Query evaluation can be fully expressed in relational algebra. The translation is quite simple and can even be done by hand, at least for moderately-sized queries.
- The query plans obtained by our translation scheme are usually handled well by the query optimizers of off-the-shelf relational DBMS, so the implementation of special operators and optimizer extensions is not strictly needed for acceptable performance.

In terms of designing an API for uncertain databases, in this work we present novel techniques for database programming on uncertain databases. To this end, the contributions are:

- **Updates.** Updating uncertain databases presents a challenge as often a compressed representation is used that stores a single copy of a tuple appearing in multiple worlds. Updates can require decompressing the representation to allow changing a tuple in some worlds only. We discuss techniques for implementing updates on several representation systems, such as U-relations. Our techniques preserve compactness of the representation despite the need for decompression.
- **Programming model.** We describe a programming model for developing applications for uncertain databases where users can interactively execute queries and updates on the database and process the results in a high-level language. Our model is independent of the underlying representation. For that we adopt the possible worlds semantics. Conceptually, programs run on all worlds in parallel.
- **Observational determinism.** We study a class of programs whose behavior on uncertain databases is indistinguishable to the user from that on

complete/certain databases, without restricting the way programs behave in the background where no user interaction occurs. We call such programs observationally deterministic and argue that this property is crucial if we want to design efficient programs for uncertain databases with intuitive user-friendly interfaces. We also devise an efficient algorithm for deciding *statically* whether a program satisfies observational determinism. The underlying idea consists of examining the output sent to the user in terms of the query that produces it and checking whether the query can return uncertain results. As a side effect we obtain an efficient heuristic for deciding tuple q-certainty, i.e. whether a tuple is certain in the answer to a query. The heuristic involves positive relational algebra operators only, which are often efficiently implementable on succinct representations.

- **Optimizing database programs.** Avoiding iteration over the possible worlds is crucial for achieving efficient execution. For that we need to map programs with update operations into ones that execute in bulk on all worlds. However, it is not clear how to deal with branching and for-loops in the programs, as the control flow can be very different in each world. For that we provide rewrite rules that map nested update programs into sequences of simple update programs that execute on all worlds. These results, while important in the context of uncertain databases, are relevant also in the case of certain databases.

The rest of the dissertation is split into three main parts. The first one proposes a model called U-relational databases for representing sets of possible worlds, and studies its properties, as well as efficient query evaluation and optimization techniques on top of them. The second part discusses challenges when designing and implementing APIs for probabilistic databases and pro-

poses solutions for optimizing programs on top of probabilistic databases. In the last part of the thesis we present the MayBMS probabilistic database management system. We discuss its implementation inside the open-source DBMS PostgreSQL, including data structures and query language. We also discuss how applications can be built on top of MayBMS, and present a web-based application that uses MayBMS as backend and provides an interface for analyzing social networks and random graphs. We start by discussing necessary definitions used throughout the dissertation in Chapter 2, and related work in Chapter 3.

CHAPTER 2

PRELIMINARIES

We use the named relational model with the operations attribute renaming ρ, selection σ, projection π, product $\times$, union $\cup$, and difference $-$. A database schema is a tuple $(R_1[U_1], \ldots, R_m[U_m])$ where each R_i is a relation name, and $U_i = \{A_{i,1}, \ldots, A_{i,m_i}\}$ is a set of attributes names. We will denote by $\mathbf{sch}(R_i)$ the set of attributes of R_i. A relation over schema $R_i[U_i]$ is a set of tuples $(A_{i,1} : a_{i,1}, \ldots, A_{i,m_i} : a_{i,m_i})$ where $a_{i,j}$ are domain values. A database $\mathcal{A}$ over schema Σ is a structure $(R_1^{\mathcal{A}}, \ldots, R_m^{\mathcal{A}})$, where $R_i^{\mathcal{A}}$ is a relation over schema $R_i[U_i]$. A (finite) set of possible worlds over schema Σ is a set of databases $\mathcal{A}_1, \ldots, \mathcal{A}_n$ over schema Σ.

A probabilistic database for a world-set over schema Σ is a pair (W, rep) where W is a database, rep is a function that maps W to a world-set $\mathbf{A}$ over Σ and a probability distribution over the instances in $\mathbf{A}$.

Let Q be a query in relational algebra, and $\mathbf{A}$ be a world-set over schema $\Sigma = (R_1[U_1], \ldots, R_m[U_m])$. The result of Q evaluated on $\mathbf{A}$ under the *possible worlds semantics* is defined as a set of worlds $\mathbf{A'}$ where:

$$\mathbf{A'} = Q(\mathbf{A}) = \left\{ \left(R_1^{\mathcal{A}}, \ldots, R_m^{\mathcal{A}}, Q(\mathcal{A}) \right) \mid \mathcal{A} \in \mathbf{A} \right\}$$

We use the possible world semantics to define the result of a query Q on top of a probabilistic database (W, rep):

$$Q(W, rep) = Q(rep(W))$$

CHAPTER 3

RELATED WORK

There has been extensive work on modelling and processing uncertainty from two major scientific communities. Uncertainty has been studied from the viewpoint of artificial intelligence in areas such as knowledge representation, answer set semantics and Bayesian networks and graphical models. The database community has realized the need to manage uncertainty soon after the introduction of the relational model by Codd, but most of the interest the problem received has occurred only recently with many of the results being produced in the past several years. Below we summarize the major developments in uncertainty management from the perspective of the two communities.

3.1 Uncertainty from the AI perspective

Graphical models and Bayesian networks

Graphical models use graphs to store and manipulate joint probability distributions [49, 42, 17] and have been studied and used by several communities, including AI, statistics, physics, vision, error-correcting codes and neural networks. Nodes in graphical models are random variables, and links between them are used to denote correlations. Given a graphical model and sets of nodes A and B, one can say that A and B are conditionally independent given a third set, C, if all paths between the nodes in A and B are separated by a node in C. Researchers sometimes use directed graphical models, called Bayesian networks [17]. The direction of the arcs can be interpreted as causality: a link from A to B indicates that A causes B. For directed networks the independence prop-

erty can be expressed as follows. Variable a is independent of variable b given evidence $E = \{e_1, \ldots, e_k\}$ if there is a *d-connecting path* from a to b given E. A path from a to b given E is called d-connecting, if every node on the path is linear and diverging and not a member of E, or it is converging and either the node or one of its descendants is in E.

Graphical models and Bayesian networks have a structural component: the graph of nodes, and a parametric component: the probability distribution. For Bayesian networks the parametric component is expressed as conditional probability tables (CPT) of a node and its parents. Given a node n with parents $p_1, \ldots, p_k$, the CPT for n contains the conditional probabilities for the values x_n of n given the values x_i for its parents: $Pr(x_n|x_1, \ldots, x_k)$. The joint probability distribution over all variables can then be expressed as the product of the CPTs for each node. For undirected graphical models the basic parametric unit is the joint probability distributions of cliques in the graph, which are also called factors. The joint probability distribution is then expressed as a product of the factors of the model.

Learning the structure and parameters of a graphical model is one of the hard tasks in this area. Graphical models are used to reason about independence of variables, and to represent the joint probability distribution of a set of variables in a more compact form as a product of factors.

One of the most common problems given a graphical model is probabilistic inference such as computing the marginal probability, or summing up over irrelevant variables. Let X be the set of random variables described the graphical model, and $E \subseteq X$ be a subset of those, and let $F = X - E$. Then marginalization

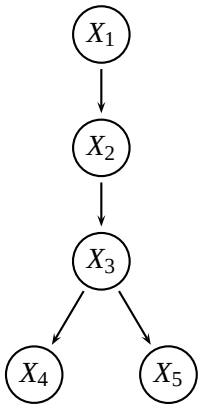


Figure 3.1: Bayesian network.

of E requires computing the probability

$$Pr(E = x_E) = \sum_{x_F} Pr(E = X_E, F = x_F)$$

Marginalization is known to be NP-hard in general [20] and different techniques exist that attempt to reduce this complexity, either by applying heuristic search techniques, or by approximation algorithms. The complexity of probabilistic inference is related to the *treewidth* of the graphical model, or how closely it resembles a tree. For graphical models with tree structure, the inference problem can be solved in polynomial time, otherwise it is exponential in the treewidth of the model [52]. Figure 3.1 shows a tree-structured directed graphical model.

Multiple techniques exist for probabilistic inference in graphical models. One of the most famous ones is the variable elimination method also called bucket elimination [70, 27]. Variable elimination is a technique for computing the marginal probabilities of a single random variable from a joint distribution. The input to the procedure is an ordered set of variables and a set of clauses. At each step the algorithm takes the first variable from the list and considers its possible values. It then sums up the probabilities of the clauses where the oc-

currences of the selected variable are replaced by the corresponding value. The variable is then removed from the list and the same technique is applied to the modified clause sets and the next variable from the list.

3.2 Uncertainty from the DB perspective

Research in managing uncertainty from the database perspective can be separated in several categories, which we review next. We start by reviewing several of the most important models for representing uncertainty in databases, and the corresponding techniques for evaluating queries on top of those models. We discuss evaluating confidence queries in a separate subsection, as this problem has received a lot of attention from the database community and has a large body of work devoted to it. We also discuss query languages, and ranking for uncertain databases.

3.2.1 Representations and query processing

Null values

The oldest and simplest form of dealing with uncertainty in the relational model was the addition of null values to the model [18, 19]. Null values can have a different meaning, the most common ones being "there is no value for the field" and "there is a value but it is unknown". In addition of being too restrictive, the implementation of null values in standard DBMSs typically uses semantics based on a three-valued logic. This can lead to confusing and unexpected re-

sults; for example the tautological query "select * from R where a < 5 or a ≥ 5" returns no results when the *a* column contains null values.

c-tables

The seminal work on managing incomplete information in the relational setting was done in the early 80s by Imielienski and Lipski [34]. They introduce several formalisms for modelling uncertainty with varying expressive power, and study their properties. *V-tables* are tables with labeled nulls, such that a variable can appear more than once in the database. Each valuation mapping the variables to domain values defines a possible world. Naturally, when a variable appears more than once in the database all possible worlds will have the same value for the fields containing that variable. *C-tables* are v-tables where each tuple is assigned a boolean condition over the variables (so-called local condition). The conditions constrain the worlds where a tuple is present: if under a certain valuation a tuple's condition is not satisfied, that tuple is not in the possible world defined by the valuation. Aside from local conditions, c-tables can also have a global condition that serves as an integrity constraint for the possible worlds: the possible worlds are computed for those valuation that satisfy the c-table's global condition.

Example 3.1. Figure 3.2 shows one way of representing the information of Example 1.4 as a c-table. The c-table uses variables x, y, z, u, v, w to represent the missing information. The global condition Φ gives the correlations between the variables. For example the second clause of the conjunction expresses that the apartment number and zip code are correlated. The local conditions of the tuples express when the tuples exist in the database. For example the fact that

R	FN	MI	LN	Street	Apt	City	State	Zip	cond
t_1	Lilly	R	An	Stewart	y	Ithaca	NY	z	$x = 1$
t_1	Lilly	NULL	An	Stewart	y	Ithaca	NY	z	$x = 2$
t_2	Mimi	u	Pan	Stewart	y	Ithaca	NY	z	
t_3	Ron	v	Wang	Aurora	10	Ithaca	NY	14850-1234	$w = 1$
t_3	Ron	v	Wang	El Camino	10	Mountain View	CA	94040-0178	$w = 2$

$$\Phi = (x = 1 \vee x = 2) \wedge (y = 5 \wedge z = 14850 - 2176 \vee y = 6 \wedge z = 14850 - 2154) \wedge (u = D \vee u = O) \wedge (v = P \vee v = Q) \wedge (w = 1 \vee w = 2)$$

Figure 3.2: c-table representation for Example 1.4.

Lilly's middle initial is either 'R' or missing is encoded by having two tuples in the c-table with mutually exclusive conditions ($x = 1$ and $x = 2$). In addition the global condition Φ specifies that the possible values for x are either 1 or 2, thus at least one of the two tuples will exist in the table in each world. □

Relations with or-sets [35] can be viewed as v-tables, where each variable occurs only at a single position in the table and can only take values from a fixed finite set, the or-set of the field occupied by the variable. An example of or-set relations was given in Figure 11.1 of Chapter 1. While v-tables and or-set relations have limited expressive power, c-tables are shown to form a strong representation system for relational query languages.

Tuple-independent and block-independent databases

A tuple-independent database (see e.g. [24, 25, 26]) is a database where each tuple in a relation is associated with a boolean random variable with given truth probability, which describes the event of the tuple being in the database. Variables are independent of each other, and a possible world is defined by selecting tuples independently at random with the corresponding probability. The prob-

S	A	B	prob
s_1	m	1	0.6
s_2	n	1	0.5

T	C	D	prob
t_1	1	p	0.4

Figure 3.3: Tuple-independent database.

ability of a world is then the product of the truth-probabilities of the tuples that were selected in the world, multiplied by the product of the falsehood probability of the ones that were not selected.

Example 3.2. Figure 3.3 shows an example of a tuple-independent database from [24]. The database has two relations S and T, each of them containing probabilistic tuples. For example S has two tuples s_1 and s_2, that exist in the database with probability 0.6 and 0.5, respectively. The probabilistic database defines a set of 8 possible worlds, corresponding to the 8 possible subsets of the three tuples s_1, s_2 and t_1. The probability of the world $\{s_1, s_2\}$ for example is $0.6 \cdot 0.5 \cdot 0.6 = 0.18$. □

Since tuple-independent databases are not expressive enough to represent the results of any query, [24] and [25] consider only queries which close the possible worlds semantics by computing the confidence of tuples in the result of the query. The authors distinguish between extensional and intensional evaluation of the confidence computation. In the first case, confidence computation is done as part of the relational algebra operators, and can thus be evaluated efficiently. For example evaluating a join results in multiplying the probabilities of the joining pair of tuples, and projection involves computing the probability of the disjunction over each set of duplicate tuples. The intensional evaluation corresponds to evaluating directly as prescribed by the definition of the confidence operator - computing the tuples in each world, and aggregating their probability across worlds. The extensional query evaluation produces correct results only

for the so-called *safe plans*. Intuitively safe plans are query plans that preserve the independence of intermediate tuples and for which one can apply simple formulas to compute the conjunction or disjunction of independent events, as needed for joins and projections. Dalvi and Suciu show a dichotomy result [26], which defines a class of queries, the so-called *hierarchical queries* for which safe plans exist. For the remaining queries they show that their evaluation is necessarily hard on tuple-independent databases. The "bad" queries are shown to be those containing the subquery

$$q() : -L(x), J(x, y), R(y)$$

The results are further extended for *block-independent* databases [51] which are a generalization of tuple-independent databases. A block-independent database contains relation where tuples are grouped in clusters, or blocks. Tuples within a block are mutually exclusive and their probabilities sum up to $p < 1$. Blocks are independent of each other. To construct a world one selects at most one tuple from each block (or exactly one, if the probabilities in the block sum up to 1). A tuple-independent database is thus a block-independent database with one tuple per block. [51] considers conjunctive queries with HAVING predicates, where the aggregate is one of EXISTS, MIN, MAX, COUNT, SUM, AVG, or COUNT(DISTINCT), and the comparison in the having clause is one of ¡,¿,*le*,≥ or =. The complexity of evaluating a HAVING query is shown to depend on the comparison that Q uses in the HAVING clause, and on the exact aggregation function. For each aggregate α, the article defines a class of HAVING queries, called α-safe, such that for any Q using α one of the following is true: (1) if Q is α-safe then Qs data complexity is in P; (2) If Q has no self joins and is not α-safe then, Q has #P-hard data complexity; (3) It cannot be decided in PTIME (in the size of Q) whether Q is α-safe.

[54] shows that safe plans correspond to inference problems in tree-structured graphical models, hence the polynomial complexity.

ULDBs

ULDBs, or databases with uncertainty and lineage [15], are the representation model employed by the Trio probabilistic DBMS [43, 53, 3] developed at Stanford. A ULDB relation is a set of *x-tuples*, where each x-tuple represents a set of alternatives. One world is defined by choosing precisely one alternative of each x-tuple. A world may contain none of the alternatives of an x-tuple, if this x-tuple is marked as optional (or *maybe*) using the **?**-symbol. Dependencies between alternatives of different x-tuples are enforced using *lineage*: An alternative i of an x-tuple s occurs in the same worlds with an alternative j of another x-tuple t if the lineage of (s, i) points either to (t, j), or to another alternative that transitively points to (t, j). The lineage of an alternative can also point to an *external* symbol (t, j), if there is no alternative (t, j) in the database [15]. During query processing tuples in the result are annotated with the tuple and alternative id of the input tuples; this is the resulting tuple's lineage.

In terms of expressive power, U-relational databases and ULDBs are equivalent. However, there are several major differences between the two. First, U-relational databases support attribute-level uncertainty and can thus represent data more compactly than ULDBs which are tuple-level. Second, the lineage of a tuple in ULDBs only points to its immediate predecessors in the query plan. Deciding on whether a tuple is possible or not requires the recursive expansion of the tuple's lineage down to the base relations to make sure it does not contain conflicting alternative ids. In U-relational databases, on the other hand,

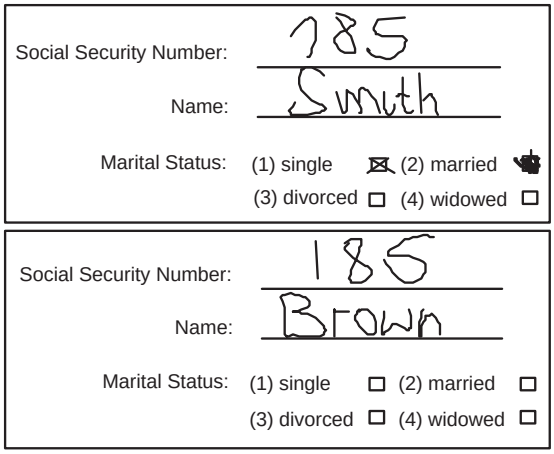
query evaluation takes care that only valid tuples are produced. What is more important, querying on U-relational databases can be encoded using relational algebra only, whereas in ULDBs it can require the transitive expansion of the lineage.

There are also differences between in the implementation of Trio as apposed to MayBMS. While MayBMS supports an extension of SQL with cleanly defined semantics based on possible worlds [9, 7], Trio's language TriQL [63] has constructs that allow the user to express conditions on the internal system representation details. This makes the language non-generic (see e.g. [2]), which means that query results depend on the representation and two equivalent world-sets represented in a different way can result in two different result sets. In addition, MayBMS is implemented directly inside of PostgreSQL as opposed to being implemented as a layer on top of that, like Trio. This is the reason for several advantages of MayBMS including as we shall see later more efficient query evaluation (see Chapter 9), and ease of writing applications that use the probabilistic database system as a backend (Chapter 13).

Probabilistic world-set decompositions

World-set decompositions [8, 10, 11, 48, 12] are the predecessor of U-relational databases in the first version of MayBMS, where each variable c_i of a U-relation corresponds to a WSD *component* relation C_i and each domain value l_i of c_i corresponds to a tuple of C_i.

Example 3.3. Figure 3.4(a) shows two example census forms that need to be scanned and stored in a database. Due to the unclear writing and ambiguous entries, several instances of the database are possible. Figure 3.4(b) shows a



(a) Two census survey forms

t_1.S t_2.S	Pr
185 186	0.2
785 185	0.4
785 186	0.4

$\times$

t_1.N	Pr
Smith	1

$\times$

t_1.M	Pr
1	0.7
2	0.3

$\times$

t_2.N	Pr
Brown	1

$\times$

t_2.M	Pr
1	0.25
2	0.25
3	0.25
4	0.25

(b) Probabilistic world-set decomposition for the two census forms of (a)

Figure 3.4: Modeling uncertainty with probabilistic world-set decomposition.

world-set decomposition representing the set of possible worlds for the census forms. The world-set decomposition consists of several component relations, where each component defines the possible values for a set of fields of the relation. Fields that are dependent are defined in the same component, and those that are independent can be kept in separate components, thus reducing the size needed to represent them. For example, due to the uniqueness constraint on the social security number, the SSN fields of the two tuples are defined in the same component, thus making sure the invalid combination where both are 185 is excluded. However, the marital status fields are independent of each and of the SSN, thus they can be kept in separate components. To obtain a possible world represented by the WSD, we choose one tuple from each component, and use

the values defined in that tuple for the respective field values. □

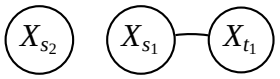
As mentioned above, WSDs are essentially normalized U-relational databases where each variable c_i of a U-relation corresponds to a WSD *component* relation C_i and each domain value l_i of c_i corresponds to a tuple of C_i. The normalization may lead to an exponential blow-up in the database sizeand accounts for U-relations with arbitrarily large ws-descriptors being more compact than U-relations with singleton ws-descriptors and thus than WSDs. Positive relational queries have polynomial data complexity for U-relations (Chapter 5) and exponential data complexity for WSDs [48]. This can be explained in close analogy to the difference in succinctness and by the fact that query evaluation creates new dependencies [24]: U-relations can efficiently store the new dependencies by enlarging ws-descriptors, whereas WSDs correspond to U-relations with normalized ws-descriptors, hence the exponential blowup. Finally, the query translations employed by the evaluation algorithms in the WSD and U-relational cases are different. Whereas for WSDs all operators are translated to sequences of relational queries and in the case of projection and join even to fixpoint programs [12], the translation remains strictly in relational algebra for U-relations.

[48] provides complexity results for different decision problems on WSDs, such as query possibility and certainty, and presents a polynomial algorithm for relational decomposition. [12] discusses the problem of chasing dependencies on probabilistic WSDs. Given a set of possible worlds and a set of dependencies such as functional dependencies and other equality generating dependencies, the chase removes those worlds where the constraints do not hold. On WSDs the procedure involved identifying tuples that potentially violate a constraint in

S	A	B	
s_1	m	1	X_{s_1}
s_2	n	1	X_{s_2}

T	C	D	
t_1	1	p	X_{t_1}

(a) Probabilistic relations



X_{s_2}	$f^{ind}_{s_2}$
0	0.5
1	0.5

X_{t_1}	X_{s_1}	$f^{nxor}_{t_1,s_1}$
0	0	0.4
0	1	0.2
1	0	0
1	1	0.4

(b) Graphical model for the variable correlations in (a)

Figure 3.5: Using graphical models for probabilistic databases.

a world, merging the respective components and excluding the violating value combinations.

