UNIVERSITÀ DELLA CALABRIA

DOCTORAL THESIS

Approximate Query Answering over Incomplete and Inconsistent Databases

Author:
Nicola FIORENTINO

Supervisors:
Cristian MOLINARO
Irina TRUBITSYNA

Dottorato di Ricerca in
Information and Communication Technologies
XXXIII ciclo

UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

Acknowledgements

I would like to express my gratitude to my supervisors, Prof. Cristian Molinaro and Prof. Irina Trubitsyna, and to the team I have worked with. Special thanks to Antonio Caliò and Domenico Mandaglio, who have been precious companions on this journey.

Contents

To my family

Preface

There are two central issues when we talk about data management: data incompleteness and data inconsistency. When we deal with incomplete databases, relations may contain only part of relevant data, whereas in inconsistent databases, data in the tables may not satisfy some of the integrity constraints defined over the schema. These are scenarios where the information in the data is not uniquely understood due to the ambiguity. As a consequence, standard query answering may not be effective and ad hoc techniques are required. This thesis focuses on the problem of querying both incomplete and inconsistent databases, with the aim to provide relevant answers in the presence of either incomplete or inconsistent information. In the years, a number of semantics for query answering in the presence of incomplete or inconsistent data have been proposed, and we embrace the most consolidated and significant of them, that is, certain and consistent query answers. These are answers we can be confident about even if ambiguity in the data has been encountered. Unfortunately, computing certain and consistent query answers are intractable problems, thus recent research has focused on developing polynomial time evaluation algorithms with correctness guarantees.

In the first part of this thesis we deal with incomplete information and we illustrate a number of these efficient techniques, capable to compute a sound but possibly incomplete set of certain answers. Then, we propose a new approach which provides better approximations than the most of the current techniques, while retaining polynomial time data complexity. The central tools of these techniques are conditional tables and the conditional evaluation of queries. Also, we propose a system prototype offering a suite of state-of-the-art approximation algorithms enabling users to choose the technique that best meets their needs in terms of balance between efficiency and quality of the results.

In the second part of this thesis, we deal with consistent query answering of (possibly inconsistent) knowledge bases. This problem relies on two central notions: a repair, that is, a maximal consistent subset of the facts in the knowledge base, and a consistent

query answer, that is, a query answer entailed by every repair of the knowledge base. We present a system, which allows users to query inconsistent knowledge bases. Specifically, equality generating dependencies are considered. Different from the standard notion of repair, where entire facts are deleted to restore consistency (which might lead to loss of useful information), the repair strategy adopted by our system performs value updates within facts, thereby preserving more information in the knowledge base. An inconsistent knowledge base can admit multiple repairs; our system computes a compact representation of all of them, called *universal repair*, which is also leveraged for query answering. Since consistent query answering is intractable in the considered setting, we implemented a polynomial time algorithm to compute a sound (but not necessarily complete) set of consistent query answers. The classification of query answers into consistent and non-consistent ones is a somewhat coarse-grained classification, as it does not provide much information about non-consistent query answers (e.g., a query answer entailed by 99 out of 100 repairs might be considered "almost consistent"). To overcome this limitation, in the last part of this thesis we propose a probabilistic approach to querying inconsistent knowledge bases, which provides more informative query answers by associating a degree of consistency with each query answer by associating a probability to each repair, depending on the changes needed to obtain it.

This thesis is organized as follows. In Chapter 1 we introduce basic concepts and notations about relational databases, representation systems to deal with incomplete information (with focus on condational tables), integrity constraints and repairs. In Chapter 2 we illustrate state-of-the-art evaluation algorithms to compute a sound but possibly incomplete set of certain query answers, whereas in Chapter 3 we present a system offering the implementation of a suite of approximation algorithms, which are those presented in Chapter 2 under the name of *GMT-approaches*. In Chapter 4 we propose a novel technique that allows us to improve the approximation algorithms presented in Chapter 2. Chapter 5 presents a system to compute the universal repair of inconsistent knowledge bases, whereas in Chapter 6 we propose a probabilistic approach to querying inconsistent databases. Finally, conclusions are drawn.

Chapter 1

Preliminaries

1.1 Relational Databases

A database is a collection of data organized to model relevant aspects of reality and to support processes requiring this information.

A database model is a theory or specification describing how a database is structured and used. It provides the means for specifying particular data structures, for constraining the data sets associated with these structures, and for manipulating the data. A number of database models have been proposed (hierarchical model, network model, object model. etc.). In 1970 E. F. Codd introduced the *relational model* as a way to make DBMs (Databse Management System) independent of any particular application, defining it as a mathematical model based on predicate logic and set theory [24]. Nowadays the relational model is the most popular and used and in this chapter we recall the basic notions of it, together with relational query languages and the basic types of data dependencies.