Relations and graphical models

Several recent work attempt to marry relational databases with AI models for representing uncertainty. In [55, 56] probabilistic databases are modeled using graphical models. Like in tuple-independent databases, each tuple has an associated random variable, specifying the existence of the tuple in the database, and correlations between tuples are given by links between the corresponding nodes in the graphical model. A factor $f(\mathbf{X})$ is defined as a function of a (small) set of random variables $\mathbf{X}$: $0 \leq f(\mathbf{X}) \leq 1$. Factors are the connected components in a graphical model for sets of dependent variables. The probability distribution for the variables is given in the form of the joint probability distribution table for each of the factors.

Example 3.4. Figure 3.5 shows a modified version of the tuple-independent

database example in 3.3 where tuples are no longer independent, but exhibit an a *nxor* relationship, that is high positive correlation between s_1 and t_1. The probability of the possible worlds in this case is computed according to the factors in the graphical model, for example

$$\begin{aligned} Pr(\{s_1, s_2\}) &= Pr(X_{s_1} = 1, X_{s_2} = 1, X_{t_1} = 0) \\ &= f^{ind}_{s_2}(X_{s_1} = 1) f^{ind}_{t_1,s_1}(X_{t_1} = 0, X_{s_1} = 1) = 0.5 \cdot 0.2 = 0.1 \end{aligned}$$

The probability of the world $\{s_2, t_1\}$ is 0, as when t_1 is in the database, s_1 must also be there according to the graphical model. □

Querying the graphical model is done by introducing new factors. For example a join will introduce an "and" factor f^{and} for each pair of joining tuples, such that the probability of the tuple being in the result is 1 iff the random variables for the input tuples have value true. A projection query will introduce an "or" factor f^{or} for each set of duplicate tuples from the input. Computing confidence of tuples is reduced to probabilistic inference in graphical models.

Note that world-set decompositions correspond to flat graphical models, where the conditional independence between variables is made explicit. Indeed, WSDs are based on the idea of independence between variables (attribute values), which is a special kind of conditional independence. In some cases, graphical models can be more succinct than WSDs. However, evaluating relational algebra queries on top of graphical models tends to produce flat models with high treewidth, which makes confidence computation hard on graphical models.

The BayesStore system [66] uses *first-order graphical models* to represent uncertainty. Those define first-order factors over so-called stripes: a family of ran-

dom variables from the same probabilistic attriibute. First-order factors represent a family of local models which share the same structure and conditional probability table. For example, in a database containing sensor readings a stripe can be defined to denote the temperature value of one particular sensor for all time points, and another stripe for all light readings of that sensor. A first-order factor can then be used represent the fact that light and temperature are correlated.

3.2.2 Confidence computation

Confidence computation is known to have very high computational complexity: the problem is #P-hard in the general case. Attempts to provide solutions to this problem have gone in several main directions: exact computation for restricted cases, exact computation using heuristics, and approximate computation using MCMC methods.

SPROUT

The SPROUT [39, 47] project aims at developing efficient exact and approximate algorithms for confidence computation in probabilistic databases. [39] proposes an approach where a DNF formula is compiled into a tree (so called ws-tree) by using variable elimination and independence decomposition techniques. Given such a ws-tree, the probability of the initial DNF can be computed in time linear in the size of the tree. While the size of the tree can in general be exponential in the DNF size, certain cases can be efficiently solved by this algorithm. [47] extends the approach by introducing efficient methods for estimating the prob-

ability of a partially refined DNF formula, and allowing the algorithm to stop early when a sufficiently close approximate probability value is reached.

MCDB

MCDB [36] is a pure Monte Carlo approach to managing uncertain data. In MCDB relations can be deterministic (i.e., the same in all worlds), or random, where uncertainty is specified with the so called VG (variable generating) functions. VG functions are responsible for generating possible values according to a given probability distribution, thus in effect generating relation instances. For example, a VG function can generate values for an attribute according to a normal distribution. A VG function can also be used to specify attribute correlations; in this case the output of the VG function will be a multicolumn table.

When queries are evaluated, the VG functions are called to generate random instances of the database, and the query is evaluated in each world. As optimization, MCDB uses the so called *tuple bundles*, that is a bundle of tuples representing the possible for a tuple across several worlds, and when possible operations are evaluated directly on the tuple bundle without decompressing it. Also, materialization is delayed as much as possible

The main advantage of MCDB is that it does not restrict the class of supported queries and can deal with complex constructs such as aggregates, distinct and difference that are typically avoided by probabilistic DBMSs. Since queries are evaluated in each world, as long as the standard DBMS query engine supports it, the query can be evaluated on the set of possible worlds. The disadvantage is that this approach may lead to premature sampling of unneces-

sary values that will later be dropped by the query. Another problem with this is that it is hard to give error guarantees for the answer, or to control the number of samples needed to provide such guarantees.

3.2.3 Ranking

Research on ranking and computing top-k answers have led to multiple publications in the recent years [60, 50, 67, 68, 41, 61, 62, 21]. Below we summarize a subset of that work.

Ranking tuples by confidence

In [50] Re et al study the problem of computing the top-k most likely tuples in the result of a query over a probabilistic database. This is motivated by the fact that typically the probabilities in a probabilistic database are uncertain themselves, and tuple probabilities should rather be taken relevant to the other probabilities in the database, as opposed to considering their absolute value. Thus the concrete probability values are not necessary, they only serve to order the tuples in the result. This work therefore considers using Monte Carlo simulation techniques to compute approximate confidence values. A property of MCMC is that the tighter the requested confidence interval is, the more simulations should be run. Since the authors are only interested in ranking the tuples, the simulations can be stopped when the guaranteed confidence interval is small enough so it does not intersect with the confidence intervals of other tuples and the tuple's rank can be determined. This observation also helps drop tuples that have no chance of entering the top-k set, and simulations can be concentrated

on the remaining tuples.

3.2.4 Ranking in uncertain databases

Several papers consider the general problem of ranking when data is uncertain. [60] considers as data model a tuple-independent database extended with so-called *tuple-generation rules*. These are logical formulas over the variables associated with each tuple and define the valid possible worlds. For example the rule $t_2 \oplus t_3$ denotes that tuples t_2 and t_3 are mutually exclusive. The paper considers several flavors of the ranking problem, including the topk tuples in the most probable world, the most probable topk tuples that belong to valid possible world, and the set of most probable topith tuples across all possible worlds, where $i = 1 \ldots k$. An *uncertain topk query* returns a vector of tuples T^* which is most likely to be the top-k result over all worlds. If $\mathcal{T}$ is a set of k-length tuple vectors such that each vector is ordered according to the given scoring function, and each $T \in \mathcal{T}$ is a top-k answer in a non-empty set of worlds $PW(T)$. Then the uncertain top-k result T^* can be defined as follows:

$$T^* = argmax_{T \in \mathcal{T}} \left(\sum_{w \in PW(T)} (Pr(w)) \right)$$

That is, the tuples returned in this case belong to the same world, and have the maximum aggregated probability of being top-k across all possible worlds. *Uncertain k-rank query* is a query returning a vector of tuples, where each tuple is the most probable one at the respective position. Evaluating uncertain ranking queries is done by navigating the search space where each state is a ranked prefix of l tuples that appears as the top-l answer in a non-empty set of possible worlds. The state is complete when $l = k$. Navigation is done by extending

the partial states to complete states, which is done in sorted order of the state probabilities.

3.2.5 Applications for probabilistic databases

Data cleaning

Data cleaning is a fully or semi-automated process involving entity resolution, clustering and data fusion. Typically, the input to data cleaning is a dirty relation, where entries have different types of errors and ambiguities, such as typos, duplicated entries, missing values, use of synonyms to denote the same objects, etc. The result is usually a set of tuple clusters where each cluster contains tuples that are duplicates of each other. Several works consider the use of probabilistic databases to represent the results of data cleaning [4, 33]. In [33] Hassanzadeh and Miller propose a framework supporting the different phases of data cleaning. The input to the system is a dirty relation, and the output is a probabilistic database. The authors consider as result block-independent (a.k.a. disjoint-independent) databases as the representation module due to their simplicity and known efficient techniques for query processing on top of that. The framework consists of the following three components:

- **Similarity join module**. This phase computes pairs of tuples together with a similarity score showing how similar the two records are. Different similarity measures can be used in this phase, such as edit distance or cosine similarity.
- **Clustering module**. The clustering module computes groups of tuples

that are duplicates of each other based on the similarity score between pairs of tuples from the previous step.

- **Probability assignment module**. This module assigns probabilities to the tuples in a cluster which reflect the error in the tuple. One of the proposed techniques involves computing a representative of each cluster, and the probabilities of the other tuples are computed with respect to their similarity to the representative of the cluster.

Part II

Representing and querying uncertain data

CHAPTER 4

U-RELATIONAL DATABASES

In this chapter we formally define and study U-relational databases, which were already introduced by example in Chapter 1.

A *world table* is a table over schema $W(Var, Rng, Pr)$, containing the domain values for a set of variables together with their probability distribution. Given a world table W, a *world-set descriptor* (or *ws-descriptor*, for short) is a partial valuation $\overline{d} = \{c_1 \mapsto l_1, \ldots, c_k \mapsto l_k\}$ consistent with W. We will use a total valuation to describe a possible world, and a world-set descriptor to describe a set of possible worlds. We will use an empty world-set descriptor to describe the entire world-set.

Example 4.1. The world-table W in Fig. 1.5 defines five variables x, y, z, v, w, whose common domain is $\{1, 2\}$. The number of worlds defined by W is $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32$. □

We are now ready to define databases of U-relations.

Definition 4.2. A *U-relational database* for a world-set over schema $\Sigma = (R_1[\overline{A_1}], \ldots, R_k[\overline{A_k}])$ is a tuple

$$(U_{1,1}, \ldots, U_{1,m_1}, \ldots, U_{k,1}, \ldots, U_{k,m_k}, W),$$

where W is a world-table and each relation $U_{i,j}$ has schema $U_{i,j}[\overline{T}_{R_i}; \overline{B_{i,j}}; \overline{D}_{i,j}]$ such that $\overline{D}_{i,j}$ defines ws-descriptors over W, $\overline{T}_{R_i}$ defines tuple ids, and $\overline{B_{i,1}} \cup \cdots \cup \overline{B_{i,m_i}} = \overline{A}_i$.

A ws-descriptor $\{c_1 \mapsto l_1, \ldots, c_k \mapsto l_k\}$ is relationally encoded in $\pi_{\overline{D}_{i,j}}(U_{i,j})$ of arity $n \geq k$ as a tuple $(c_1, l_1, \ldots, c_k, l_k, c_{k+1}, l_{k+1}, \ldots, c_n, l_n)$, where we pad the vector

up to arity n by reusing pairs c_i, l_i where $1 \leq i \leq k$ in positions $j > k$.

Although we speak of vertical partitioning, we do not require the value columns of $U_{i,j}$ to disjointly partition the columns of R_i. Indeed, overlap, as considered in view materialization may be useful to speed up query evaluation, see e.g. [64].

We next define the semantics of a U-relational database. To obtain a possible world we first choose a total valuation f over W. We then process the U-relations tuple by tuple. If the function f extends[1] the ws-descriptor $\overline{d}$ of a tuple of the form $(\bar{t}, \overline{a}, \overline{d})$ from a U-relation of schema $(\overline{T}, \overline{A}, \overline{D})$, we insert in that world the values $\overline{a}$ into the $\overline{A}$-fields of the tuple with identifier $\bar{t}$. In general this may leave some tuples partial in the end (i.e., the values for some fields have not been provided). These tuples are removed from the world.

We require, for a U-relational database $(U_1, \ldots, U_n, W)$ to be considered valid, that the representation does not provide several contradictory values for a tuple field in the same world. Formally, we require, for all $1 \leq i, j \leq n$, and tuples $t_1 \in U_i[\overline{T}_i, \overline{A}_i, \overline{D}_i]$ and $t_2 \in U_j[\overline{T}_j, \overline{A}_j, \overline{D}_j]$ such that U_i and U_j are vertical partitions of the same relation, that if there is a world that extends both $t_1.\overline{D}_i$ and $t_2.\overline{D}_j$, then for all $A \in (\overline{A}_i \cap \overline{A}_j)$, $t_1.A = t_2.A$ must hold.

Example 4.3. Suppose there are two U-relations with schemata $U_1[T_R; A, B; \overline{D_1}]$ and $U_2[T_R; B, C; \overline{D_2}]$ that jointly represent columns A, B, and C of a relation R. Assume tuples $(t_1, a, b, c_1, 1) \in U_1$ and $(t_1, b', c, c_2, 2,) \in U_2$, $b \neq b'$. Then U_1 and U_2 cannot form part of a valid U-relational database because there would be a world with $c_1 \mapsto 1, c_2 \mapsto 2$ in which the tuple from U_1 requires field $t_1.B$ to take value b while the tuple from U_2 requires the same field to take value b'. □

[1] That is, for all x on which $\overline{d}$ is defined, $\overline{d}(x) = f(x)$.

An important property of U-relational databases is that they form a *complete representation system* for finite world-sets.

Theorem 4.4. *Any finite set of worlds can be represented as a U-relational database.*

Proof. Let $\mathbf{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ be a finite world-set over schema Σ, and let $Pr(\mathcal{A}_i) = p_i$ be the probability distribution for the world-set. We encode $\mathbf{A}$ as a U-relational database in the following way. Let x be a variable with domain [1,n]. Then the world-table $W = \{(x, i, p_i) \mid 1 \leq i \leq n\}$. For each relation $R_i[U_i]$ in Σ create a U-relation $U_i[T_{R_i}; U_i; D]$ where we have a tuple $(t; a; x \mapsto k) \in U_i$ iff $(t, a) \in R_i^{\mathcal{A}_k}$. □

Since any finite world-set can be represented as a U-relational database, the result of any query evaluated on top of a U-relational database can be represented in U-relational form as well:

Corollary 4.5. *U-relational databases are closed for any relational query language.*

Remark 4.6. We will sometimes inline the world table W in the ws-descriptors of the U-relations. Thus the ws-descriptors will be sets of triples $(x \mapsto i, p)$ where x is a variable, i is a domain value for x and the tuple $(x, i, p) \in W$.

CHAPTER 5

QUERY PROCESSING

We use the possible worlds semantics to evaluate queries on top of U-relational database. Recall that according to this semantics the result of a query is equivalent to the result obtained by evaluating the query on each world. We are interested in techniques for evaluating the query directly on the representation which computes the *representation* of the result. Note that for U-relational databases this is always possible, as they are complete and closed for relational query languages.

Queries on vertical decompositions. U-relations rely essentially on vertical decomposition for succinct (attribute-level) representation of uncertainty. To evaluate a query, we first need to reconstruct relations from vertical decompositions by (1) joining two partitions on the common tuple id attributes and (2) discarding the combinations that yield inconsistent ws-descriptors. We call this operation *merge* and give its precise definition in Fig. 5.3, where conditions (1) and (2) from above are defined by α and ψ, respectively.

Example 5.1. Consider the U-relational database of Fig. 1.5. The query $\pi_{FName,MI,LName,City}\sigma_{\text{FName='Lilly'}}(R)$ lists the name and city of customers with first name Lilly. To answer this query, we need to *merge* the necessary partitions of R and obtain a new query with $merge(merge(\pi_{\text{FName,LName}}(R), \pi_{\text{MI}}(R)), \pi_{City})$ in the place of R. □

Our query evaluation approach can take full advantage of query evaluation and optimization techniques on vertical partitions. First, it does not require to reconstruct the entire relations involved in the query, but rather only the nec-

essary vertical partitions. Second, necessary partitions can be flexibly merged in during query evaluation. Thus early and late tuple materialization [64] carry over naturally to our framework. For this, our *merge* operator allows to merge two partitions not only if they are given in their original form, but also if they have been modified by queries.

The first advantage only holds for so-called *partition-independent* U-relational databases, which do not have dependencies that need to be propagated from a partition to the other partitions before the partition can be projected away.

An example of U-relational databases that are not partition-independent are non-*reduced* U-relational databases. A U-relational database is reduced if there are no tuples that cannot be completed in any world. That is, each tuple of a reduced U-relation can always be completed to an actual tuple in a world. The advantage becomes evident even for a simple projection query. Consider a partition-independent database containing a U-relation U defining the A attribute of R. To evaluate $\pi_A(R)$ we do not need to merge in all U-relations defining the attributes of R and later project on A. Instead, the answer is simply U.

In the following, we assume that we have no information about the input database. The starting point of our query rewriting will thus be to reconstruct each relation required by the query by merging all partitions of that relation:

$$R = \mathit{merge}(U_1, \ldots, U_n), \quad \text{where } U_1, \ldots, U_n \text{ - partitions for } R \tag{5.1}$$

As we will discuss next, the query evaluation techniques of this section always produces partition-independent U-relations for partition-independent U-relational databases. In Chapter 7 we study partition-independent U-relational databases in more depth and extend the query-evaluation technique

$$merge(\pi_{\overline{X}}(R), \pi_{\overline{A}-\overline{X}}(R)) = R, \qquad \text{where } \overline{A} = \mathbf{sch}(R) \tag{5.2}$$

$$merge(R, S) = merge(S, R) \tag{5.3}$$

$$merge(merge(R, S), T) = merge(R, merge(S, T)) \tag{5.4}$$

$$\sigma_{\phi(\overline{X})}(merge(R, S)) = merge(\sigma_{\phi(\overline{X})}(R), S) \qquad \text{where } \overline{X} \subseteq \mathbf{sch}(R) \tag{5.5}$$

$$merge(R, S) \bowtie_{\phi(\overline{X},\overline{Y})} T = merge(R \bowtie_{\phi(\overline{X},\overline{Y})} T, S) \qquad \text{where } \overline{X} \cup \overline{Y} \subseteq \mathbf{sch}(R) \cup \mathbf{sch}(T) \tag{5.6}$$

$$\pi_{\overline{X}}(merge(R, S)) = merge(\pi_{\overline{X}\cap\overline{A}}(R), \pi_{\overline{X}\cap\overline{B}}(S)) \qquad \text{where } \mathbf{sch}(R) = \overline{A}, \mathbf{sch}(S) = \overline{B} \tag{5.7}$$

Figure 5.1: Algebraic equivalences for relational algebra queries with merge operator.

to databases which are not partition-independent such that we merge only the necessary partitions.

Example 5.2. Consider the following non-reduced database of two U-relations:

U_1	T	A	D
	t_1	a_1	$c_1 \mapsto 1$
	t_2	a_2	$c_2 \mapsto 1$

U_2	T	B	D
	t_1	b_1	$c_1 \mapsto 1$
	t_1	b_2	$c_1 \mapsto 2$

In each U-relation the second tuple cannot find a partner in the other U-relation with which a complete tuple (with both attributes A and B) can be formed. If these second tuples are removed, the database is reduced. □

We can always make a U-relational database partition-independent as follows: We filter each U-relation using semijoins with each of the other U-relations representing data of the same relation R_i. The semijoin conditions are the α and ψ-conditions. For more details, see Algorithm 3 in Chapter 7.

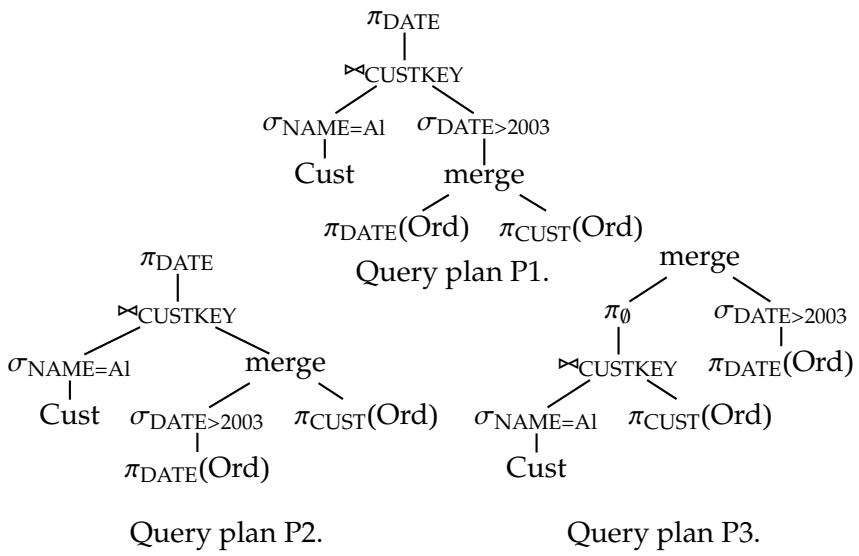


Figure 5.2: Three equivalent query plans.

Algebraic equivalences. Fig. 5.1 gives algebraic equivalences of relational algebra expressions with merge operator on vertical decompositions: merging is the reverse of vertical partitioning, it is commutative and associative, it commutes with selections, joins, and projections.

Standard heuristics known from classical query optimization for relational algebra apply here as well. Intuitively, we usually push down projections and selections and merge in U-relations as late as possible. An interesting new case is the decision on join ordering among an explicit join from the input query and a join due to merging: If the merge is executed before the explicit join, it may reduce the size of an input relation to join. We have seen in our experiments that the standard selectivity-based cost measures employed by relational database management systems do a good job, as long as the queries remain reasonably small.

Example 5.3. Consider a U-relational database $\mathcal{U}$ that represents a set of possible worlds over two TPC-H relations Ord and Cust (short for Order and Cus-

tomer, respectively) [65]. $\mathcal{U}$ has one U-relation for each attribute of the two relations, of which we only list DATE and CUSTKEY for Ord, and NAME and CUSTKEY for Cust. The following query finds all dates of orders placed by Al after 2003:

$$\pi_{\text{DATE}}(\sigma_{\text{NAME}='\text{Al}'}(\text{Cust}) \bowtie_{\text{CUSTKEY}} \sigma_{\text{DATE}>2003}(\text{Ord}))$$

Fig. 5.2 shows three possible plans P1, P2, and P3 using operators on vertical decompositions. The naïve plan P1 first reconstructs Ord from its two partitions then applies the selection and the join with Cust. In P2 and P3 the merge operator is pushed up in the plans, first immediately above the selection (P2), and then above the join operator (P3). Among the three plans, P1 is clearly the least efficient. However, without statistics about the data, one cannot tell which of P2 and P3 should be preferred. If DATE>2003 is very selective, then merging immediately thereafter as in P2 will lead to the filtering of tuples from π_{CUSTKEY}(Ord) and thus fewer tuples will be processed by the join. Is this not the case, then first merging only increases the number and size of the tuples that have to be processed by the join. Also, in P3 all value attributes except for DATE are projected away after the join as they are not needed for the final result. □

Queries on U-relations. Fig. 5.3 gives the function $[\![\cdot]\!]$ that translates positive relational algebra queries with *possible* and *merge* operators into relational algebra queries on U-relational databases.

The possible operator applied on a U-relation U closes the possible worlds semantics by computing the set of tuples possible in U. It thus translates to a simple projection on the value attributes of U. The result of a projection is a U-relation whose value attributes are those from the projection list (thus the input

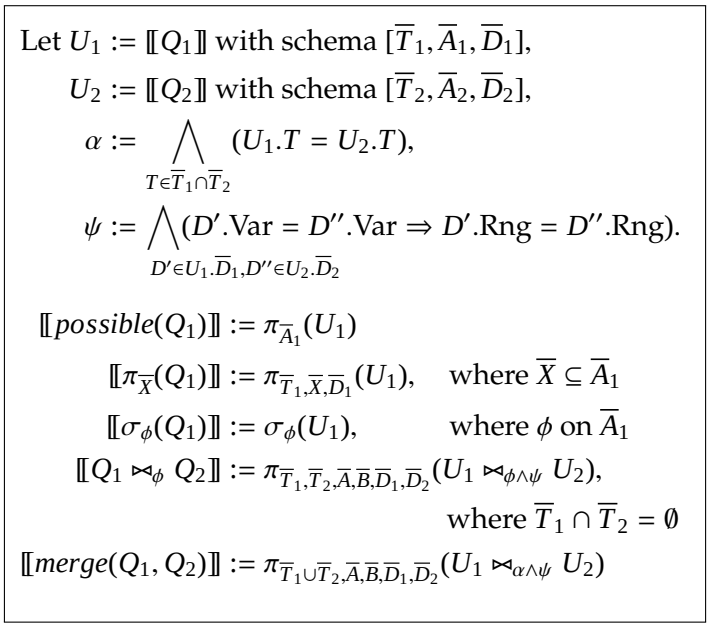


Figure 5.3: Translation of queries with merge into queries on U-relations.

ws-descriptors and tuple ids are preserved). Selections apply conditions on the value attributes.

The merge operator that reconstructs a relation from its vertical partitions was already explained. Similarly to the merge, the join uses the ψ-condition to discard tuple combinations with inconsistent ws-descriptors. Fig. 5.3 gives the translation in case U_1 and U_2 do not contain partitions of the same relation. For the case of self-joins we require aliases for the copies of the relation involved in it such that they do not have common tuple id attributes.

The union of U_1 and U_2 like the ones from Fig. 5.3 is sketched next. We assume that $\overline{A}_1 = \overline{A}_2$, $\overline{T}_1 \cap \overline{T}_2 = \emptyset$, and the tuples of different relations have different ids. To bring U_1 and U_2 to the same schema, we first ensure ws-descriptors of the same size by padding the smaller ws-descriptors with already contained variable assignments, and add new (empty) columns $\overline{T}_2$ to U_1 and $\overline{T}_1$ to U_2. We then perform the standard union.

From our translation $[\![\cdot]\!]$ it immediately follows that

Theorem 5.4. *Positive relational algebra queries extended with the possible operator can be evaluated on U-relational databases using relational algebra only.*

Example 5.5. Recall the U-relational database of Fig. 1.5 storing deduplicated customer information. Consider a query asking for names of people living in Ithaca:

$$S = \pi_{\text{FName,MI,LName}}(\sigma_{\text{City='Ithaca'}}(R))$$

After merging the necessary partitions of relation R and translating it into positive relational algebra, we obtain

$$\pi_{\text{FName,MI,LName}}(\sigma_{\text{City='Ithaca'}}(U_1 \bowtie_{\alpha_1 \wedge \psi_1} U_2 \bowtie_{\alpha_2 \wedge \psi_2} U_3)),$$

where the conditions ψ_1, ψ_2, α_1, and α_2 follow the translation given in Fig. 5.3. The three vertical partitions are joined on the tuple id attributes (α_1 and α_2) and the combinations with conflicting mappings in the ws-descriptors are discarded (ψ_1 and ψ_2). Before and after translation, the query is subject to optimizations as discussed earlier. (In this case, a good query plan would first apply the selections on the partitions, then project away the irrelevant value attributes Street, City and State, and then merge the partitions).

U_6	T_S	FName	MI	LName	D_1	D_2
	t_1	Lilly	R	Ann	$x \mapsto 1$	
	t_1	Lilly	NULL	Ann	$x \mapsto 2$	
	t_2	Mimi	D	Pan	$y \mapsto 1$	
	t_2	Mimi	O	Pan	$y \mapsto 2$	
	t_3	Ron	P	Wang	$v \mapsto 1$	$w \mapsto 1$
	t_3	Ron	Q	Wang	$v \mapsto 2$	$w \mapsto 1$

The above U-relation U_6 encodes the query answer. □

Our translation yields relational algebra queries, whose evaluation always produces tuple-level U-relations, i.e., U-relations without vertical decompositions, by joining and merging vertical partitions of relations. Following the definition of the merge operator, if the input U-relations are partition-independent, then the result of merging vertical partitions is also partition-independent. We thus have that

Proposition 5.6. *Given a positive relational algebra query Q and a partition-independent U-relational database U, $[\![Q]\!](U)$ is a partition-independent U-relational database.*

CHAPTER 6

NORMALIZATION OF U-RELATIONS

U-relations do not forbid large ws-descriptors. The ability to extend the size of ws-descriptors is what yields efficient query evaluation on U-relations. However, large ws-descriptors cause an inherent processing overhead. Also, after query evaluation or dependency chasing on a U-relational database, it may happen that tuple fields, which used to be dependent on each other, become independent. In such a case, it is desirable to optimize the world-set representation [11]. We next discuss one approach to normalize U-relational databases by reducing large ws-descriptors to ws-descriptors of size one. Normalization is an expensive operation per se, but it is not unrealistic to assume that uncertain data is initially in normal form [8, 11] and can subsequently be maintained in this form.

Definition 6.1. A U-relational database is normalized if all ws-descriptors of its U-relations have size one.

Algorithm 1 gives a normalization procedure for U-relations that determines classes of variables that co-occur in some ws-descriptors and replaces each such class by one variable, whose domain becomes the product of the domains of the variables from that class. Fig. 6.1 shows a U-relational database and its normalization.

Theorem 6.2. *Given a reduced U-relational database, Algorithm 1 computes a normalized reduced U-relational database that represents the same world-set.*

Computing certain answers. Given a set of possible worlds, we call a tuple certain iff it occurs in each of the worlds. It is known that the tuple certainty

Algorithm 1: Normalization of ws-descriptors.

Input: Reduced U-relational database $\mathcal{U} = (U_1, \ldots, U_m, W)$
Output: Normalized reduced U-relational database.

begin
  R := the relation consisting of all pairs of variables (c_i, c_j) that occur together in some ws-descriptor of $\mathcal{U}$;
  $\mathcal{G}$:= the graph whose node set is the set of variables and whose edge relation is the reflexive and transitive closure of R;
  Compute the connected components of $\mathcal{G}$;
  foreach *U-relation* $U_j(\overline{T}, \overline{A}, D_1, \ldots, D_n)$ *of* $\mathcal{U}$ **do**
    U'_j := empty U-relation over $U'_j(\overline{T}, \overline{A}, \mathrm{Var}, \mathrm{Rng})$;
    foreach $t \in U$ **do**
      G_i := connected component of $\mathcal{G}$ with id i such that the nodes $t.\mathrm{Var}_1, \ldots, t.\mathrm{Var}_n$ are in G_i;
      $\{c_{i_1}, \ldots, c_{i_k}\} = G_i - \{t.\mathrm{Var}_1, \ldots, t.\mathrm{Var}_n\}$;
      foreach $l_{i_1}, p_{i_1} : (c_{i_1}, l_{i_1}, p_{i_1}) \in W, \ldots, l_{i_k}, p_{i_k} : (c_{i_k}, l_{i_k}, p_{i_k}) \in W$ **do**
        /* Compute a new domain value ($f_{|G_i|}$ is either the identity or better, for atomic l's, an injective function $\mathrm{int}^{|G_i|} \to \mathrm{int}$) */;
        $l := f_{|G_i|}(t.\overline{\mathrm{Rng}}, l_{i_1}, \ldots, l_{i_k})$;
        $U'_j := U'_j \cup \{(t.\overline{T}, t.\overline{A}, G_i, l)\}$;
  $W' := \bigcup_i \{(g_i, (l_1, \ldots, l_m), p) \mid G_i = \{c_1, \ldots, c_m\}$ and
                $(c_1, l_1, p_1), \ldots, (c_m, l_m, p_m) \in W$ and $p := \prod_j p_j\}$;
  Output $(U'_1, \ldots, U'_m, W')$;
end

problem is coNP-hard for a number of representation systems, ranging from attribute-level ones like WSDs to tuple-level ones like ULDBs [11]. In case of tuple-level normalized U-relations, however, we can efficiently compute the certain tuples using relational algebra.

Lemma 6.3. *Tuple $\overline{a}$ is certain in a tuple-level normalized U-relation U iff there exists a variable x such that $(\bar{t}, \overline{a}, x \mapsto l) \in U$ for each domain value l of x and some tuple id $\bar{t}$.*

The condition of the lemma can be encoded as the following domain calculus expression:

$$cert(U) := \{\overline{a} \mid \exists x \forall l \forall p\ (x, l, p) \in W \Rightarrow \exists \bar{t}(\bar{t}, \overline{a}, x, l) \in U\}$$

U	T	A	D_1	D_2
	t_1	a_1	$c_1 \mapsto 1$	$c_1 \mapsto 1$
	t_2	a_2	$c_1 \mapsto 1$	$c_2 \mapsto 2$
	t_2	a_3	$c_1 \mapsto 2$	$c_1 \mapsto 2$
	t_3	a_4	$c_3 \mapsto 1$	$c_3 \mapsto 1$
	t_3	a_5	$c_3 \mapsto 2$	$c_3 \mapsto 2$

W	Var	Rng	Pr
	c_1	1	0.5
	c_1	2	0.5
	c_2	1	0.3
	c_2	2	0.7
	c_3	1	0.6
	c_3	2	0.4

(a) U-relational database

U'	T	A	D
	t_1	a_1	$c_{12} \mapsto (1,1)$
	t_1	a_1	$c_{12} \mapsto (1,2)$
	t_2	a_2	$c_{12} \mapsto (1,2)$
	t_2	a_3	$c_{12} \mapsto (2,1)$
	t_2	a_3	$c_{12} \mapsto (2,2)$
	t_3	a_4	$c_3 \mapsto 1$
	t_3	a_5	$c_3 \mapsto 2$

W'	Var	Rng	Pr
	c_{12}	$(1,1)$	0.15
	c_{12}	$(1,2)$	0.35
	c_{12}	$(2,1)$	0.15
	c_{12}	$(2,2)$	0.35
	c_3	1	
	c_3	2	

(b) Database from (a) normalized

Figure 6.1: Normalization example.

The equivalent relational algebra query on a tuple-level normalized U-relational database $(U[\overline{T_R}, \overline{A}, \text{Var}, \text{Rng}], W)$ is

$$\pi_{\overline{A}}(\pi_{Var}(W) \times \pi_{\overline{A}}(U) - \pi_{Var,\overline{A}}(W \times \pi_{\overline{A}}(U) - \pi_{Var,Rng,\overline{A}}U)).$$

CHAPTER 7

OPTIMIZING QUERIES ON VERTICALLY PARTITIONED U-RELATIONAL DATABASES

U-relational databases rely on vertical partitioning to achieve compact representation of large world-sets. In the realm of complete databases, column stores are known to bring significant advantages when evaluating projection queries. A column store database is called reduced, if there are no dangling tuples, that is, no tuples with missing values for some attributes. Evaluated on a reduced column store database, a query can only look at the partitions for the attributes that are either involved in selection or join conditions, or are projected on in the final result. However, one should proceed with caution when applying such an optimization to query evaluation over probabilistic databases. Consider for example the U-relational database in Figure 7.1 (a), which represents the four worlds in (b). The second world with no tuple for R is obtained under the valuation $x \mapsto 1, z \mapsto 2$. Consider now a query $\pi_B(R)$. If we proceed naively and simply take partition U_B defining attribute B of R as the result, we will not obtain a correct answer, since evaluated on the set of possible worlds the query produces a tuple in each world, except for the second one where no result is returned. Intuitively, there is a dependency between the A and B attributes of R due to the variable z that the two partitions share, which needs to be propagated before partitions are discarded by the query. To correctly evaluate the query in this case we need to first merge the two partitions which will effectively augment the world-set descriptors of each tuple, and then project away

U_A	T_R	A	D_1	D_2
	t_1	1	$x \mapsto 1$	$z \mapsto 1$
	t_1	2	$x \mapsto 2$	

U_B	T_R	B	D_1
	t_1	3	$z \mapsto 1$
	t_1	4	$z \mapsto 2$

(a)

R^1	A	B
t_1	1	3

R^2	A	B

R^3	A	B
t_1	2	3

R^4	A	B
t_1	2	4

(b)

U'_B	T_R	B	D_1	D_2
	t_1	3	$x \mapsto 1$	$z \mapsto 1$
	t_1	4	$x \mapsto 2$	$z \mapsto 1$
	t_1	4	$x \mapsto 2$	$z \mapsto 2$

(c)

Figure 7.1: (a) U-relational database, (b) the set of possible worlds it represents, and (c) result of applying $\pi_B(R)$.

the unnecessary attributes. The result is given in Figure 7.1 (c).

Recall that in the definition of U-relational databases we did not require the attributes of a relation R_i to be defined in non-overlapping partitions $U_{i,j}$. However, for simplicity in the remainder of the section we will assume that each attribute A of a relation R_i is defined in a single partition $U_{i,j}$. All definitions and techniques can be easily generalized to include overlapping partitions.

We extend the definition of a reduced database to U-relational databases.

Definition 7.1. A U-relational database $(U_{1,1}, \ldots, U_{1,m_1}, \ldots, U_{k,1}, \ldots, U_{k,m_k}, W)$ is *reduced*, if for all vertical partitions $U_{i,j}$ containing attributes for some relation R_i, and tuples $(\bar{t}, \overline{b_{i,j}}, \overline{d_{i,j}}) \in U_{i,j}$, the tuple can be completed to a full tuple in some world, that is, there are tuples $(\bar{t}, \overline{b_{i,k}}, \overline{d_{i,k}}) \in U_{i,k}$ such that $\bigcup_k \overline{B_{i,k}} = \overline{A}_i$ and $\bigwedge_k \overline{d_{i,k}}$ is satisfiable.

A U-relational database is non-reduced if either of the following conditions is true:

U_A	T_R	A	D_1
	t_1	1	$x \mapsto 1$
	t_2	2	$y \mapsto 1$

U_B	T_R	B	D_1
	t_1	3	$z \mapsto 1$
	t_1	4	$z \mapsto 2$

(a) Missing values for a tuple id

U_A	T_R	A	D_1	D_2
	t_1	1	$x \mapsto 1$	$y \mapsto 1$
	t_1	2	$x \mapsto 2$	$y \mapsto 2$

U_B	T_R	B	D_1	D_2
	t_1	1	$y \mapsto 1$	$z \mapsto 1$
	t_1	2	$y \mapsto 2$	$z \mapsto 2$

U_C	T_R	C	D_1	D_2
	t_1	1	$x \mapsto 1$	$z \mapsto 2$
	t_1	2	$x \mapsto 2$	$z \mapsto 1$

(b) Invalid combinations

Figure 7.2: Examples of non-reduced U-relational databases.

1. There is a tuple id $\bar{t}$ for some relation R_i, which has values for some attributes but not for all. Formally, there exists a vertical partition $U_{i,j}$ containing attributes for relation R_i, and a tuple $(\bar{t}, \overline{b_{i,j}}, \overline{d_{i,j}}) \in U_{i,j}$ such that for some attribute $A \in \mathbf{sch}(R_i)$ and partition $U_{i,k}$ with $A \in sch(R_i)$, there is no tuple $(\bar{t}, \overline{b_{i,k}}, \overline{d_{i,k}}) \in U_{i,k}$.
2. There is a tuple id $\bar{t}$ for some relation R_i such that no consistent combination of values for $\bar{t}$ can be built. Formally, there is a relation R_i, and a tuple id $\bar{t}$ for R_i such that for all $(\bar{t}, \overline{b_{i,k}}, \overline{d_{i,k}}) \in U_{i,k}$: $\bigwedge_k d_{i,k} \models \mathbf{false}$.

Example 7.2. *Figure 7.2 gives examples of two U-relational databases that are not reduced. The database in (a) defines no B-values for tuple t_2, and in (b) no combination of values for tuple t_1 is consistent.* □

Algorithm 2 removes dangling tuples from a U-relational database by making sure that neither of the two conditions for a non-reduced U-relation database from above is satisfied.

Algorithm 2: reduce

Input: U-relational database U for a world-set over schema Σ
Output: reduced U-relational database U' equivalent to U

1 **foreach** $R_i \in \Sigma$ **do**
2 **foreach** *partition* $U_{i,j}$ *for* R_i *and tuple* $r_i = (\bar{t}, \overline{b_{i,j}}, \overline{d_{i,j}}) \in U_{i,j}$ **do**
3 **if** *not exist tuples* $r_k = (\bar{t}, \overline{b_{i,k}}, \overline{d_{i,k}}) \in U_{i,k}, k \neq j$ *such that* $\overline{d_{i,j}} \wedge \bigwedge_k d_{i,k}$ *is satisfiable* **then**
4 delete r_i from $U_{i,j}$

U_1	T_R	A	D_1
	t_1	1	$(x \mapsto 1, 0.6)$
	t_1	2	$(x \mapsto 2, 0.4)$

U_2	T_R	B	D_1
	t_1	3	$(z \mapsto 1, 0.3)$
	t_1	4	$(z \mapsto 2, 0.2)$

(a)

U_2	T_R	B	D_1
	t_1	3	$(z \mapsto 1, 0.3)$
	t_1	4	$(z \mapsto 2, 0.2)$

(b)

Figure 7.3: (a) U-relational database, and (b) result of applying $\pi_B(R)$.

Non-reduced databases are not only suboptimal in terms of storage, as they keep tuples that can never be instantiated, but can also lead to suboptimal query evaluation plans. Consider for example the projection $\pi_B(R)$ executed on the U-relational database of Figure 7.2 (a). To ensure a correct result, we need to first merge the two vertical partitions for R before projecting away the A column. In this way we will not produce erroneous tuples in the result. However, the reduced property is not always sufficient to allow us ignore vertical partitions for attributes that do not show up in the query. We have already seen one such example in Figure 7.1 - we could not discard the partition U_1 in the query plan for $\pi_B(R)$ since U_1 and U_2 were dependent. However, if we consider the U-relational database in Figure 7.3 and the same projection query, the answer to it can be obtained by simply taking the partition for attribute B since it is independent of the one for A.

To support efficient query evaluation we will take advantage of a different property that can hold on the data, namely independence of attribute values. We are interested in U-relational databases in which projections will need to only merge relevant partitions, that is, partitions defining attributes from the projection set. Formally,

Definition 7.3. Let $U = (U_{1,1}, \ldots, U_{1,m_1}, \ldots, U_{k,1}, \ldots, U_{k,m_k})$ be a U-relational database. U is called *partition-independent*, if for each set of attributes $A \in \mathbf{sch}(R_i)$

$$[\![\pi_A(R_i)]\!] = [\![\pi_A\left(merge\left(U_{i,j_1}, \ldots, U_{i,j_k}\right)\right)]\!], \quad \text{where } A \cap U_{i,j_l} \neq \emptyset$$

It is easy to see that

Proposition 7.4. *For each U-relational database U for a worldset over schema Σ there is a partition-independent U-relational database U' equivalent to U.*

Proof. We construct a tuple-level U-relational database U' from U in the following way: for each relation $R \in \Sigma$ merge all partitions defining attributes of R. Thus U' is partition-independent and equivalent to U. □

Even if we start with a partition-independent U-relational database, queries and updates might destroy this property. We will therefore consider independence properties that hold between partitions, or sets of partitions. Intuitively, two partitions are independent of each other, if projecting on the attributes of one of the partitions need not merge the other partition to compute the result.