1.1.1 Relational Model

We assume the existence of the following pairwise disjoint sets:

- a countably infinite set `Const` of *constants*, called *database domain*;
- a countably set $\mathcal{A}$ of *attributes*, where each attribute $A_i \in \mathcal{A}$ is associated with a set of constants called *attribute domain* and denoted as $dom(A_i)$;
- a countably infinite set $\mathcal{R}$ of *relation names*, where each relation name $R \in \mathcal{R}$ is associated with a finite sequence of attribute $A_1, \ldots, A_n$, where n is the *arity* of R.

We say that $R(A_1, \ldots, A_n)$ is a *relation schema* and it may be referred to as $R(U)$, where $U = \{A_1, \ldots, A_n\}$.

A *relation* r over $R(A_1, \ldots, A_n)$ is a finite subset of $dom(A_1) \times \cdots \times dom(A_n)$. We also say that r is a relation of R. Each element t of r is called a *tuple*. The notation $t[A_i]$ is used to denote the A_i-component of t. Similarly, for a set of attributes $X \subseteq \{A_1, \ldots, A_n\}$, $t[X]$ is used to denote the restriction of tuple t to X. A *database schema* is a a non-empty finite set $\mathbf{R} = \{R_1(U_1), \ldots, R_m(U_m)\}$ of relation schemas. A *database instance* (or simply *database*) D over $\mathbf{R}$ is a finite set of relations $\{r_1, \ldots . r_m\}$, where each r_i is a relation over $R_i(U_i)$. We use the notation $R_i(t)$ to indicate that a tuple t belongs to a relation r_i over schema $R_i(U_i)$, and call it a *fact*. A database can be viewed as a finite set of facts. Now we give the definition of *atom*: an atom A is on object of the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate and the t_i's are terms, and a term is a constant or a variable. From this perspective, a $fact$ can be viewed as an atom without variables.

Integrity Constraints

Integrity Constraints (or *Data Dependencies*) express semantic information about data, i.e. relationship that should hold among data. More formally, an integrity constraint can be viewed as a first-order logic sentence of the form:

$$\forall \mathbf{x} \forall \mathbf{y}\ \phi(\mathbf{x}, \mathbf{y}) \rightarrow \exists \mathbf{z}\ \psi(\mathbf{x}, \mathbf{z})$$

where $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are tuples of variables, $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$ are conjunctions of (relation and equality) atoms. $\phi(\mathbf{x}, \mathbf{y})$ (resp. $\psi(\mathbf{x}, \mathbf{z})$) is called the *body* (resp. the *head*) of the constraint. Without loss of generality we can assume that equality atoms may appear only in the head of the dependency and that there is no existentially quantified variable involved in an equality atom.

Below we present some of the most common kinds of data dependencies.

Equality Generating Dependencies (EGDs). An *equality generating dependency* (EDG) is a data dependency whose head is a single equality atom. It is a first-order logic sentence of the form:

$$\forall \mathbf{x}\ \phi(\mathbf{x}) \rightarrow x_i = x_j$$

where $\mathbf{x}$ is a tuple of variables, $x_i, x_j \in \mathbf{x}$ and $\phi(\mathbf{x})$ is a conjunctions of relation atoms.

Tuple Generating Dependencies (TGDs). A *tuple generating dependency* (TGD) is a first-order logic sentence of the form:

$$\forall \mathbf{x} \forall \mathbf{y}\ \phi(\mathbf{x}, \mathbf{y}) \rightarrow \exists \mathbf{z}\ \psi(\mathbf{x}, \mathbf{z})$$

where $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are tuples of variables, $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$ are conjunctions of relation atoms (no equality atoms occurring in it).

Functional Dependencies. Given a relation schema $R(U)$, a *functional dependency* f over $R(U)$ is an expression of the form $X \rightarrow Y$, where $X, Y \subseteq U$. If Y is a just one attribute, then f is said to be in *standard form*, whereas if $Y \subseteq X$, then the functional dependency is *trivial*. A relation r over $R(U)$ *satisfies* f, denoted $r \models f$, iff $\forall t_1, t_2 \in r, t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$ (and this means that r is *consistent* w.r.t. f). Furthermore, r satisfies (or is consistent w.r.t.) a set F of function dependencies over $R(U)$, denoted as $r \models F$, iff r satisfies each functional dependency in F. We say that F *logically implies* a functional dependency f, denoted $F \models f$, iff for each relation r over $R(U)$, if r satisfies F, then r satisfies f.

A functional dependency can be viewed as a special case of equality generating dependency.