Definition 7.5. Let $(U_{1,1}, \ldots, U_{1,m_1}, \ldots, U_{k,1}, \ldots, U_{k,m_k}, W)$ be a reduced U-relational database. For partitions $U_{i,j}, U_{i,l}$ for a relation R_i it holds that $U_{i,j}$ is *independent* of $U_{i,l}$ iff:

$$[\![\pi_{A_j}(R_i)]\!] = [\![\pi_{A_j}\left(merge\left(U_{i,1}, \ldots, U_{i,l-1}, U_{i,l+1} \ldots, U_{i,m_k}\right)\right)]\!] \tag{7.1}$$

In partition-independent relational databases each partition $U_{i,j}$ is independent of all other partitions $U_{i,l}$ for the same relation R_i. Then, as expected, for partition-independent U-relational databases projecting on any set of attributes $A \subseteq A_j$ can be done by a simple projection on the partition $U_{i,j}$:

$$[\![\pi_A(R)]\!] = [\![\pi_A(U_{i,j})]\!] = \pi_{\overline{T},A,\overline{D}}(U_{i,j}), \quad \text{where } \mathbf{sch}(U_{i,j}) = [\overline{T}; A_j; \overline{D}]$$

The independence property is in general not commutative: partition $U_{i,j}$ can be independent of $U_{i,l}$ while $U_{i,l}$ is not independent of $U_{i,j}$. Consider for example relation $R[AB]$ with vertical partitions U_A and U_B defined as follows:

U_A	T_R	A	D_1
	t_1	1	$(x \mapsto 1, 0.6)$

U_B	T_R	B	D_1
	t_1	2	

Partition U_A is independent of U_B, while U_B is not independent of U_A. Indeed, the condition for t_1 says that the tuple only exists in the worlds where $x \mapsto 1$. Thus when evaluating the projection $\pi_B(R)$ we need to merge in U_A to propagate this constraint, while $\pi_A(R)$ is simply the partition U_A iteself.

The above example shows that two partitions can be dependent even if they don't have variables in common, which can be somewhat counterintuitive. Replacing U_B by $\pi_B(\mathit{merge}(U_A, U_B))$ will propagate the condition for t_1 to U_B, making the two partitions mutually independent.

Algorithm 3 normalizes a U-relational database by making all partitions independent of each other. To do so, it merges dependent partitions in order to propagate dependencies from one partition to another.

Algorithm 3: make-independent

Input: U-relational database U for a world-set over schema Σ
Output: Partition-independent U-relational database U' equivalent to U

```
1 foreach R_i ∈ Σ do
2     foreach partitions U_{i,j}, U_{i,k} for R_i such that U_{i,j} - dependent on U_{i,k} do
3         replace U_{i,j} by π_{B_{i,j}}(merge(U_{i,j}, U_{i,k}));
```

We use the dependence relationship to build a graph G_U[1] whose nodes are the partitions of the U-relational database and there is a link between two nodes if the first node is dependent on the second one. We will call the connected components in this graph *dependency clusters*. We will assume that a partition is always dependent on itself, so the dependency graph will contain self loops at each node. For a U-relational database $U = (U_{1,1}, \ldots, U_{1,m_1}, \ldots, U_{k,1}, \ldots, U_{k,m_k}, W)$, we will denote by $\{K_{i,m_i}\}_i$ the set of its dependency clusters. If U is a partition-independent U-relational database, then $K_{i,j} = \{U_{i,j}\}$. For an attribute A we will denote by A_{G_U} the set of partitions reachable from A's partition in G_U:

$$A_{G_U} = \{U_{k,j} \mid U_{k,j} \text{ reachable from } U_{i,j} \text{ in } G_U, U_{i,j} \text{ partition for } A\}$$

We will reuse this definition for a set of attributes S, thus

$$S_{G_U} = \bigcup_{A \in S} A_{G_U}$$

Intuitively, if we want to project on an attribute A of a relation, we need to merge the partitions reachable from A in the dependency graph, or in general, when evaluating a query, we need to merge the partitions for those attributes reachable from the attributes required by the query. This will be discussed in detail in the following section.

[1] To be more precise, G_U is in general a forest, with one graph (or subforest) for each relation R of the database schema.

We will further expand the definition of independence to consider *horizontal slices* of the partitions. A horizontal slice in partition $U_{i,j}$ for tuple id t is the set of all tuples in $U_{i,j}$ that have tuple id t. We will mark this slice by $U^t_{i,j}$. The motivation for this is that two partitions can be dependent only for some of the tuple ids and independent for the others. We will modify definition 7.5 to allow independence between horizontal slices of two partitions:

Definition 7.6. Let $(U_{1,1}, \ldots, U_{1,m_1}, \ldots, U_{k,1}, \ldots, U_{k,m_k}, W)$ be a reduced U-relational database. For partitions $U_{i,j}, U_{i,l}$ for a relation R_i it holds that $U_{i,j}$ is *independent* of $U_{i,l}$ with respect to tuple id t iff:

$$[\![\pi_{A_j}(\sigma_{T=t}R_i)]\!] = [\![\pi_{A_j}(\sigma_{T=t}(merge(U_{i,1}, \ldots, U_{i,l-1}, U_{i,l+1} \ldots, U_{i,m_k})))]\!]$$

Algorithm 4: make-independent

Input: U-relational database U for a world-set over schema Σ
Output: U-relational database U' equivalent to U where all partitions are independent

1 **foreach** $R_i \in \Sigma$ **do**
2 **foreach** *tuple id t and partitions* $U_{i,j}, U_{i,k}$ *for* R_i *such that* $U_{i,j}$ *dependent on* $U_{i,k}$ *- dependent wrt to the t-slice* **do**
3 replace the t-slice in $U_{i,j}$ by $\pi_{\overline{B_{i,j}}}\left(merge\left(\sigma_{T_{R_i}=t}(U_{i,j}), \sigma_{T_{R_i}=t}(U_{i,k})\right)\right)$;

Algorithm 3 can be easily modified to take advantage of slicing, the modified version is given in Algorithm 4.

Using the dependency definition for horizontal slices of a partition, we can construct a dependency graph G_U whose nodes are horizontal slices $U^t_{i,j}$ and there is an edge from $U^t_{i,j}$ to $U^t_{i,k}$ if $U_{i,j}$ is dependent on $U_{i,k}$ with respect to the t horizontal slice. For a tuple id t and an attribute A, $A_{G_U,t}$ will denote the set of partitions in G_U reachable from the t slice of the partition for A. Similarly, we can define the set of partitions $S_{G_U,t}$ for a set of attributes S.

7.1 Queries on reduced U-relations

Consider a U-relational database U with dependency clusters $K_1, \ldots, K_s$. The baseline for our query evaluation operators are the ones defined in Chapter 5. In short, the implementation of the operations undoes the vertical partitioning using the `merge` operation, and adds additional conditions to join queries that ensure consistency of the world-set descriptors. In this section we are interested in cases where we can avoid merging in partitions for attributes that are not needed for evaluating the query, and methods for keeping the result of a query in partitioned form.

Example 7.7. Consider relation $R_i[A, B, C]$ and a query $\pi_A(\sigma_{B=1}(R))$. Suppose R_i is represented by three partitions U_A, U_B, U_C. In the default implementation the query will be evaluated as

$$\pi_A(\sigma_{B=1}(\texttt{merge}(\texttt{merge}(U_A, U_B), U_C)))$$

If both U_A and U_B are independent of U_C, we do not need to merge U_C in when evaluating the query, thus we can use the following query plan that will produce equivalent results:

$$\pi_A(\sigma_{B=1}(\texttt{merge}(U_A, U_B)))$$

However, if U_C is reachable from either U_A or U_B in the respective dependency clusters we need to merge it in in order to propagate the dependencies before we can project out column C. □

We will consider two modes of evaluating queries on vertically partitioned U-relational databases:

- Queries produce tuple-level representations

- Queries produce vertically partitioned representations

In the first mode queries on vertically partitioned U-relations will result in a tuple-level U-relation. This mode will typically be applied to cases where the query is shown directly to the user, or used in database programs for iterating over query results. The second mode of operation can be used in cases where query results are saved to disk for later use, for example with the "create table" statement. To demonstrate the second case consider the selection query $S := \sigma_{A=1}(R)$ expressed with the following SQL statement:

```
create table S as
select * from R
where A = 1;
```

Suppose R has schema $R[A, B, C]$ and partitions $U_{R,A}, U_{R,B}, U_{R,C}$ for each of the attributes, respectively. If we evaluate the query according to the first mode, we will produce a relation S represented by a single partition $U_{S,ABC}$ that can be computed with the query

$$\sigma_{A=1}(merge(U_{R,A}, U_{R,B}, U_{R,C}))$$

Trivially, the result will be a partition-independent U-relation.

If we follow the second evaluation strategy we can produce three partitions for S: $U_{S,A}, U_{S,B}, U_{S,C}$ by applying the selection on the $U_{R,A}$ partition of R to obtain $U_{S,A}$, and computing $U_{S,B}, U_{S,C}$ from $U_{R,B}, U_{R,C}$, respectively in a way that will ensure that S is reduced. Such processing can naturally destroy the independence between partitions, unless additional measures are taken to guarantee it.

$$
\begin{aligned}
[\![\mathit{possible}(Q_1), S]\!] &:= [\![Q_1, S]\!] \\
[\![\pi_{\overline{X}}(Q_1), S]\!] &:= [\![\pi_{\overline{X}}(Q_1), S \cap X]\!] \\
[\![\sigma_{\phi}(Q_1), S]\!] &:= [\![Q_1, S \cup \overline{A}_1]\!], \qquad \text{where } \phi \text{ on } \overline{A}_1 \\
[\![Q_1 \bowtie_{\phi} Q_2, S]\!] &:= [\![Q_1, S \cup \overline{A}]\!] \bowtie_{\phi} [\![Q_2, S \cup \overline{A}]\!] \\
&\qquad\qquad \text{where } \phi \text{ on } \overline{A} \\
[\![R_i, S]\!] &:= R_i^{S \cap \mathbf{sch}(R_i)}
\end{aligned}
$$

Figure 7.4: Annotating relation names in a query with sets of attributes.

Next, we discuss how queries affect vertically partitioned databases, and present an algorithm for maintaining partitions and dependency information. We will consider as input a pair (U, G_U) of a vertically partitioned U-relational database U and its dependency graph G_U, as defined above, and query Q on U. The result of querying will be a pair $(U', G_{U'})$, where U' is the U-relational database obtained from U by adding the result of Q, and $G_{U'}$ is the updated graph containing dependencies for the result of the query Q.

First we define a simple procedure, that will take a relational algebra query Q as input, and will annotate the relation names appearing in Q with the sets of attributes needed for that relation. The result of this procedure will be a query Q' in which each relation name R_i is replaced by the symbol $R_i^{S_i}$ where S_i is the set of attributes of R_i that are needed in Q. Figure 7.4 shows a top-down rewriting scheme implementing the annotation procedure. Without loss of generality we assume that attribute names are unique across relations. For a query Q, the annotate rewriting scheme is seeded with $S = \bigcup_{R_i \text{ in } Q} \mathbf{sch}(R_i)$, the set of all attributes of the relations mentioned in Q.

Given input query Q annotated with the attributes needed from each relation, and a dependency graph G_U for the U-relational database, we modify the rewriting of the base case for relation R of Figure 5.3 and Equation (5.1) in the following way:

$$[\![R^S]\!] = merge(U_1, \ldots, U_l), \text{ where } U_i \in S_{G_U} \tag{7.2}$$

In the above rewriting relation R is replaced by the merge of all partitions reachable from partitions for attributes needed by the query Q.

Theorem 7.8. *Let Q be a query involving relation R_i, $A_1, \ldots, A_n$ be the attributes of R_i involved in Q, and G_U be the dependency graph for the U-relational database where Q is being evaluated. Let $\hat{Q}$ be the implementation of Q according to Chapter 5, that merges all partitions for R_i. Let $\hat{Q}'$ be the query obtained from $\hat{Q}$ using the rewriting of Chapter 5 with Rule* (7.2) *replacing Rule* (5.1)*. Then $\hat{Q} \cong \hat{Q}'$.*

The theorem states that we only need to merge in those partitions that are required by the query, or when a partition required by the query depends on them.

Consider for example the U-relational database for relation $R(A, B)$ in Figure 7.5 (a). Partitions U_1 and U_2 are independent with respect to tuples t_1 and t_3 but are dependent for the t_2-slice. The result of evaluating $\pi_B(R)$ is shown in Figure 7.5 (b), and can be obtained by the query plan

$$\sigma_{T_R \neq t_2}(U_2) \cup \pi_B(\texttt{merge}(\sigma_{T_R = t_2} U_1, \sigma_{T_R = t_2} U_2))$$

The slicing of vertical partitioning can be exploited in queries where tuples are discarded due to a selection or join condition. For example, the selection in the query $\pi_B(\sigma_{B \neq 8 \wedge B \neq 9}(R))$ will discard tuple t_2 in Figure 7.5 and will eliminate

U_1	T_R	A	D_1
	t_1	1	$(x \mapsto 1, 0.6)$
	t_1	2	$(x \mapsto 2, 0.4)$
	t_2	3	$(y \mapsto 1, 0.6)$
	t_3	4	$(u \mapsto 1, 0.5)$
	t_3	5	$(u \mapsto 2, 0.5)$

U_2	T_R	B	D_1
	t_1	6	$(z \mapsto 1, 0.3)$
	t_1	7	$(z \mapsto 2, 0.2)$
	t_2	8	$(v \mapsto 1, 0.2)$
	t_2	9	$(v \mapsto 2, 0.2)$
	t_3	10	$(w \mapsto 1, 0.2)$
	t_3	11	$(w \mapsto 2, 0.8)$

(a)

U_2	T_R	B	D_1	D_2
	t_1	6	$(z \mapsto 1, 0.3)$	
	t_1	7	$(z \mapsto 2, 0.2)$	
	t_2	8	$(v \mapsto 1, 0.2)$	$(y \mapsto 1, 0.6)$
	t_2	9	$(v \mapsto 2, 0.2)$	$(y \mapsto 1, 0.6)$
	t_3	10	$(w \mapsto 1, 0.2)$	
	t_3	11	$(w \mapsto 2, 0.8)$	

(b)

Figure 7.5: (a) U-relational database, and (b) result of applying $\pi_B(R)$.

the need to merge in partition U_1, since the t_1 and t_3 slices of U_2 are independent of the corresponding slices in U_1.

For query Q, U-relational database U and dependency graph G_U on the horizontal slices in U, the rewrite rule for a relation R looks like this:

$$[\![R^S]\!] = \bigcup_{t-\text{ tuple id in } R} \{merge(\sigma_{T_R=t}U_1, \ldots, \sigma_{T_R=t}U_l) \mid U_i^t \in S_{G_U,t}\} \tag{7.3}$$

Queries producing vertical partitions

Here we investigate evaluating queries of the form `create table S as Q;` that save the result of querying to disk, and are thus natural candidates for producing results in partitioned form. The idea is that we want to execute the query on each partition in parallel and update the dependency graph accord-

ingly with new dependencies introduced by the query operators.

Before we present the algorithm for query processing, let us consider a selection $\sigma_{A=c}(R)$, and let $U_{i,j}$ be the partition containing $R.A$. To compute the result we need to apply the condition on that partition, i.e. $U'_{i,j} = \sigma_{A=c}U_{i,j}$. To compute the partitions for the remaining attributes we have several choices. Let $U_{i,k}[\overline{T}_R, \overline{B_{i,k}}, \overline{D}_{i,k}]$ be a partition for R, $k \neq j$. To compute $U'_{i,k}$ we can do one of the following two things:

1. $U'_{i,k} = \pi_{\overline{B}_{i,k}}(merge(U'_{i,j}, U_{i,k}))$, that is, we resolve the dependency between the resulting partitions $U'_{i,j}$ and $U'_{i,k}$ by doing a merge. The resulting dependency graph G'_U will have no edge $(U'_{i,k}, U_{i,j})$' even if the input graph had an edge $(U_{i,k}, U_{i,j})$. Note that this is precisely the effect of applying `make-independent` on the two partitions.
2. $U'_{i,k} = U_{i,k} \ltimes_{T_R} U'_{i,j}$, that is, we filter out the tuples from $U_{i,k}$ that were filtered by the selection and will not appear in the result in any world, but we do not resolve any dependency from $U'_{i,j}$ to $U'_{i,k}$ that may have been introduced by the selection. In the resulting graph we therefore add an edge $(U'_{i,k}, U'_{i,j})$. Note that we will not add an edge $(U'_{i,j}, U'_{i,k})$ unless it was already in: applying a filter on $U_{i,j}$ can make the remaining partitions dependent on it (since alternatives for tuples may be dropped from the partition), but cannot make $U_{i,j}$ dependent on the other partitions (unless it was already dependent).

We will consider the effect of each query operator on the partitions and the dependency graph, assuming the result of each operation is a relation S. For selections and joins we use Rule (2) from above, but we can apply any combina-

tion of the two rules for computing the partitions and the resulting dependency graph.

1. **Base case:** $Q = R$

   In the base case the partitions of the input relation, together with their dependencies are copied over to the resulting relation:

   - Partitions: for each partition $U_{i,j}$ for R create a partition $U'_{i,j}$ for the result of Q, such that $U'_{i,j} = \rho_{T_R \mapsto T_S}(U_{i,j})$
   - Dependency graph: extend $G_U(V, E)$ to graph $G'_U(V', E')$ such that:

     $$V' = V \cup \{U'_{i,j} \mid U_{i,j} - \text{ partition for } R\}$$

     $$E' = E \cup \{(U'_{i,j}, U'_{i,k}) \mid U_{i,j}, U_{i,k} - \text{ partitions for } R, (U'_{i,j}, U_{i,k}) \in E\}$$

2. **Selection with constant:** $Q = \sigma_{A=c}(Q_1)$

   In the selection case we apply the selection condition on the partition containing the selection attribute, and copy over the rest of the partition using a semijoin on the tuple id to filter out tuples that are not in the result for any of the worlds.

   - Partitions: let $U_{i,j}$ be the partition for Q_1 containing A, and let R be the result of Q_1. Create partition $U'_{i,j}$ for the result of Q, such that $U'_{i,j} = \rho_{T_R \mapsto T_S}(\sigma_{A=c} U_{i,j})$. For each other partition $U_{i,k}$ for Q_1 create a partition $U'_{i,k}$ for the result of Q, such that $U'_{i,k} = \rho_{T_R \mapsto T_S}(U_{i,j}) \ltimes_{T_S} U'_{i,j}$.
   - Dependency graph: extend $G_U(V, E)$ to graph $G'_U(V', E')$ such that:

     $$V' = V \cup \{U'_{i,j} \mid U_{i,j} - \text{ partition for } R\}$$

     $$E' = E \cup \{(U'_{i,j}, U'_{i,k}) \mid U_{i,j}, U_{i,k} - \text{ partitions for } R, (U_{i,j}, U_{i,k}) \in E\}$$

     $$\cup \{(U'_{i,k}, U'_{i,j}) \mid U_{i,j} - \text{ partition for } A \text{ in } R, U_{i,k} - \text{ partition for } R\}$$

Join selection: $Q = \sigma_{A=B}(Q_1)$

Join selections are processed in the same way as selections with a constant with the only difference that if A and B are defined in separate partitions, we need to merge those to evaluate the condition.

- Partitions: let $U_{i,j}, U_{i,k}$ be the partition for Q_1 containing A and B, respectively, and let R be the result of Q_1. If $U_{i,j} = U_{i,k}$, let $U_{i,l} = U_{i,j}$, otherwise let $U_{i,l} = merge(U_{i,j}, U_{i,k})$. Create partition $U'_{i,l}$ for the result of Q, such that $U'_{i,l} = \rho_{T_R \mapsto T_S}(\sigma_{A=B} U_{i,l})$. For each other partition $U_{i,m} (m \neq j, m \neq k)$ for Q_1 create a partition $U'_{i,m}$ for the result of Q, such that $U'_{i,m} = \rho_{T_R \mapsto T_S}(U_{i,m}) \ltimes_{T_S} U'_{i,l}$.
- Dependency graph: extend $G_U(V, E)$ to graph $G'_U(V', E')$ such that:

$$
\begin{aligned}
V' &= V \cup \{U'_{i,l}\} \cup \{U'_{i,m} \mid U_{i,m} - \text{ partition for } R, m \neq j, m \neq k\} \\
E' &= E \cup \\
&\quad \{(U'_{i,m}, U'_{i,n}) \mid U_{i,m}, U_{i,n} - \text{ partitions for } R, (U_{i,m}, U_{i,n}) \in E, \\
&\quad \{m, n\} \cap \{j, k\} = \emptyset\} \cup \\
&\quad \{(U'_{i,m}, U'_{i,l}) \mid U'_{i,l} - \text{ partition for } A, B \text{ in } S, U'_{i,m} - \text{ partition for } S\}
\end{aligned}
$$

Projection: $Q = \pi_{\overline{B}}(Q_1)$

When we project away attributes we need to make sure dependencies are propagated to partitions for attributes in the projection set.

- Partitions: let $U_{i,j}[\overline{T}_R, \overline{B_{i,j}}, \overline{D}_{i,j}]$ be a partition for Q_1 containing $A \in \overline{B}$, and let R be the result of Q_1. Create partition $U'_{i,j} = \rho_{T_R \mapsto T_S}(\pi_{\overline{B_{i,k}} \cap \overline{B}}(U_{i,j}))$. For each partition $U_{i,k}[\overline{T}_R, \overline{B_{i,k}}, \overline{D}_{i,k}]$ for Q_1 such that $\overline{B_{i,k}} \cap \overline{B} = \emptyset$ find a partition $U_{i,j}$ defining $A \in \overline{B}$ such that $U_{i,k}$ is reachable from $U_{i,j}$. If

such $U_{i,j}$ exists, propagate the dependency from $U_{i,k}$ to $U'_{i,j}$: $U'_{i,j} = \pi_{\overline{B_{i,k}}}(\rho_{T_R \mapsto T_S}(merge(U_{i,j}, U_{i,k})))$.

- Dependency graph: extend $G_U(V, E)$ to graph $G'_U(V', E')$ such that:

$$
\begin{aligned}
V' &= V \cup \{U'_{i,j} \mid U_{i,j} - \text{ partition for attribute } A \in \overline{B} \text{ for } R\} \\
E' &= E \cup \\
&\quad \{(U'_{i,j}, U'_{i,k}) \mid U_{i,j}, U_{i,k} - \text{ partitions for attributes in } \overline{B} \text{ for } R, \\
&\quad ((U_{i,j}, U_{i,k}) \in E, \\
&\quad \text{ or there is partition } U_{i,l} \text{ whose dependencies were} \\
&\quad \text{propagated to } U_{i,k} \text{ and } (U_{i,j}, U_{i,l}) \in E)\}
\end{aligned}
$$

CHAPTER 8

PROBABILISTIC U-RELATIONS

The U-relational databases defined in Chapter 4 can elegantly model probabilistic information by adding storing the probability distribution of the variables in the world table W. Recall that W contains tuples (x, v, p) for all domain values v of a variable x, and p is the probability of $x \mapsto v$. For each variable x defined by W, the sum of the values $\pi_{Pr}(\sigma_{\text{Var}=x})(W)$ must equal one.

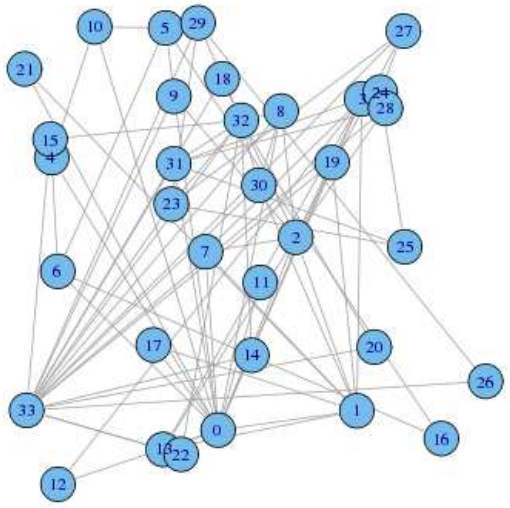
We use a function P to define the probability of a valuation as the product of probabilities of its variable assignments:

$$P(\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}) = \prod_{i=1}^{n} P(\{x_i \mapsto v_i\}) \qquad (*)$$

The probabilistic extension is orthogonal to the techniques for evaluating positive relational algebra queries described in Chapter 5. Since processing relational algebra queries only extends each world with the result of the query in it without changing the world's probabilities, the algorithms carry over with no change to the probabilistic case as well. A different class of queries are those that ask for confidence of tuples in the result of a query. Let U be a U-relation representing the answer to a query q on a U-relational database. Then, the *confidence* of a tuple $\overline{a}$ in the answer to q is the sum of the probabilities of the worlds defined by U that contain $\overline{a}$. Computing the confidence by enumerating all possible worlds, as the above definition suggested, is, however, not feasible. A better approach is to compute the probability of the world-set represented by the union of ws-descriptors associated with $\overline{a}$ in U:

$$P(\{\overline{d} \mid \exists \overline{s}(\overline{d}, \overline{s}, \overline{a}) \in U\})$$

In case only one tuple with ws-descriptor $\overline{d}$ in U matches the given tuple $\overline{a}$, then



(a) Network of friendships in Zachary's karate club

Edge	T_E	u	v	D
	t_1	27	34	$(x_1 \mapsto 1, 0.12)$
	t_2	27	30	$(x_2 \mapsto 1, 0.12)$
	t_3	27	30	$(x_3 \mapsto 1, 0.15)$
	t_4	27	8	$(x_4 \mapsto 1, 0.20)$
	t_5	30	29	$(x_5 \mapsto 1, 0.89)$
	t_6	8	29	$(x_6 \mapsto 1, 0.05)$
	...			

NoEdge	T_N	u	v	D
	s_1	27	34	$(x_1 \mapsto 2, 0.88$
	s_2	27	30	$(x_2 \mapsto 2, 0.88)$
	s_3	27	30	$(x_3 \mapsto 2, 0.85)$
	s_4	27	8	$(x_4 \mapsto 2, 0.80)$
	s_5	30	29	$(x_5 \mapsto 2, 0.11)$
	s_6	8	29	$(x_6 \mapsto 2, 0.95)$
	...			

(b) Representing the network as a probabilistic U-relational database

U	u	w	v	D_1	D_2
	27	30	29	$(x_3 \mapsto 1, 0.15)$	$(x_5 \mapsto 1, 0.89)$
	27	8	29	$(x_4 \mapsto 1, 0.20)$	$(x_6 \mapsto 1, 0.0.5)$

(c) Possible connections of length no more than 2 between nodes 27 and 29

Figure 8.1: Modeling social networks as probabilistic U-relational databases.

the confidence of $\overline{a}$ can be trivially computed as $P(\overline{d})$ using formula (*) above. In the general case, however, the computation is #P-complete. This follows from the mutual reducibility of the problem of computing the probability of the union of the (possibly overlapping) world-sets represented by a set of ws-descriptors and of the #P-complete problem of counting the number of satisfying assignments of Boolean formulas in disjunctive normal form (DNF). Indeed, we can encode a set of k ws-descriptors $\{x_1^i \mapsto v_1^i, \ldots, x_{m_i}^i \mapsto v_{m_i}^i\}$ $(1 \leq i \leq k)$ as a formula $\bigvee_{1 \leq i \leq k} (x_1^i = v_1^i \wedge \ldots \wedge x_{m_i}^i = v_{m_i}^i)$.

Example 8.1. Figure 8.1 (a) shows a graph of friendship relationships in Zachary's karate club dataset [69]. Nodes are members of the club, and links denotes that the two members are friends. Suppose friendship is not absolute, thus each link is quantified with the likelihood that the two people are friends. Figure 8.1 (b) shows a U-relational representation of (part of) the graph. We use two relations Edge and NoEdge to describe edges which are in the database and those that are not. Note that we use the same variable to make sure that the existence of an edge is mutually exclusive from its non-existence.

Given this representation, we can compute the likelihood that nodes 27 and 29 are within distance two in the network, that is, they are either direct friends, or share a friend in common. This query can be expressed as a self join on the Edge relation. Assuming the U-relational database of Figure 8.1 is complete, the possible paths between 27 and 29 of length no more than two are given in U-relational form in subfigure (c) of that Figure. The probability p that the two nodes are connected in the required way can be computed as the probability of the formula $p = Pr(x_3 \mapsto 1 \wedge x_5 \mapsto 1 \vee x_4 \mapsto 1 \wedge x_6 \mapsto 1)$. Since the two clauses of the DNF are independent (they share no variable in common), we compute the

confidence value as

$$\begin{aligned} p &= 1 - (1 - Pr(x_3 \mapsto 1 \wedge x_5 \mapsto 1))(1 - Pr(x_4 \mapsto 1 \wedge x_6 \mapsto 1)) \\ &= 1 - (1 - 0.1335)(1 - 0.01) = 0.142165 \end{aligned}$$

□

We next describe an approximation algorithm that computes the probability of a DNF. The algorithm is an extension of the SPROUT techniques from [39], also described in Section 3 for compiling a DNF formula into a so-called *ws-tree*. Recall the definition from [39]: a ws-tree is a tree whose internal nodes are either $\oplus$ or $\otimes$, variable elimination and independence decomposition, respectively, and whose leaves are the vacuously true ws-descriptor $\mathbf{t}$. Edges are labeled with variable assignments from the world-table W such that the following constraints hold: (1) a variable can appear at most once on a path from the root to the leaf, (2) the outgoing edges from a $\oplus$ node are labeled with different assignment of the same variable, and (3) variables in the different subtrees rooted at the same $\otimes$ node are disjoint. Given a ws-tree T for a DNF Φ, the probability of Φ can be computed using one traversal of T in the following way:

- If T is a leaf node $\{\mathbf{t}\}$, then $P(T) = 1$.
- If the root of T is $\otimes$ with subtrees T_1, T_2 with probabilities p_1, p_2, respectively, then $P(T) = 1 - (1 - p_1)(1 - p_2)$.
- If the root of T is $\oplus$ with edges $x \mapsto i, i \in [1, k]$, subtrees $T_1, \ldots, T_k$ and probabilities $p_1, \ldots, p_k$, respectively, then $P(T) = \sum_i p_i$.

Our approach to approximating confidence values is to partially materialize ws-trees[1]. Informally, a partial ws-tree is one where the leaf nodes do not necessarily contain the constant **t**, but can contain the subset of clauses from the given DNF consistent with the assignments on the respective path. Given a partial ws-tree we can give a confidence interval to each node of the tree, such that the real probability of the subtree rooted at that node lies in that interval. The idea will then be to expand a partial ws-tree until the confidence interval becomes tight enough, as specified by a user-defined error bound. The probability of a partial ws-tree can be bound in the following way. Let $C = \{c_1, \ldots, c_n\}$ be a leaf node in a partial tree. Then the probability f of C lies in the interval $[f_{min}, f_{max}]$ where $f_{min} = \min_i P(c_i)$, $f_{max} = \min(1, \sum_i P(c_i))$. Then the probability of the tree is a function of $P(f_1), \ldots, P(f_n)$, where $f_1, \ldots, f_n$ are the leaves of the tree, and lies in the interval $[p_{min}, p_{max}]$ where p_{min} is obtained under $P(f_i) = f_i^{min}$, and p_{max} is obtained under $P(f_i) = f_i^{max}$. It is easy to see that the confidence intervals are shrinking with the expansion of a partial ws-tree. Moreover, if S, T are two partial ws-trees such that T is an expansion of S, and $[s_{min}, s_{max}], [t_{min}, t_{max}]$ are the corresponding confidence intervals, then the confidence intervals are nested, that is $s_{min} \leq t_{min} \leq t_{max} \leq s_{max}$.

Given a partial ws-tree T, the error bounds for T can be computed according to the rules for computing the probability of a wsd-tree:

- If T is a leaf node with a set of clauses C, then

$$[t_{min}, t_{max}] = [\max_{c \in C} P(c), \min(1, \sum_{c \in C} P(c))]$$

[1]The techniques in this chapter were developed in parallel and are close to the spirit of [47]. The difference, as will become clear later our approach builds approximate ws-trees in a bottom-up fashion, by stitching partially constructed ws-trees to form bigger ones, as opposed to the aforementioned work where leaves of a partial ws-tree are expanded one at a time until the desired bound is reached.

- If the root of T is $\otimes$ with subtrees T_1, T_2 and error bounds $[t^1_{min}, t^1_{max}], [t^2_{min}, t^2_{max}]$, respectively, then the error bound for T is:

$$[t_{min}, t_{max}] = [1 - (1 - t^1_{min})(1 - t^2_{min}), 1 - (1 - t^1_{max})(1 - t^2_{max})]$$

- If the root of T is $\oplus$ with edges $x \mapsto i$, subtrees $T_1, \ldots, T_k$ and error bounds $[t^i_{min}, t^i_{max}]$, respectively, then the error bound for T is:

$$[t_{min}, t_{max}] = [\sum_i P(x \mapsto i) \cdot t^i_{min}, \sum_i P(x \mapsto i) \cdot t^i_{max}]$$

Example 8.2. Consider the partial ws-tree in Figure 8.2 which has two unexpanded leaf nodes. We can bound the probability of the tree in a bottom up fashion, starting at the leaves and propagating the bounds further up in the tree. For example, the probability of the leaf node $S\{\{y \mapsto 1\}, \{z \mapsto 1\}\}$ can be bound as

$$s_{min} = P(y \mapsto 1) = 0.2, s_{max} = P(y \mapsto 1) + P(z \mapsto 1) = 0.6$$

For the probability bounds of the parent node we obtain

$$t_{min} = .1 + .4 \cdot s_{min} = .18, t_{max} = .1 + .4 \cdot s_{max} = .34$$

In the end we obtain as confidence interval for the tree [.426,0.769]. If the user is willing to accept an error bigger than 0.343, then we can stop and return any number from the confidence interval as result. Otherwise we can continue expanding the ws-tree until the size of the confidence interval is below the user specified error. □

We want to design an algorithm that quickly finds a "good" partial ws-tree, i.e. one with small confidence interval. A simple idea would be to construct the tree step by step, where at each step we decide to expand one leaf node of

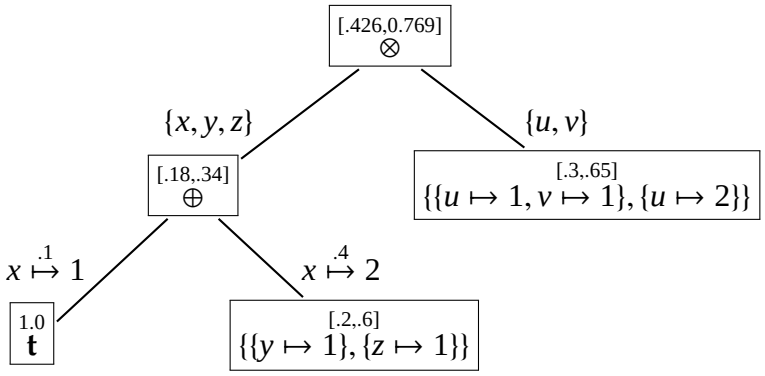
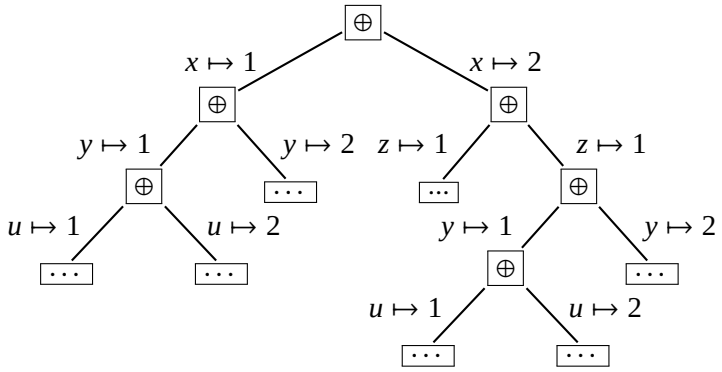
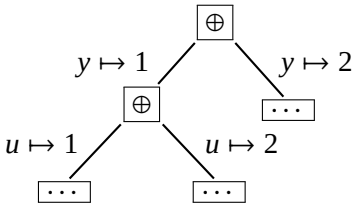


Figure 8.2: Bounding the confidence interval of a partial ws-tree.



(a)



(b)

Figure 8.3: Partial ws-tree (a) and repeated fragment in it (b).

the partial ws-tree until we obtain a ws-tree satisfying the error requirements. However, such a 1-step expansion can lead to suboptimal performance, as we might be forced to perform the same expansion step more than once in different branches of a partial ws-tree.

Algorithm 5: Approx-confidence

Input: $D = \{C_1, ..., C_m\}$: set of clauses, W: probability distribution for the variables in D, ϵ: acceptable error

Output: p: probability that is within ϵ from $P(D)$

```
// Initialization
```

1 For each variable x_i in D, construct tree T_i: $root(T_i) = \oplus$, and for each domain value j of x_i there is an edge from x_i with label $x_i \mapsto j$, to the set of clauses from D consistent with $x_i \mapsto j$;

2 Let S be the set of trees T_i;

3 For each i compute $(min_i, max_i) = \text{er}(T_i)$;

```
// Expansion
```

4 **while** *true* **do**

5 Choose trees T_i, T_j from S and path K from T_i such that $can\_combine(T_i, T_j, K)$;

6 Let $T_k = combine(T_i, T_j, K)$;

7 $T_k = expand\_ind(T_k)$;

8 $(min_k, max_k) = \text{er}(T_k)$; `// conf bounds for new tree`

`// Stopping condition`

9 **if** $(max_k - min_k) \leq \epsilon$ **then return** $P(T_k)$ **else**

10 Add T to S;

Example 8.3. Consider for example the partial ws-tree in Figure 8.3 (a). Note that both the left and the right subtree below the root contain the tree shown in Figure 8.3 (b). □

With this motivation we will design a dynamic programming approach where we compute parts of the partial ws-tree (also called shrubs) that can later be stitched together to form new partial ws-trees.

Algorithm 4 shows the skeleton of constructing partial ws-trees for approximating confidence. The algorithm maintains a set of partially completed trees S. Initially S contains one tree representing the possible assignments for a single variable x_i (lines 1-3). For example Figure 8.4 shows the initial step of the algorithm given binary variables x, y, z, u, v. It then recursively proceeds to compute

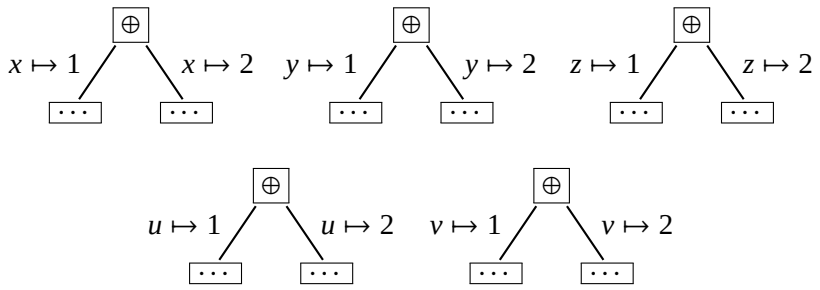


Figure 8.4: Initial set of ws-trees for binary variables x, y, z, u, v.

new partial ws-trees by combining pairs of trees from the tree pool, and computes their confidence bounds (lines 6-8). The algorithm also checks whether the leaves of newly computed tree can be decomposed using independence decomposition. If the confidence bound for the new tree is smaller than the given threshold, the algorithm returns the probability of the new tree as result, otherwise the combination step is repeated with two new trees from the pool. The function *can_combine*() in line 5 checks whether condition (1) of the definition of ws-trees is satisfied, namely that a variable can appear at most once on each root-to-leaf path. Algorithm 6 implements this function. For efficiency, we can keep a bitvector for each tree and path denoting which variables appear in that tree or path, respectively.

Algorithm 6: can_combine(T, T', K)

Input: T, T': partial ws-trees, K: path in T
Output: true, iff T' can be stitched with T via K

```
1 Let X_K be the set of variables appearing on path K;
2 Let X' be the set of variables appearing on edges in T';
3 if X_K ∩ X' = ∅ then
4     return true
5 return false
```

We next discuss in more detail how to choose and combine partial ws-trees from the pool, and how to recompute the confidence interval for newly created

trees obtained by combining existing trees.

We will first consider how to construct trees that only contain the mutex combination node $\oplus$. Consider first a complete ws-tree T which has only has $\oplus$ nodes, and which consists of a set of paths $K_1, \ldots, K_n$. Since the paths are mutually exclusive, the probability of T is simply the sum of the probabilities of the paths it contains. Formally, let each path K_i be a sequence of variable assignments $x_{i_1} \mapsto j_1, \ldots, x_{i_k} \mapsto j_k$. Then

$$P(T) = \sum_i P(K_i) = \sum_i P(x_{i_1} \mapsto j_1) \cdot \ldots \cdot P(x_{i_k} \mapsto j_k)$$

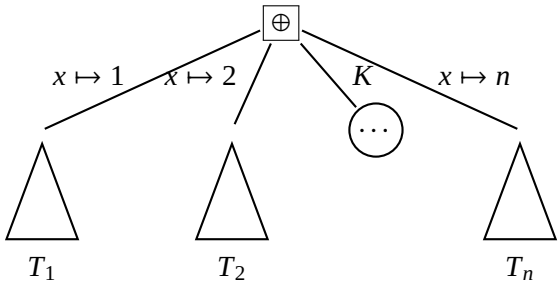
Let now $T = \{K_1, \ldots, K_n\}$ be a partial ws-tree, let C_i be the set of clauses associated with K_i, and let C_i has confidence interval $[min_i, max_i]$. Then the confidence interval for T is simply

$$[t_{min}, t_{max}] = \left[\sum_i P(K_i) \cdot min_i, \sum_i P(K_i) \cdot max_i \right]$$

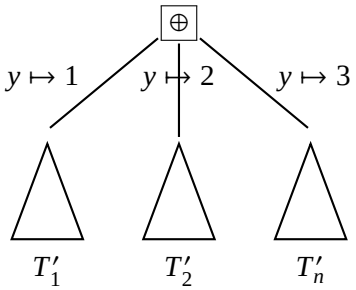
Each path K_i in T contributes $P(K_i) \cdot (max_i - min_i)$ to error for the error probability of T. We will denote by $\delta_i = max_i - min_i$ the length of the confidence interval for the set of clauses C_i. Then for the length δ_T of the confidence interval of T we get

$$\delta_T = \sum_i P(K_i) \cdot \delta_i \tag{8.1}$$

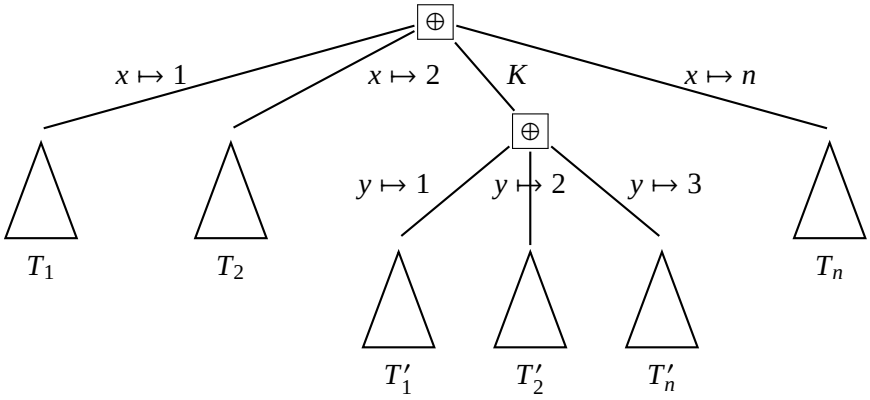
Suppose now T is a tree that has relatively tight confidence interval and K is a path in T that still has uncertainty. If $P(K)$ is relatively high, then cutting down the uncertainty in the leaf of K will have big impact on the error bounds for T. Suppose now T' is another tree that has relatively tight confidence interval, and suppose T' has no overlap with the variables on K. Then we can expand K by



(a) Tree T and path K in it.



(b) Tree T'.



(c) Tree resulting from stitching T' at the end of K in T.

Figure 8.5: Stitching partial ws-trees.

stitching T' to T by coping it at the end of K. This step is graphically depicted in Figure 8.5.