Key Dependencies. A *key dependency* is a special case of a functional dependency (and of an EGD), characterized by the form $X \rightarrow U$. Given a set F of functional dependencies, a *key* of R is a minimal (under set inclusion) set K of attributes of R such that F logically implies $K \rightarrow U$. We say that every attribute in K a *key attribute*, and a *primary key* of R is a designated key of R.

Foreign Key Constraints. Given two relation schemas $R(U)$ and $S(V)$, a *foreign key constraint* fk is an expression of the form $R(W) \subseteq S(Z)$, where $W \subseteq U, Z \subseteq V, |W| = |Z|$ and Z is a key of S (if Z is a primary key of S we call fk a *primary foreign key constraint*). Two relations r and s over $R(U)$ and $S(V)$, respectively, *satisfy* fk iff for each tuple $t_1 \in r$ there is a tuple $t_2 \in s$ such that $t_1[W] = t_2[Z]$ (r and s are said to be *consistent* w.r.t. fk).

A foreign key constraint can be viewed as a special case of tuple generating dependency.

Example 1.1. Assume you have a database schema consisting of the following two relation schemas: *Student(stud_id, stud_name, dept)* and *Department(dept_id, dept_name)*. We want to force the condition that every relation over the first relation schema cannot contain two different tuples with the same id (i.e. the same value on *stud_id*). Similarly, we want to force that every relation over the second relation schema cannot contain two different tuples with the same id (i.e. the same value on *dept_id*). Moreover, it would be reasonable to impose that if a department code appears in the student relation, it must appear in the department relation too. The first two constraints mentioned above are examples of *key dependencies* - attribute *stud_id* is said to be a key of *Student* and *dept_id* is said to be a key of *Department*. The third dependency mentioned above is an example of *foreign key constraint* - particularly, attribute *dept* of *Student* is a foreign key, referring to attribute *dept_id* of *Department*. □

Example 1.2. Consider the following database:

stud_id	*stud_name*	*dept*
s1	Tom	d1
s2	Bill	d2
s3	Eustace	d3

dept_id	*dept_name*
d1	Maths
d2	Biology

Attribute *stud_id* is a key of the first relation, whereas attribute *dept_id* is a key of the second one (this is illustrated by underlining *stud_id* and *dept_id*). We may denote the foreign key constraint imposing that each department appearing in the employee relation must appear in the department relation with the expression $Student[dept] \subseteq Department[dept\_id]$. While the database above satisfies the key constraints, you may notice that the foreign key constraint is not satisfied because there doesn't exist a tuple in the department relation having d3 as *dept_id*-value. □

1.1.2 Query Languages

Query languages are tools whose aim is to derive information from databases. A *query* Q is a function that takes a database D as input and returns a relation as output. We indicate the result of applying Q to D as $Q(D)$. There are a number of query languages, such as *relational algebra* [24], *relational calculus* [27] and *SQL* (Structured Query Language) [80]. Each language has an expressive power defined as the set of queries that can be expressed by means of that language.

Relational Algebra

Relational algebra is an extension of the algebra of sets and consists of five basic operators. Assume you have two relation schemas, $R(A_1, \ldots, A_n)$ and $S(B_1, \ldots, B_m)$, and let r and s be two relations over the first and second schemas, respectively. The basic relational algebra operators have the following definitions:

- *Cartesian product*: $r \times s = \{(r_1, \ldots, r_n, s_1, \ldots, s_m) \mid (r_1, \ldots, r_n) \in r \wedge (s_1, \ldots, s_m) \in s\}$;
- *Union*: $r \cup s = \{t \mid t \in r \vee t \in s\}$;
- *Difference*: $r - s = \{t \mid t \in r \wedge t \notin s\}$;
- *Projection*: $\pi_A(r) = \{t[A] \mid t \in r\}$, where $A \subseteq \{A_1, \ldots, A_n\}$;
- *Selection*: $\sigma_F(r) = \{t \mid t \in r \wedge F(t)\}$, where F is a logical formula built using propositional logical connectives and atomic formulas of the form $E_1 \; op \; E_2$, where op is a comparison operator, whereas E_1 and E_2 are constants or attributes names. $F(t)$ denotes the logical value given as result of the logical evaluation of F over tuple t.

The relational algebra defined so far is also called *named*, since attribute names are used in the relational algebra operators. There exists an *unnamed* relational algebra where attributes are referred to by their positions in the relation schema. In the named relational algebra we also have a unary operator called *renaming* and defined as follows: $\rho_{A_1/B_1,\ldots,A_k/B_k}(r)$, where the A_i's and B_j's are attribute names such that $A_i \neq B_i$, returns the input relation r with a schema derived from the schema of r by renaming each attribute A_i as B_i, for $i = 1, \ldots, k$.