Algorithm 7 describes the combination step for stitching two partial ws-trees T and T' via path K in T. The new leaves of T are computed by intersecting the set of clauses of T' with the clause set at K: these are the clauses that are consistent with both the path K and the respective path from T'. If no clauses remain consistent after the intersection, we can remove that path from the resulting tree.

To recompute the error bound of the new tree (assuming there are only mutex nodes in it), we do the following. Let δ_T be the original error bound of tree T, δ be the original error bound for the leaf at path K, $P(K)$ be the probability of

Algorithm 7: *combine*(T, T', K)

Input: T, T': partial ws-trees, K: path in T
Output: T^*: partial ws-trees obtained by stitching T' at the end of K in T

```
1 Let C be the clause set at the end of K;
2 foreach path K' in T' do
3     Let C' be the clause set at the end of K';
4     Replace C' by C ∩ C', where the intersection is done based on the
      original ids of the clauses, and the resulting clause set is simplified by
      removing the variable assignments from K;
5     if C' = ∅ then Remove K' from T';
6 Replace C in T by T';
```

the path K and δ' be the modified probability of tree T' after it has been stitched to K. Then the new error bound δ'_T for the modified tree T can be computed as:

$$\delta'_T = \delta_T + P(K)(\delta' - \delta) \tag{8.2}$$

Stitching two ws-trees together when those contain also independence decomposition nodes $\otimes$ can be done in the same way as for the restricted case of $\oplus$-only ws-trees using Algorithm 7. Recomputing the bounds however is slightly more complicated than in the simple case, as the probability is no longer purely additive. We will therefore store at each node the error bounds for the subtree rooted at that node. After stitching two trees together via a path, we will recompute the intermediate probabilities of all nodes on that path, where the error bounds at the root node will be the desired new error bounds of the modified tree. Note that we do not to recompute error bounds in other subtrees.

CHAPTER 9

EXPERIMENTS

Prototype Implementation. We implemented the query translator of Fig. 5.3. We also extended the C implementation of the TPC-H population generator version 2.6 build 1 [65] to generate attribute and tuple-level U-relations and ULDBs. The code is available on the MayBMS project page (http://www.cs.cornell.edu/database/maybms).

Setup. The experiments were performed on a 3GHZ/1GB Pentium running Linux 2.6.13 and PostgreSQL 8.2.3.

Generation of uncertain data. The following parameters were used to tune the generation: *scale* (s), *uncertainty ratio* (x), *correlation ratio* (z), and *maximum alternatives per field* (m). The (dbgen standard) parameter s is used to control the size of each world; x controls the percentage of (uncertain) fields with several possible values, and m controls how many possible values can be assigned to a field. The parameter z defines a Zipf distribution for the variables with different dependent field counts (DFC). The DFC of a variable is the number of tuple fields dependent on that variable. We use the parameter z to control the attribute correlations: For n uncertain fields, there are $\lceil C * z^i \rceil$ variables with DFC i, where $C = n(z-1)/(z^{k+1}-1)$, i.e., $n = \sum_{i=0}^{k}(C * z^i)$. Thus greater z values correspond to higher correlations in the data. The number of domain values of a variable with DFC $k > 1$ is chosen using the formula $p^{k-1} * \prod_{i=1}^{k}(m_i)$, where m_i is the number of different values for the i-th field dependent on that variable and p is the probability that a combination of possible values for the k fields is valid. This assumption fits naturally to data cleaning scenarios. Previous work [8] shows that chasing dependencies on WSDs enforces correlations between field values

Q_1: **possible** (**select** o.orderkey, o.orderdate, o.shippriority **from** customer c, orders o, lineitem l
where c.mktsegment = 'BUILDING'
and c.custkey = o.custkey and o.orderkey = l.orderkey
and o.orderdate > '1995-03-15' **and** l.shipdate < '1995-03-17')

Q_2: **possible** (**select** extendedprice **from** lineitem **where**
shipdate **between** '1994-01-01' **and** '1996-01-01'
and discount **between** '0.05' **and** '0.08' **and** quantity < 24)

Q_3: **possible** (**select** n1.name, n2.name **from** supplier s, lineitem l,
orders o, customer c, nation n1, nation n2 **where** n2.nation='IRAQ'
and n1.nation='GERMANY' **and** c.nationkey = n2.nationkey
and s.suppkey = l.suppkey **and** o.orderkey = l.orderkey
and c.custkey = o.custkey **and** s.nationkey = n1.nationkey)

Figure 9.1: Queries used in the experiments.

s	z	TPC-H dbsize	#worlds	Rng	dbsize	#worlds	Rng	dbsize	#worlds	Rng	dbsize
0.01	0.1	17	$10^{857.076}$	21	82	$10^{7955.30}$	57	85	$10^{79354.1}$	57	114
0.01	0.5	17	$10^{523.031}$	71	82	$10^{4724.56}$	901	88	$10^{46675.6}$	662	139
0.05	0.1	85	$10^{4287.23}$	22	389	$10^{39913.8}$	33	403	10^{396137}	65	547
0.05	0.5	85	$10^{2549.14}$	178	390	$10^{23515.5}$	449	416	10^{232650}	1155	672
0.10	0.1	170	$10^{8606.77}$	27	773	$10^{79889.9}$	49	802	10^{793611}	53	1090
0.10	0.5	170	$10^{5044.65}$	181	776	$10^{46901.8}$	773	826	10^{466038}	924	1339
0.50	0.1	853	$10^{43368.0}$	49	3843	10^{400185}	71	3987	$10^{3.97e+06}$	85	5427
0.50	0.5	853	$10^{25528.9}$	214	3856	10^{234840}	1832	4012	$10^{2.33e+06}$	2586	6682
1.00	0.1	1706	$10^{87203.0}$	57	7683	10^{800997}	99	7971	$10^{7.94e+06}$	113	11264
1.00	0.5	1706	$10^{51290.9}$	993	7712	10^{470401}	1675	8228	$10^{4.66e+06}$	3392	13312
		x = 0.0	**x** = 0.001			**x** = 0.01			**x** = 0.1		

Figure 9.2: Total number of worlds, max. number of domain values for a variable (Rng), and size in MB of the U-relational database for each of our settings.

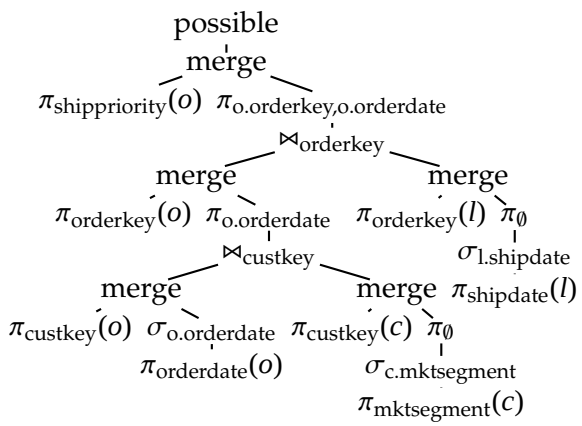


Figure 9.3: Query plan for Q_1 using merge.

and removes combinations that violate the dependencies. We considered here that after correlating two variables with arbitrary DFCs, only $p * 100$ percent of the combinations satisfy the constraints and are preserved.

The uncertain fields are assigned randomly to variables. This can lead to correlations between fields belonging to different tuples or even to different relations. This fits to scenarios where constraints are enforced across tuples or relations. We do not assume any kind of independence of our initial data as done in several other approaches [24, 15].

For the experiments, we fixed p to 0.25, m to 8, and varied the remaining parameters as follows: s ranges over $(0.01, 0.05, 0.1, 0.5, 1)$, z ranges over $(0.1, 0.25, 0.5)$, and x ranges over $(0.001, 0.01, 0.1)$.

An important property of our generator is that any world in a U-relational database shares the properties of the one-world database generated by the original dbgen: the sizes of relations are the same and the join selectivities are approximately equal. We checked this by randomly choosing one world of the U-relational database and comparing the selectivities of joins on the keys of the

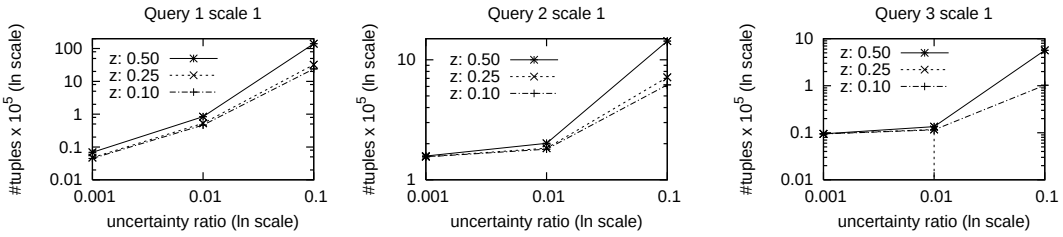


Figure 9.4: Sizes of query answers for settings with scale 1.

TPC-H relations for different scale factors and uncertainty ratios.

Queries. We used the three queries from Fig. 9.1. Query Q_1 is a join of three relations of large sizes. Query Q_2 is a select-project query on the relation lineitem (the largest in our settings). Query Q_3 is a fairly complex query that involves joins between six relations. All queries use the operator 'possible' to retrieve the set of matches across all worlds. Note that these queries are modified versions of Q_3, Q_6, and Q_7 of TPC-H where all aggregations are dropped (dealing with aggregation is subject to future work).

Fig. 9.4 shows that our queries are moderately selective and their answer sizes increase with uncertainty x and marginally with correlation z. For scale 1, the answer sizes range from tens of thousands to tens of millions of tuples. There is only one setting ($z = 0.25$ and $x = 0.1$) where one of our queries, Q_3, has an empty answer. Before the execution, the queries were optimized using our U-relation-aware optimizations. Fig. 9.3 shows Q_1 after optimizations.

Characteristics of U-relations. Following Fig. 9.2, the U-relational databases are exponentially more succinct than databases representing all worlds individually: while the number of worlds increases exponentially (when varying the uncertainty ratio x), the database size increases only linearly. The case of $x = 0$ corresponds to one world generated using the original dbgen. Interestingly, to

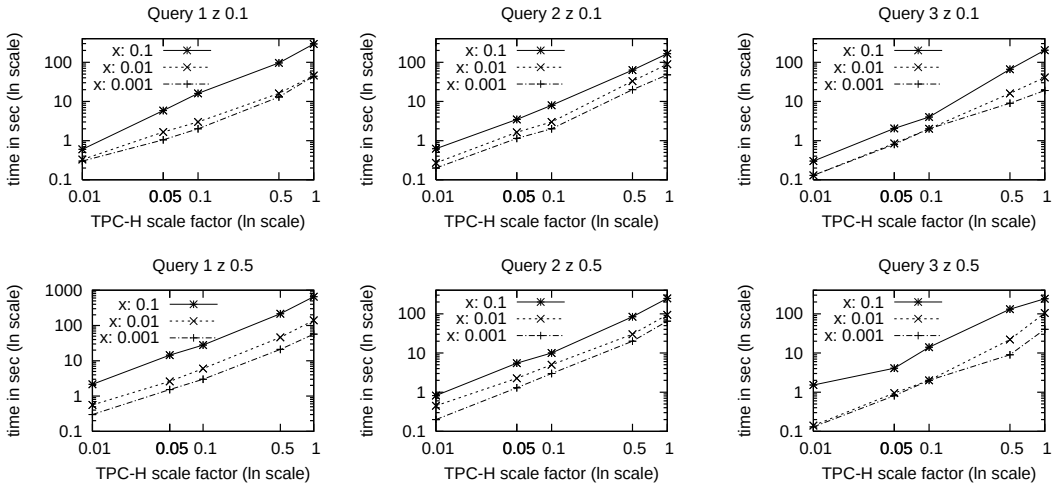


Figure 9.5: Performance of query evaluation for various scale, uncertainty, and correlation.

represent $10^{8 \cdot 10^6}$ worlds, the U-relational database needs about 6.7 times the size of one world.

An increase of the scaling factor leads to an exponential increase in the number of worlds and only to a linear increase in the size of the U-relational database. Although we only report here on experiments with scale factors up to 1, further experiments confirmed that similar characteristics are obtained for larger scales, too. An increase of the correlation parameter leads to a moderate relative increase in the database size. When compared to one-world databases, the sizes of U-relational databases have increase factors that vary from 6.2 (for $z = 0.1$) to 8.2 (for $z = 0.5$).

Query Evaluation on U-relations. We run four times our set of three queries on the 45 different datasets reported in Fig. 9.2. For each query and correlation ratio, Fig. 9.5 has a log-log scale diagram showing the median evaluation (including storage) time in seconds as a function of the scale and uncertainty parameters ([5] also shows diagrams for $z = 0.25$). The different lines in each of

the diagrams correspond to different uncertainty ratios.

Fig. 9.5 shows that the evaluation of our queries is efficient and scalable. In our largest scenario, where the database has size 13 GB and represents $10^{8 \cdot 10^6}$ worlds with 1.4 GBs each world, query Q_3 involving five joins is evaluated in less than two and a half minutes. One explanation for the good performance is the use of attribute-level representation. This allows to first compute the joins locally using only the join attributes and later merge in the remaining attributes of interest. Another important reason for the efficiency is that due to the simplicity of our rewritings, PostgreSQL optimizes the queries in a fairly good way. ([5] shows an optimized query plan produced by the PostgreSQL 'explain' statement for the rewriting of Q_2.)

The evaluation time varies linearly with all of our parameters. For Q_1 (Q_2 and Q_3 respectively) we witnessed a factor of up to 6 (4 and 10 respectively) in the evaluation time when varying the uncertainty ratio from 0.001 to 0.1. When the correlation ratio is varied from 0.1 to 0.5, the evaluation time increases by a factor of up to 3; this is also explained by the increase in the input and answer sizes, cf. Fig.s 9.2 and 9.4. When the scale parameter is varied from 0.01 to 1, the evaluation time increases by a factor of up to 400; in case of Q_3 and $z = 0.5$, we also noticed some outliers where the increase factor is around 1000.

Effect of attribute-level representation. We also performed query evaluation on tuple-level U-relations, which represent the same world-set as the attribute-level U-relations of Fig. 9.2, and on Trio's ULDBs [15] obtained by a (rather direct) mapping from the tuple-level U-relations. To date, Trio has no native support for the possible operator or the removal of erroneous tuples in the query answer, though this effect can be obtained as part of the confidence computa-

tion[1]. For that reason, we decided to compare the evaluation times of queries without the possible operator and without the (expensive) removal of erroneous tuples or confidence computation (which is an exponential-time problem). Since our data exhibits a high degree of (randomly generated) dependency, its ULDB representation has lineage and thus join queries can introduce erroneous tuples in the answer. The Trio prototype was set to use the (faster) SPI interface of PostgreSQL (and not its default python implementation).

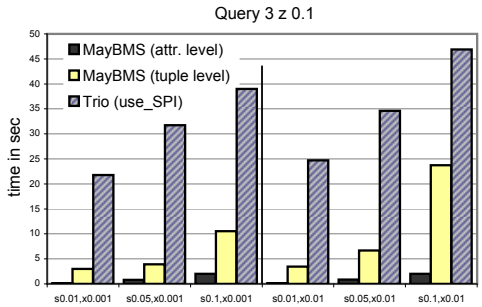


Figure 9.6: Querying attribute-level and tuple-level U-relations in MayBMS and ULDBs in Trio.

Fig. 9.6 compares the evaluation time on attribute- and tuple-level U-relations in MayBMS, and ULDBs for small scenarios of 1% uncertainty, our lowest correlation factor 0.1, and scale up to 0.1. On attribute-level U-relations, the queries perform several times better than on tuple-level U-relations and by an order of magnitude better than ULDBs. This is because attribute-level data allows for late materialization: selections and joins can be performed locally and tuple reconstruction is done only for successful tuples. We witnessed that an increase in any of our parameters would create prohibitively large (exponential in the arity) tuple-level representations. For example, for scale 0.01 and uncertainty 10%, relation lineitem contains more than 15M tuples compared to 80K in each of its vertical partitions.

[1]Personal communication with the TRIO team as of June 2007.

Part III

APIs for probabilistic databases

CHAPTER 10

DATABASE PROGRAMMING

A database programming model enables the development of applications that access and manipulate data stored in a database from a high-level programming language. A database program connects to a DBMS and can execute update commands, or issue queries and obtain cursors to iterate over the result. APIs provide means of accessing the data without knowing how the data is stored on disk, and often allow for porting programs between different database management systems. We next propose and study the properties of a programming model for uncertain data.

10.1 Queries and updates on uncertain DBMSs

We consider queries and updates specified using the SQL select, insert, update and delete statements. We take the possible worlds semantics to define the meaning of queries and updates. A query, applied on a set of possible worlds extends each world with the result of the query in that world. Similarly, an update operation is executed on each world of the world-set. For querying we will also consider the operator $conf$ for computing the confidence of the possible tuples in the result of a query. The confidence of a tuple t is defined as the sum of the probabilities of all worlds containing t. We also consider two special cases of this operator: *possible*, which computes all possible tuples (tuples with confidence greater than 0), and *certain*, which returns the tuples appearing in all possible worlds (i.e., confidence = 1), see e.g. [9].

There have been a number of studies on how to evaluate queries on uncer-

U_1	*TID*	A	D_1
	t_1	1	$(x_1 \mapsto 1, 0.2)$
	t_1	2	$(x_1 \mapsto 2, 0.8)$
	t_2	3	$(x_2 \mapsto 1, 0.6)$
	t_2	4	$(x_2 \mapsto 2, 0.4)$

U_2	*TID*	B	D_1
	t_1	1	$(y_1 \mapsto 1, 0.1)$
	t_1	2	$(y_1 \mapsto 2, 0.9)$
	t_2	2	$(y_2 \mapsto 1, 1)$

(a)

U_1	*TID*	A	D_1	D_2
	t_1	8	$(x_1 \mapsto 1, 0.2)$	$(y_1 \mapsto 1, 0.1)$
	t_1	1	$(x_1 \mapsto 1, 0.2)$	$(y_1 \mapsto 2, 0.9)$
	t_1	8	$(x_1 \mapsto 2, 0.8)$	$(y_1 \mapsto 1, 0.1)$
	t_1	2	$(x_1 \mapsto 2, 0.8)$	$(y_1 \mapsto 2, 0.9)$
	t_2	3	$(x_2 \mapsto 1, 0.6)$	
	t_2	4	$(x_2 \mapsto 2, 0.4)$	

(b)

Figure 10.1: Updating uncertain databases: (a) U-relational database; (b) relation U_1 after applying the update of Example 10.1.

tain databases by translating them into queries on the representation, see e.g. [34, 14, 6]. None of the present works however has considered the problem of applying updates on uncertain databases. As with querying, we would like to execute updates on the representation rather than iterate over the possible worlds.

Most systems for managing uncertain data use a compact representation for storing large sets of worlds. This usually means that tuple or attribute values that appear in several worlds are stored only once, with additional constraints that describe to which worlds they belong. Correlations are represented by means of lineage in Trio, using world-set descriptors in MayBMS, and with graphical models in [55].

We will use as running example the U-relational database of Figure 10.1 (a) representing a relation $R(A, B)$ in an uncertain database. It consists of two vertical partitions $U_1[TID; A; D_1]$ and $U_2[TID; B; D_1)$ containing the possible values for the A and the B attribute of R, respectively. The column D_1 in the two relations is used to specify correlations of the possible values. For example $t_1.A$ has a value of 1 whenever the variable x_1 is mapped to 1 (which happens with

probability 0.2), and with probability 0.8 has value 2 whenever $x_1 \mapsto 2$. In this way the U-relational database represents compactly eight possible worlds, one for each of the possible combinations of values for the variables x_1, x_2, y_1, y_2.

Example 10.1. Let T1 be an operation that updates R:

```
T1: update R set A = 8 where B = 1;
```

This operation will update tuple t_1 for the worlds where $B = 1$. These worlds are constructed by taking y_1 to be 1. However, the current representation does not capture the desired correlation; therefore we need to create two copies of each of the alternatives of tuple t_1 in U_1 for the cases where it has to be updated or not. To compute the result we have to undo part of the decomposition, as shown in Figure 10.1 (b). □

Intuitively, each update operation consists of two steps: in the first one we create copies of those tuples that will need to be updated in some worlds only, and in the second step we execute the update. The first step only changes the representation, but not the world-set itself. We only decompress when it is necessary – when in some world the attribute value needs to be updated, and we do not merge in tuples that will not be updated in any world.

10.2 Programming interface

The programming language we will use is summarized in Figure 10.2. As in the classical setting of certain databases, a database program is a sequence of

construct	meaning
update statements	Execution of SQL create table, insert, update and delete queries. The query statements can be constructed using constants, values read from the database or user-supplied values.
read($\$x$,$\$x_{in}$);	Read user input into variable $\$x_{in}$. The user is displayed a prompt $\$x$.
write($x);	User output operation. The program can output messages to the user, including values stored in tuple variables.
+,-,*,/	Arithmetic operations on variables and constants.
for($t in Q){P}	Iterate over the result of a query Q and execute the nested program P for each binding of the tuple variable $t. The query language we consider is an extension of SQL with the keywords **possible** and **certain**, that compute the tuples appearing in some, or all worlds, respectively, and the construct **conf** returning the confidence of a tuple in the result of a query.

Figure 10.2: Language constructs for database programming.

statements that can execute select and update commands on the database, iterate over query results and provide user interaction through read and write commands.

How are programs executed on an uncertain database? We strive for a model for database programs for uncertain databases that satisfies the following desiderata. *First*, the execution model should be independent of the underlying uncertain database management system. This is important as it will allow porting of programs between different uncertain DBMSs. *Second*, programs should allow for efficient execution. *Third*, despite the fact that data is uncertain, programs should have intuitive user interfaces and should not expect users of the program to be aware of the uncertainty.

To satisfy the first requirement, we naturally adopt the possible worlds semantics that has been the standard semantics used to define the meaning of queries on uncertain databases. According to this, the program is executed in

all worlds in parallel; within a world it behaves in the same way as on a complete database; all updates the program makes are applied to the current world. Of course, a direct implementation of this semantics is unrealistic as there are far too many worlds that can be represented by an uncertain database. In the context of querying several works have studied [34, 6, 14] how to avoid iterating over the possible worlds and evaluate a query directly on the representation. While for queries specified in a relational query language it is often possible to find an efficient translation of the query into one on the representation, a database programming language provides richer capabilities, such as executing updates, branching and user interaction that complicate the situation. In the next sections we study the implications of the requirements specified above. We will study criteria for programs to have an observationally deterministic behavior and will provide algorithms for optimizing database programs for set-based execution.

CHAPTER 11

OPTIMIZING DATABASE PROGRAMS

The programming model of Chapter 10 is very powerful and allows for the writing of interesting but also potentially infeasible programs that access the database. We defined the semantics of a database program on uncertain databases to be the one where the program is executed on all worlds in parallel. A direct implementation of this semantics is not possible as uncertain DMBSs often represent compactly a large number of possible worlds. We would therefore like to execute programs in bulk on all worlds at the same time.

Chapter 5 has studied how to translate queries on world-sets into ones on the representation, and in Chapter 10 we have seen how to do this for updates as well. A database program has richer constructs such as branching and loops and it is not clear how to map those to operations on the representation, as the flow of execution can be different on each world of the world-set. For that we will study techniques for unnesting programs, i.e. mapping programs to a sequence of read and write operations with no branching and loops. Another major issue is that a database program allows users to interact with the database via the read() and write() commands. This can cause problems whenever the user is returned output that is not the same in all worlds or when user is asked to supply different input in each world. Given that uncertain DBMSs often store an exponential number of worlds, such behavior is clearly unacceptable. We will therefore require that the uncertain database be observationally indistinguishable from a complete database (i.e. single-world database). We word this property *observational determinism*: a program is called *observationally deterministic* if the user interaction in terms of input and output of the program in one

world is identical to the user interaction in all other worlds of the world-set.

We will first treat the problems of checking for observational determinism and of unnesting updates in isolation. We will start by discussing a method for *statically* checking whether a given program is observationally deterministic using a technique called *c-indicators*. We will then present rules that map update programs to linear sequences of update statements. The techniques from Sections 11.2 and 11.1 will be then used as building blocks of an algorithm that optimizes a database program containing both user interaction and updates.

11.1 Checking observational determinism

A program P is not observationally deterministic whenever it involves different user interaction in each world of the world-set. Let $x()$ be a user interaction operation that appears in P and let Q be the query that binds the values for $x()$. Then we can reason about whether $x()$ is the same in all worlds by checking whether the query Q produces only certain results. We illustrate the idea with the following examples.

R	A	B	C
t_1	{1,2,3}	{4,5}	6
t_2	7	8	9
t_3	10	{11,12,13}	14

Figure 11.1: Or-set relation

Example 11.1. Consider for example an uncertain database that stores data as the *or-set relation* of Figure 11.1. Some fields of the table contain sets, the semantics being that one of the values in the set is the correct one for the respective field. In our example the table represents 3 * 2 * 3 = 18 possible worlds over schema $R(A, B, C)$, one for each combination of values in the or-sets. Consider

the following three programs that run on R[1]:

```
P1: for($t in "select * from R") write($t);
P2: for($t in "select possible * from R") write($t);
P3: for($t in select certain * from R where A=1
        union select * from R where A <> 1")
      if($t.A=1) write($t);
```

When executing $P1$ we run into the problem that the output is not the same in all worlds. On the other hand $P2$ is observationally deterministic, as in each world it outputs the possible tuples, i.e. the tuples that appear in at least one world. For our example the program will print 10 tuples in total: the different versions of t_1, t_2 and t_3. Deciding whether a program is observationally deterministic is not always simple. Consider $P3$: If we trace the origin of the tuples that are output by this program, we will see that these are exactly the certain tuples with A-value 1 in R. Thus the program satisfies observational determinism. □

We next present our algorithm for deciding whether a program is observationally deterministic. We first construct a query corresponding to each user interaction (UI) operation in the program, and we couple this with a procedure for deciding whether a query produces only certain results.

Let us suppose that we have annotated the input database and we know which tuples are certain and which are not. More precisely, for each input relation R, we think of R as consisting of two disjoint partitions $R = R^c \cup R^u$, where R^c contains the certain tuples of R, and R^u: the tuples that appear only in

[1]While the programs are somewhat artificial, they are simple enough and help demonstrate the challenges when deciding whether a program is observationally deterministic.

$$
\begin{aligned}
&\text{Let } R - \text{relation name}, \phi - \text{boolean condition} \\
&Q, Q_1, Q_2 - \text{queries in } RA^+ \cup \{\text{conf,possible,certain}\} \\
&[\![R]\!] := (R^c, R^u) \\
&[\![\pi_U(Q)]\!] := (\pi_U([\![Q]\!]^c), \pi_U([\![Q]\!]^u)) \\
&[\![\sigma_\phi(Q)]\!] := (\sigma_\phi([\![Q]\!]^c), \sigma_\phi([\![Q]\!]^u)) \\
&[\![Q_1 \bowtie_\phi Q_2]\!] := ([\![Q_1]\!]^c \bowtie_\phi [\![Q_2]\!]^c, [\![Q_1]\!]^u \bowtie_\phi [\![Q_2]\!]^u \cup [\![Q_1]\!]^c \bowtie_\phi [\![Q_2]\!]^u \cup [\![Q_1]\!]^c \bowtie_\phi [\![Q_2]\!]^u) \\
&[\![Q_1 \cup Q_2]\!] := ([\![Q_1]\!]^c \cup [\![Q_2]\!]^c, [\![Q_1]\!]^u \cup [\![Q_2]\!]^u) \\
&[\![\text{conf}(Q)]\!] := ([\![Q]\!]^c \cup \text{possible}([\![Q]\!]^u), \emptyset) \\
&[\![\text{possible}(Q)]\!] := ([\![Q]\!]^c \cup \text{possible}([\![Q]\!]^u), \emptyset) \\
&[\![\text{certain}(Q)]\!] := ([\![Q]\!]^c, \emptyset)
\end{aligned}
$$

Figure 11.2: Propagation of certainty during querying and update operations.

some worlds. R^c and R^u can be computed with the queries $R^c = \text{certain}(R)$ and $R^u = R - \text{certain}(R)$. According to our semantics both queries R^c and R^u produce one result for each world, where the result of R^c is the same in all worlds, and the one of R^u is potentially different.

It is interesting to see how the annotations propagate from the input into the results of querying. For example a selection applied on a relation R can only discard tuples, but cannot make any uncertain tuples certain, and vice versa: no certain tuple will become uncertain. A 'possible' query will make all tuples from the input certain in the result since every world will contain those tuples.

Figure 11.2 defines an operator $[\![\cdot]\!]$ that takes a query expressed in positive relational algebra extended with the confidence computation predicate and its two special cases: possible and certain, and returns a pair of queries (Q^c, Q^u), whose components compute the certain and the uncertain tuples in the result, respectively. This construction is conservative in the sense that it produces correct results but can omit some on particular inputs. This is the case for queries

containing either a projection or a union operation. For example if we apply the projection $\pi_C(R)$ on the world-set represented as the or-set relation of Figure 11.1, all tuples in the result are certain although the only certain tuple in the input was t_2. Nevertheless, we shall see that for performing static checks, no other construction will produce better results and will be correct on all inputs.

Example 11.2. The query $Q = \sigma_{A=1}(\text{certain}(\sigma_{A=1}(R)) \cup \sigma_{A\neq 1}(R))$ corresponds to the write statement of $P3$ of Example 11.1. Applying the construction of Figure 11.2 yields the pair of queries $(\sigma_{A=1}(R^c), \sigma_{A=1 \wedge A\neq 1}(R^u))$. Independent of the actual instance of the database, we can conclude that $P3$ is observationally deterministic, as the query defining the uncertain partition of the result has an unsatisfiable selection condition and will always return the empty set as result. □

Using these ideas we can construct a procedure, called *c-indicator*, that "certifies" tuples that are certain in the result of a query.

Ideally we would like to design c-indicators that do not require evaluation of the whole query and then checking which tuples are certain in the output, but instead try to predict this information based on the query and possibly on some constraints that hold on the data. We can measure the quality of a c-indicator based on two criteria. The first one asks for soundness of the c-indicator, that is, that no false positives are produced. The second condition requires that the c-indicator is as close as possible in predicting which tuples are certain in the output. We formalize these requirements below.

Definition 11.3. We say that a c-indicator C is *sound* if for all queries Q, databases $\mathbf{A}$ and tuples t, if $C(Q)(t, \mathbf{A}) = \mathit{true}$, then t is certain in $Q(\mathbf{A})$. A c-indicator C *dominates* another c-indicator C' ($C \supset C'$) iff for all tuples t, queries Q and databases $\mathbf{A}$, if $C'(Q)(t, \mathbf{A}) = \mathit{true}$, then $C(Q)(t, \mathbf{A}) = \mathit{true}$, and there is a tuple t such that

$C(Q)(t, \mathbf{A}) = true$ and $C'(Q)(t, \mathbf{A}) = false$. This means that C identifies strictly more tuples as being certain than C'. A c-indicator C is *maximal* iff there is no other c-indicator C' for the same query Q such that $C' \supset C$.

There are two obvious solutions to the problem of constructing a c-indicator which is sound. The first one is a procedure that rejects all tuples and is therefore trivially guaranteed to produce no false positives. The second way is to enclose the input query Q in a 'certain' construct and check whether the condition $t \in \text{certain}(Q)$ is satisfied. This c-indicator will not only be sound but maximal as well. However, deciding tuple Q-certainty, that is, whether a tuple is certain in the result of a query is coNP-hard on succinct representation systems [34]. Succinct representation systems are such that can represent an exponentially large, or even infinite set of worlds. For example, the tuple-independent model of [24] can represent 2^n possible worlds using n tuples only. Ideally we would like to have c-indicators that use relational algebra only. We will relax this condition to allow querying certain tuples in the input relations. Intuitively, we can annotate the certain tuples once at the beginning, and then incrementally maintain the annotations when the database is updated.

Theorem 11.4. *For each query Q expressed in positive relational algebra with possible and certain there exists a c-indicator using relational algebra operators only which is sound and maximal when we assume no knowledge about the data.*

Proof sketch. Using the construction $[\![\cdot]\!]$ of Figure 11.2, we define a c-indicator C^{pr} (pr stands for propagating uncertainty, the idea used in defining $[\![\cdot]\!]$) as the following test: $C^{pr}(Q)(t, \mathbf{A}) := \{t \in [\![Q]\!]^c(\mathbf{A})\}$.

C^{pr} is sound and maximal. Let $\mathbf{A}$ be an uncertain database, Q be a query ex-

Algorithm 8: Checking observational determinism

Input: P: program
Output: true or false

```
1 foreach UI operation x() in P do
2     Q_x be the query corresponding to x();
3     if [[Q_x]]^u is satisfiable then
4         return false;
5 return true;
```

pressed in positive relational algebra with possible and certain over the schema of $\mathbf{A}$ and t be a tuple in the schema of Q. By induction on the structure of the query Q we show that if $t \in [\![Q]\!]^c(\mathbf{A})$, then t is certain in $Q(\mathbf{A})$. On the other hand, if $t \notin [\![Q]\!]^c(\mathbf{A})$, then there is a witness world-set $\mathbf{B}$ such that t is not possible in $Q(\mathbf{B})$. If Q is a positive relational algebra query, we can take as witness the world-set $\mathbf{B}_0$ containing one world where all relations are empty. □

Finally, using the idea of c-indicators we construct a procedure that automatically decides whether a given program is observationally deterministic or not on all inputs. For each UI operation $x()$ of the given program P we compute the query Q_x that corresponds to $x()$ by composing the for-loop statements that are ancestors of $x()$ in the parse tree of P. Algorithm 5 rewrites Q_x using the c-indicator rules of Figure 11.2 and checks whether the uncertain partition of that query is satisfiable.

Theorem 11.5. *Algorithm 5 rejects all programs that are not observationally deterministic.*

```
read("Enter license plate, location and color:",
     $x,$loc,$col);
if (select * from cars where num=$x != NULL) {
  if (select * from cars where num=$x and loc=$loc
      and color=$col != NULL)
    update cars set wit=wit+1
    where num=$x and loc=$loc and color=$col;
  else insert into cars values($x,$loc,$col,1);
}
else insert into cars values ($x,$loc,$col,1);
```

Figure 11.3: Modification of the program of Figure 1.9 for adding new evidence to a police database.

11.2 Unnesting updates

We next study how an update program can be turned into a sequence of update statements without for-loops or if-conditions.

Example 11.6. Consider a modification of the program from Chapter 1 that satisfies observational determinism in Figure 11.3.

We can linearize the program by mapping the if-else block to the following three update statements:

```
update cars set wit=wit+1
where num=$x and loc=$loc and color=$col;
insert into cars select $x,$loc,$col,1
where not exists (select * from cars where num=$x
                        and loc=$loc and color=$col)
      and exists (select * from cars where num=$x);
insert into cars select $x,$loc,$col,1
where not exists (select * from cars where num=$x);
```

This program is equivalent to the first one. Using the techniques of Chapter 10 we can translate it into a program on the representation, which then executes on all worlds at the same time. □

As seen in the above example we can often push if-conditions into the where-clause of an update operation. For this to work however we need to restrict the update such that it does not interfere with subsequent operations necessary to evaluate a query. For example if we exchange the first and the second update statement of the second program we will obtain a different result, although exchanging the if and the else block of the first program does not change its semantics.

We next formalize rules for unnesting update programs. We consider a somewhat simplified version of the API from Chapter 10, where the control structures are only for-loops. Moreover, for the time being, we restrict ourselves to updates of the following kind. Let R be a relation, $\{A_1, \ldots, A_m\} \subseteq \mathbf{sch}(R)$, $c_1, \ldots, c_n$ be constants and ϕ be a condition involving constants and attributes of R. We will restrict ourselves to updates that add tuples of constant values, or change tuples fields to constant values. The updates are thus of the following form:

update R set $\overline{A} = \overline{c}$ where ϕ;

where $\overline{A} = \overline{c}$ is a shortcut for $A_1 = c_1, \ldots, A_m = c_m$.

Let Q_ϕ denote the semi-join query returning the tuples of R that need to be updated. Moreover, suppose that $\mathbf{sch}(R) = \{A_1, \ldots, A_m, B_1, \ldots B_n\}$, which we abbreviate as $\overline{A}, \overline{B}$. Then the relation R' resulting from executing U on R can be

Let Q be a query and ϕ a condition involving constants and attributes of R.
Moreover, let $\phi(t)$ be obtained from ϕ by replacing all occurrences of $\$t$ by t.

for($\$t$ in Q){update R set $\overline{A} = \overline{c}$ where ϕ;}
$\vdash$ update R set $\overline{A} = \overline{c}$ where ϕ'; with $\phi' = \bigvee_{t \in Q} \phi(t)$.

Figure 11.4: Simple rule for unnesting update programs.

obtained by the query:

$$R' := (R - Q_\phi) \cup \pi_{\overline{B},\overline{c} \text{ as } \overline{A}}(Q_\phi)$$

We can define in a similar way the result of SQL insert and delete statements.

To define rules for optimizing programs, we rely on the independence of queries from the updates:

Definition 11.7. Let Q be a query and U be an update operation. For a database $\mathcal{A}$ let $U(\mathcal{A})$ denote the database obtained as a result of executing U on $\mathcal{A}$. We say that Q is *independent* of U iff for any input database $\mathcal{A}$, $Q(\mathcal{A}) = Q(U(\mathcal{A}))$.

Figure 11.4 shows a rewrite rule that can be applied iteratively to optimize a database program. Recall that SQL updates have transactional semantics: changes made by update U are not visible to the update condition before the end of the update operation. However, if the update is nested within a for-loop, U will in general be executed multiple times - once for each result tuple returned by Q. Thus, the changes made by U will be visible in subsequent loop iterations. Below, we give a sufficient criterion that ensures that the rewrite rule in Figure 11.4 produces an equivalent program.

Lemma 11.8. *Consider the update statement U defined as follows:*

update R set $\bar{A} = \bar{c}$ where ϕ;

Moreover, let Q_ϕ denote the semi-join query returning the tuples of R that need to be updated. If Q_ϕ is independent of the update U, then the rule in Figure 11.4 preserves equivalence.

Proof. Let P and P' denote the programs on the lhs and rhs of the rule in Figure 11.4, respectively. Let $r \in R$ be a tuple that is updated in P and let this occur when iterating over tuple t from Q. Then $r \in Q_{\phi(t)}(\mathcal{A}')$ where $\phi(t)$ is the condition obtained by substituting the variable \$$t$ in ϕ with the values from t and $\mathcal{A}'$ is the state of the database at the beginning of that iteration of the loop. Since Q_ϕ is independent of U, $Q_\phi(\mathcal{A}') = Q_\phi(U(\mathcal{A}')) = Q_\phi(\mathcal{A})$, where $\mathcal{A}$ is the initial database before the start of P. But then this is equivalent to $r \in Q_{\phi'}(\mathcal{A})$, where $Q_{\phi'}$ is the semi-join query corresponding to the condition $\phi' = \bigvee_{t \in Q} \phi(t)$\$ on the rhs of the rule in Figure 11.4. Thus, r is also updated in P', and P and P' are equivalent. □

We can extend the idea of unnesting updates in several ways. For instance, we are interested in loops with update statements of the form

for(\$$t$ in Q){update R set $\bar{A} = \bar{x}$ where ϕ;}

where $\bar{A} = \bar{x}$ is short for $A_1 = x_1, \ldots, A_m = x_m$ and x_i is either a constant, an attribute A_j of R, or a reference of the form \$$t.B_j$. Some care is required when rewriting such a for-loop with nested update into a single update statement, since it is possible that the same field is set to two different values in two different loop iterations. Hence the final value of the field depends on the order of reading the for-loop tuples, which is normally undesirable. We shall require

that this never happens, that is, a field is always set to the same value or is left unchanged.

Checking independence of queries from updates. The problem of statically deciding whether a query is independent from an insertion or deletion update has been studied in [28] and [40]. The proposed solution consists in checking equivalence of two programs: one that computes the query answer before the update, and one after the update. In our setting, we are also considering updates specified with an SQL update statement. Since those can be simulated with a pair of insert/delete, one can reduce the problem of deciding independence from a general update statement to independence of insertion and deletions. Note however that for the rule in Figure 11.4, we need to check independence of the query corresponding to the where clause of an update statement from the update itself. Thus deciding independence at the level of inserts and deletes will be unnecessarily restrictive, as it will always return a negative answer. We can use a simpler but more precise condition to check update independence, namely: if no attribute appears both on the left-hand side of the set-clause and in the where-clause of an update statement, then the update query is independent of the update.

11.3 Rewriting database programs for bulk execution

We will consider database programs where both updates and user interaction commands can be nested. Let P be the parse tree for a program, where P's nodes are sequences of for-loops, update statements and user interaction (UI) commands. Algorithm 6 shows an algorithm that optimizes a program for bulk

Algorithm 9: Optimize programs

Input: P: program
Output: P': program equivalent to P that can be executed on all worlds or FAIL.

```
1 foreach maximal subtree P_0 of P with no UI operations do
2     Let L:=unnest(P_0);
3     return 'FAIL' if P_0 cannot be unnested;
4     Replace P_0 by L in P;
5 return 'FAIL' if P is not observationally deterministic;
6 Otherwise return P;
```

execution on all worlds, or returns 'FAIL' if no optimization is found. On success the algorithm returns a program that satisfies the following two conditions:

1. All update operations are on the top-level or are nested within loops that operate on certain query results only.
2. The program is observationally deterministic.

The algorithm considers all maximal rooted subtrees of the parse tree for P that do not contain any UI operations. For those the algorithm applies the unnesting techniques (presented in Section 11.2) to turn them into flat sequences of update statements. Let P' be the program that results from this step. If we can verify that P' is observationally deterministic, then P' is the result of the optimization procedure. Finally, we can state our main result:

Theorem 11.9. *If Algorithm 6 returns program P', P' is equivalent to the input program P, satisfies observational determinism, and all update operations are either on the top level or are nested in for-loops that iterate over certain results.*

Part IV

The MayBMS system

CHAPTER 12

SYSTEM ARCHITECTURE

MayBMS is a probabilistic database management system based on the results presented in this thesis. It is developed in a collaborative effort between Cornell and Oxford, and is available as open-source project at sourceforge at

`http://maybms.sourceforge.net`

MayBMS is implemented in C as an extension of the open-source DBMS PostgreSQL [30]. It introduces modification to PostgreSQL in several different parts:

- The parser is extended with uncertainty-aware constructs such as repair-key, `pick-tuples` and the approximate aggregates such as `conf` and `tconf`. See Section 12.2 for an explanation of the query language of MayBMS.
- After the user query is parsed, MayBMS checks whether the query is supported and if so the query is rewritten into a query on U-relational databases as defined by rewritings of Chapter 5. The rewritten query is passed further to the rule-rewriter, optimizer and executor of PostgreSQL.
- The approximate aggregates (described below) are registered as custom aggregates in the system catalog, and implemented at the executor level.

For a discussion of PostgreSQL's internals see e.g. [59] or Chapter 43 of [32].

In contrast to other probabilistic database management systems including but not limited to Trio [15, 43, 63], MystiQ [16, 23] and PrDB [55, 57, 58],

MayBMS is integrated directly inside the DBMS as opposed to being implemented as a layer on top of that. This brings us several advantages which we literally get for free. First, we can take full advantages of PostgreSQL's optimizer. This we owe to the fact that most of the query rewriting is implemented immediately after the parsing step, and the rewriting itself essentially preserves the complexity of the queries. At the same time it allows for implementing optimization techniques directly in the optimizer and executor, like the customized lower level operators for efficient confidence computation [39, 44, 46, 45, 47]. Moreover, users of the system can take advantage of the standard database connectivity interfaces such as JDBC and write programs that use MayBMS as a backend.

We next discuss the representation, and update and query language used by MayBMS. More details about the system, together with a tutorial of how to use it is available in the MayBMS Manual [31].

12.1 Representation

MayBMS uses U-relational databases as its representation system, which are represented as relational tables in the storage layer. Currently, MayBMS supports tuple-level U-relations. Let U be U-relation for attributes $\overline{A}$ and ws-descriptor $\overline{D} = \{D_0, \ldots, D_n\}$ where for a tuple t $t.D_i = (x_i \mapsto j)$ is a mapping from a variable to a domain value. We store U as a table $U[tid, \overline{A}, \_V0, \_D0, \_P0, \ldots, \_Vn, \_Dn, \_Pn]$, where a triple $\_Vi, \_Di, \_Pi$ is used to store the i-th pair of the ws-descriptor of the respective tuple, together with the probability of the variable-to-domain value mapping. While we inline the

world table W in the U-relations, we nevertheless keep a copy of it on disk to ensure consistency among the probability values appearing in different places of the U-relational database, as well as for quick lookups of a variable's probability distribution.

12.2 Query language

The query language of MayBMS is an extension of SQL with uncertainty aware constructs. For syntax and semantics of the standard SQL see for example [32]. We tag tables as t-certain, uncertain and tuple-independent and use those properties to speed up evaluation. A *t-certain* table is a table with no ws-descriptor columns. The t-certainty is a syntactic property guaranteeing that a relation is the same in all worlds. We extend the definition of t-certainty to queries, and define a procedure that decides whether a query is t-certain or not based on the idea of the c-indicators of Section 11.1. We next describe the new constructs introduced in MayBMS's query language, including `repair-key`, `pick-tuples`, the aggregates `conf`, `aconf`, `esum` and `ecount`, and `possible`.

repair-key.

The syntax of `repair-key` is the following:

```
repair key <attributes> in
   (<t-certain-query> | <t-certain-relation>)
   [weight by <expression>];
```

This construct enforces a key constraint on the database by constructing the set of possible repairs, where each repair becomes a possible world. The key attributes are given in the `<attributes>` list, and one can specify a weight for the newly created worlds using the `weight by` clause. Note that `repair-key` is a query, rather than an update statement. One can of course save the result of a repair key query using the `create table` construct. While the number of possible repairs can in general be exponential, `repair key` is evaluated on U-relational databases using a projection query, which produces a table of at most twice the size of the input table.

pick-tuples.

The construct can be used in the following way:

```
pick tuples from
   <t-certain-query> | <t-certain-relation>
   [independently]
   [with probability <expression>];
```

The `pick tuples` construct creates a tuple-independent table in the spirit of [24], and assign probabilities to the tuples. This construct is a syntactic sugar to the language, it can be expressed using `repair key`.

Aggregates.

MayBMS supports the following aggregate functions: `conf, aconf, tconf, esum, ecount`. The `conf` aggregate computes the confidence of tuples in the result of a query. This is the most inefficient operation in general, as the problem is provably #P-complete [29, 25]. MayBMS also supports an ap-

proximation version of the aggregate, `aconf`, which implements the Karb-Luby fully polynomial-time randomized approximation scheme [38, 37, 22].

The aggregates `esum` and `ecount` compute expected sums and counts across groups of tuples. While it may seem that these aggregates are at least as hard as confidence computation (which is #P-hard), this is in fact not so. These aggregates can be efficiently computed exploiting linearity of expectation.

Supported Queries

To guarantee efficiency of query evaluation, we exclude certain kinds of queries whose evaluation is intractable on U-relational databases. We use a static check to verify before execution whether a query is supported or not.

MayBMS supports full SQL on t-certain tables. In addition it supports a large subset of SQL on t-uncertain tables, with even more features supported when fragments of the uncertain query involve t-certain subqueries. The following restrictions apply:

- Exact aggregates and duplicate elimination using `distinct` in a select statement are supported as long as the from clause subqueries and the subqueries in the where condition are t-certain.
- If a t-certain subquery Q in the where condition of a select statement contains references to t-uncertain tables, then the containing query is supported if Q is not correlated with it.
- The set operations `except` and `union` with duplicate elimination are supported when both the left and the right argument are t-certain queries.
- `repair-key` and `pick-tuples` are supported on t-certain queries.

12.3 Updates, concurrency control and recovery

MayBMS supports the usual schema modification and update statements of SQL. As a consequence of our choice of a purely relational representation system, these issues cause surprisingly little difficulty. U-relations are just relational tables and updates are just modifications of these tables that can be expressed using the standard SQL update operations. While the structure of the rewritings could allow for optimizations in the concurrency and recovery managers, those are currently left to the underlying DBMS. We review the main update constructs below, and describe some restrictions to the allowed updates for efficiency and cleanliness of design purposes.

An insertion of the form

```
insert into <uncertain-table> (<uncertain-query>);
```

is just the standard SQL insertion for tables we interpret as U-relations. Thus, the table inserted into must have the right number (that is, a sufficient number) of condition columns. Schema-modifying operations such as

```
create table <uncertain-table> as (<uncertain-query>);
```

are similarly straightforward. A deletion

```
delete from <uncertain-table>
where <condition>;
```

admits conditions that refer to the attributes of the current tuple and may use

t-certain subqueries. One can also update an uncertain table with an update statement

```
update <uncertain-table>
set <attribute> = <expr> [,...]
where <condition>;
```

where the set list does not modify the condition columns and the where condition satisfies the same conditions as that of the delete statement. MayBMS allows users to insert a constant tuple by specifying values for the data columns in an insert statement:

```
insert into <uncertain-table> [<attribute_list>] <tuple>;
```

The requirement that users cannot modify the world-set descriptors and world table directly are in sync with the underlying goal of making the representation system transparent to the user.

CHAPTER 13

APPLICATIONS

The implementation of MayBMS inside the open-source DBMS PostgreSQL allows users to take full advantage of the multiple database connectivity interfaces such as ODBC, JDBC, etc. provided by PostgreSQL, and to write applications in nearly every programming language that use MayBMS as a backend and connect to the database server through those interfaces. This is different from systems such as Trio [15, 43, 63] and MystiQ [16, 23] which are implemented as a software layer on top of an existing DBMS and which require all interaction to the system to be accomplished only through the provided command-line or graphical user interfaces.

We next describe a web-based application which uses MayBMS as backend. The application is implemented in PHP and allows users to model and query random graphs and social networks as probabilistic databases. Users can generate a random graph with certain parameters such as number of nodes and edge probability, or load a social network with probability-weighted edges. Figure 13.1 shows a screenshot of the application.

Chapter 8 already introduced an example of querying a social network stored in a probabilistic database. We consider graphs in which each edge is assigned a probability for its existence, and where edges are independent of each other. Recall that to model such a network as a probabilistic database we store the edge relation as an uncertain table, where each edge is associated with a boolean random variable with a given probability of being true. We use two relations - *edge* and *no_edge* containing the edges in the network and their complement, respectively. An edge $e_i = (u, v)$ is associated with a boolean random

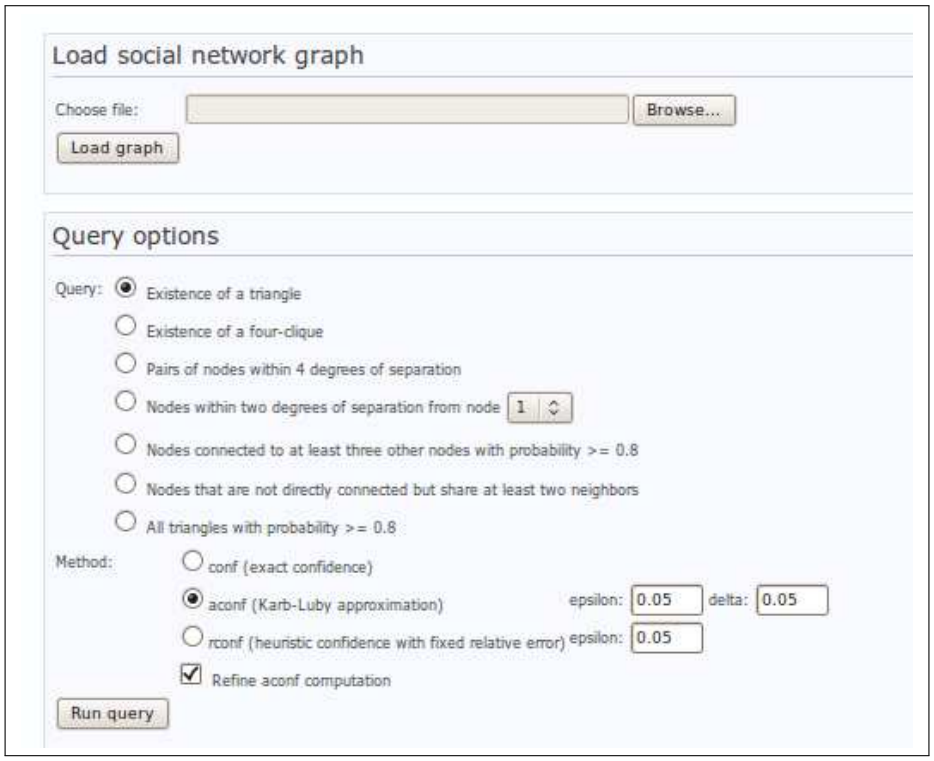


Figure 13.1: Social networks application in MayBMS.

variable x_i where the probability p_i of $x_i = t$ corresponds to the probability that the edge exists. For e_i the *edge* relation will contain a tuple $(u, v, (x_i \mapsto t, p_i))$, and the relation *no_edge* - a tuple $(u, v, (x_i \mapsto f, 1 - p_i))$. For simplicity in this application we assume that edges are independent of each other, but in general MayBMS poses no restrictions on the existence of correlations between them.

The applications allows users to load the social network data, and compute the probability of different patterns in the network. Some of the currently supported queries are:

- probability for the existence of a triangle (or a four-clique)
- pairs of nodes within four degrees of separation
- nodes with 7at least three neighbors with probability higher than 80%

```
select e1.u,e2.v, aconf(0.05, 0.05) as aconf
from edge e1, edge e2, edge e3, edge e4, no_edge ne
where e1.u = e3.u and e2.v = e4.v and e1.v = e2.u and
      e3.v = e4.u and e1.v < e3.v and e1.u = ne.u and
      e2.v = ne.v and e1.u <> e2.v
group by e1.u,e2.v;
```

Figure 13.2: Computing pairs of nodes which are not directly connected but share at least two neighbors in MayBMS.

- nodes which are not directly connected but share at least two neighbors

The last query from above computing pairs of nodes which are not directly connected but share at least two neighbors can be expressed in MayBMS with the query in Fig 13.2. The query uses the approximate confidence computation algorithm with parameters 0.05, 0.05, denoting that the computed probability is within 0.05 of the real probability with high likelihood (above 0.95).

In addition to loading social networks from a file, the user can generate a random graph by specifying properties such as number of nodes, and probability of edges (either fixed or random).

Part V

Epilog

CHAPTER 14

CONCLUSION AND FUTURE WORK

This dissertation presented different aspects of managing uncertain information, including representation system, query language and evaluation, confidence computation and designing APIs for probabilistic databases. U-relational databases, the model behind MayBMS are a simple representation system for uncertain data. They are more succinct than other existing representation systems and at the same time allow positive relational algebra queries to be evaluated purely relationally on U-relations, a property not shared by any other previous succinct representation system. Also, U-relations are a simple formalism which poses a small burden on implementors.

In terms of future work, several main directions can be identified, among which improved confidence computation algorithms for special cases of data and queries, and cost-based optimization of queries on partitioned data. Algorithms for confidence computation, a problem with known hard complexity, can benefit from additional knowledge about the data and queries, such as independence, dependencies etc. Moreover, a probabilistic database management system should be able to decide which algorithm to use at run time based on static analysis and statistics about the data. A cost-based model can be built also for evaluating queries on data that is partitioned, which will allow to leverage late materialization techniques to speed up execution.

BIBLIOGRAPHY

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proc. VLDB*, 2007.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Parag Agrawal and Jennifer Widom. Confidence-Aware Join Algorithms. In *Proc. ICDE*, 2009.

[4] Periklis Andritsos, Ariel Fuxman, and Renee J. Miller. Clean Answers over Dirty Databases: A Probabilistic Approach. In *Proc. ICDE*, 2006.

[5] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and Simple Relational Processing of Uncertain Data. Technical Report cs.DB/0707.1644, ACM CORR, 2007.

[6] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. "Fast and Simple Relational Processing of Uncertain Data". In *Proc. ICDE*, 2008.

[7] Lyublena Antova and Christoph Koch. On APIs for Probabilistic Databases. In *Proc. QDB/MUD*, 2008.

[8] Lyublena Antova, Christoph Koch, and Dan Olteanu. 10^{10^6} Worlds and Beyond: Efficient Representation and Processing of Incomplete Information. In *Proc. ICDE*, 2007.

[9] Lyublena Antova, Christoph Koch, and Dan Olteanu. From Complete to Incomplete Information and Back. In *Proc. SIGMOD*, 2007.

[10] Lyublena Antova, Christoph Koch, and Dan Olteanu. MayBMS: Managing Incomplete Information with Probabilistic World-Set Decompositions. In *Proc. ICDE*, 2007. Demonstration Paper.

[11] Lyublena Antova, Christoph Koch, and Dan Olteanu. World-set Decompositions: Expressiveness and Efficient Algorithms. In *Proc. ICDT*, 2007.

[12] Lyublena Antova, Christoph Koch, and Dan Olteanu. 10^{10^6} Worlds and Beyond: Efficient Representation and Processing of Incomplete Information. *The VLDB Journal*, 18(**5**), 2009.

[13] Don S. Batory. On Searching Transposed Files. *ACM Trans. Datab. Syst.*, **4**(4):531–544, 1979.

[14] Omar Benjelloun, Anish Das Sarma, Alon Halevy, Martin Theobald, and Jennifer Widom. Databases with Uncertainty and Lineage. *The VLDB Journal*, 17(2):243–264, 2008.

[15] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. ULDBs: Databases with Uncertainty and Lineage. In *Proc. VLDB*, 2006.

[16] Jihad Boulos, Nilesh Dalvi, Bhushan Mandhani, Shobhit Mathur, Chris Re, and Dan Suciu. MYSTIQ: a System for Finding More Answers by Using Probabilities. In *Proc. SIGMOD*, 2005.

[17] Eugene Charniak. Bayesian Networks without Tears. *AI Magazine*, **12**(4), 1991.

[18] E. F. Codd. "Understanding Relations (Installment #7)". *FDT - Bulletin of ACM SIGMOD*, 7(3):23–28, 1975.

[19] E. F. Codd. "Extending the Database Relational Model to Capture More Meaning". *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[20] Gregory F. Cooper. The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artificial Intelligence*, **42**(2-3), 1990.

[21] Graham Cormode, Feifei Li, and Ke Yi. Semantics of Ranking Queries for Probabilistic Data and Expected Ranks. In *Proc. ICDE*, 2009.

[22] Paul Dagum, Richard Karp, Michael Luby, and Sheldon Ross. An optimal algorithm for monte carlo estimation. *SIAM Journal of Computing*, **29**(5), 2000.

[23] Nilesh Dalvi, Christopher Ré, and Dan Suciu. Probabilistic Databases: Diamonds in the Dirt. *Communications of the ACM*, **52**(7), 2009.

[24] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *Proc. VLDB*, 2004.

[25] Nilesh Dalvi and Dan Suciu. "Efficient query evaluation on probabilistic databases". *The VLDB Journal*, **16**(4), 2007.

[26] Nilesh Dalvi and Dan Suciu. The Dichotomy of Conjunctive Queries on Probabilistic Structures. In *Proc. PODS*, 2007.

[27] Rina Dechter. Bucket Elimination: a Unifying Framework for Processing-Hard and Soft Constraints. *Constraints*, **2**(1), 1997.

[28] Charles Elkan. Independence of Logic Database Queries and Updates. In *Proc. PODS*, pages 154–160, 1990.

[29] Erich Grädel, Yuri Gurevich, and Colin Hirsch. "The Complexity of Query Reliability". In *Proc. PODS*, 1998.

[30] PostgreSQL Global Development Group. PostgreSQL Database Management System, 1996-2009.

[31] The MayBMS Development Group. *MayBMS: A Probabilistic Database System User Manual*, 2005-2009. Available at `http://maybms.sourceforge.net/`.

[32] The PostgreSQL Global Development Group. *PostgreSQL 8.4.1 Documentation*, 1996-2009.

[33] Oktie Hassanzadeh and Rene J. Miller. Creating Probabilistic Databases From Duplicated Data. *The VLDB Journal*, **18**(5), 2009.

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

[35] T. Imielinski, S. Naqvi, and K. Vadaparty. Incomplete Objects – a Data Model for Design and Planning Applications. In *Proc. SIGMOD*, 1991.

[36] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J. Haas. MCDB: A Monte Carlo Approach to Managing Uncertain Data. In *Proc. SIGMOD*, 2008.

[37] R. M. Kapp, M. Luby, and N. Madras. Monte-Carlo Approximation Algorithms for Enumeration Problems. *Journal of Algorithms*, **10**(3), 1989.

[38] Richard M. Karp and Michael Luby. Monte-Carlo Algorithms for Enumeration and Reliability Problems. In *Proc. SFCS*, 1983.

[39] Christoph Koch and Dan Olteanu. Conditioning Probabilistic Databases. *PVLDB*, **1**(1), 2008.

[40] Alon Y. Levy and Yehoshua Sagiv. Queries Independent of Updates. In *19th International Conference on Very large Data Bases*, pages 171–181, 1993.

[41] Jian Li, Barna Saha, and Amol Deshpande. A Unified Approach to Ranking in Probabilistic Databases. *PVLDB*, **2**(1), 2009.

[42] Jordan M.I. and Weiss Y. *Graphical Models: Probabilistic Inference*. MIT Press, 2nd edition, 2002.

[43] Michi Mutsuzaki, Martin Theobald, Ander de Keijzer, Jennifer Widom, Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Raghotham Murthy, and Tomoe Sugihara. Trio-One: Layering Uncertainty and Lineage on a Conventional DBMS (Demo). In *Proc. CIDR*, 2007.

[44] Dan Olteanu and Jiewen Huang. Using OBDDs for Efficient Query Evaluation on Probabilistic Databases. In *Proc. SUM*, 2008.

[45] Dan Olteanu and Jiewen Huang. Secondary-storage Confidence Computation for Conjunctive Queries with Inequalities. In *Proc. SIGMOD*, 2009.

[46] Dan Olteanu, Jiewen Huang, and Christoph Koch. SPROUT: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases. In *Proc. ICDE*, 2009.

[47] Dan Olteanu, Jiewen Huang, and Christoph Koch. Approximate Confidence Computation in Probabilistic Databases. In *Proc. ICDE*, 2010.

[48] Dan Olteanu, Christoph Koch, and Lyublena Antova. World-set decompositions: Expressiveness and efficient algorithms. *Theoretical Computer Science*, **403**(2-3), 2008.

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

[50] Christopher Re, Nilesh Dalvi, and Dan Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In *Proc. ICDE*, 2007.

[51] Christopher Ré and Dan Suciu. The Trichotomy of HAVING Queries on a Probabilistic Database. *The VLDB Journal*, **18**(5), 2009.

[52] Irina Rish. *Efficient Reasoning in Graphical Models*. PhD thesis, 1999.

[53] Anish Das Sarma, Martin Theobald, and Jennifer Widom. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *Proc. ICDE*, 2008.

[54] Prithviraj Sen and Amol Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. Technical Report CS-TR-4820, University of Maryland, College Park, 2006.

[55] Prithviraj Sen and Amol Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In *Proc. ICDE*, 2007.

[56] Prithviraj Sen, Amol Deshpande, and Lise Getoor. Representing Tuple and Attribute Uncertainty in Probabilistic Databases. In *ICDM Workshops*, 2007.

[57] Prithviraj Sen, Amol Deshpande, and Lise Getoor. Exploiting Shared Correlations in Probabilistic Databases. *PVLDB*, **1**(1), 2008.

[58] Prithviraj Sen, Amol Deshpande, and Lise Getoor. PrDB: Managing and Exploiting Rich Correlations in Probabilistic Databases. *The VLDB Journal*, **18**(5), 2009.

[59] Stefan Simkovics. Enhancement of the ANSI SQL Implementation of PostgreSQL. Master's Thesis.

[60] Mohamed Soliman, Ihab Ilyas, and Kevin C. Chang. "Top-k Query Processing in Uncertain Databases". In *Proc. ICDE*, 2007.

[61] Mohamed A. Soliman and Ihab F. Ilyas. Ranking with Uncertain Scores. In *Proc. ICDE*, 2009.

[62] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang.

[63] Stanford Trio Project. *TriQL – The Trio Query Language*, 2006. Available at `http://infolab.stanford.edu/~widom/triql.html`.

[64] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden,

Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In *Proc. VLDB*, 2005.

[65] Transaction Processing Performance Council. *TPC Benchmark H (Decision Support)*, revision 2.6.0 edition, 2006. Available at `http://www.tpc.org/tpch/spec/tpch2.6.0.pdf`.

[66] Daisy Zhe Wang, Eirinaios Michelakis, Minos Garofalakis, and Joseph M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, **1**(1), 2008.

[67] Ke Yi, Feifei Li, George Kollios, and Divesh Srivastava. Efficient Processing of Top-k Queries in Uncertain Databases. In *Proc. ICDE*, 2008.

[68] Ke Yi, Feifei Li, George Kollios, and Divesh Srivastava. Efficient Processing of Top-k Queries in Uncertain Databases with x-Relations. *IEEE Trans. Knowl. Data Eng.*, **20**(12), 2008.

[69] W. W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, **33**, 1977.

[70] Nevin Zhang and David Poole. A Simple Approach to Bayesian Network Computations. In *Canadian Conference on Artificial Intelligence*, 1994.