A number of derived operators have been defined too. Derived operators do not increase the expressive power of the language (i.e. they do not allow us to express further queries), but are introduced to make expressions more comprehensible and their evaluation more efficient. As an example, the derived operators intersection and (theta) join are defined as follows:

- *Intersection*: $r \cap s = \{t \mid t \in r \wedge t \in s\} = r - (r - s) = s - (s - r)$;
- *Join*: $r \bowtie_F s = \{(r_1, \ldots, r_n, s_1, \ldots, s_m) \mid (r_1, \ldots, r_n) \in r \wedge (s_1, \ldots, s_m) \in s \wedge F(r_1, \ldots, r_n, s_1, \ldots, s_m)\} = \sigma_F(r \times s)$, where F is a selection expression.

Natural join, different types of outer joins and semi-join, division are examples of other derived operators.

Example 1.3. Assume you have two relations *student* and *department* over schema $Student(stud\_id, stud\_name, dept)$ and $Department(dept\_id, dept\_name)$, respectively. The query asking for names of the students who study in the department *Maths* may be expressed as:

$$\pi_{stud\_name}(student \bowtie_{dept=dept\_id} \sigma_{dept\_name="Maths"}(department)).$$

□

We use the notation $\mathcal{RA}$ to denote the set of queries expressible in relational algebra.

Relational Calculus

Another formalism to express queries in the relational model is *relational calculus*, which is described below. Particularly, the language we present here is also called *domain calculus*, because variables vary over the underlying database domain; in *tuple calculus*, which is not considered here, variables vary over tuples.

We assume the existence of a set V of variables. Recall that `Const` is used to denote the database domain. A *term* is a constant in `Const` or a variavle in V. *Formulas* are inductively defined as follows:

- a (basic) formula, also called *atom*, is (i) an expression of the form $R(w_1, \ldots, w_n)$, where R is a relation name with arity n and w_i's are terms - this is called a *standard atom*, or (ii) an expression of the form $w_1\ op\ w_2$, where w_1, w_2 are terms and op is a comparison operator (i.e. $op \in \{>, <, \geq, \leq, =, \neq\}$) - this is called a *built-in atom*;
- if G and H are formulas, then $(G \wedge H)$, $(G \vee H)$, $\neg G$ are formulas;
- if x is a variable in V and G is a formula, then $\exists x G$ and $\forall x G$ are formulas.

We now define *free* variable occurrences. We say that an occurrence of a variable x in a formula is *free* if one of the following holds:

- F is a basic formula;

- $F = \neg G$ and the occurrence of x is free in G;
- $F = G \wedge H$ (resp. $F = G \vee H$) and the occurrence of x is free in G or H;
- $F = \exists y G$ (resp. $F = \forall y G$), x and y are distinct variable, and the occurrence of x is free in G.

We use $free(F)$ to denote the set of free variables of F, that is the variables appearing in F having at least one free occurrence in F.

A *relational calculus query* is an expression of the form:

$$\{(u_1, \ldots, u_n) \mid F\}$$

where the u_i's are terms (the same term can be repeated), F is a formula, and the variables in $\{u_1, \ldots, u_n\}$ are exactly the free variables of F.

Example 1.4. Consider again the database schema consisting of the two relation schemas $Student(stud\_id, stud\_name, dept)$ and $Department(dept\_id, dept\_name)$. The query asking for the names of the employees working in the *Maths* department is expressed by the following relational calculus query:

$$\{(y) \mid \exists x \, \exists z \, Student(x, y, z) \wedge Department(z, Maths)\}$$

□

The semantics of a relational calculus query over a database is defined w.r.t. a particular domain $\texttt{Const}'$, called *evaluation domain*, which is intended to specify the constants over which variables can range. Before defining the semantics of relational calculus queries, we introduce some notations and terminology used in the following. A *valuation* for a set of variables $V' \subseteq V$ is a mapping $\nu : V' \cup \texttt{Const} \rightarrow \texttt{Const}$ such that $\nu(c) = c, \forall c \in \texttt{Const}$. We use $\nu|_{V''}$ to denote the restriction of ν to $V'' \subseteq V'$. The *active domain* of a database D, denoted $adom(D)$, is the set of constants appearing in D. Likewise, we use $adom(Q)$ and $adom(F)$ to denote the set of constants appearing in relational calculus query Q and formula F, respectively.

Let D be a database, $\texttt{Const}'$ be the *evaluation domain* such that $adom(D) \subseteq \texttt{Const}' \subseteq \texttt{Const}$, F a formula such that $adom(F) \subseteq \texttt{Const}'$, and ν a valuation for the free variables of F with range contained in $\texttt{Const}'$. Then, we say that D *satisfies* F for ν *relative* to $\texttt{Const}'$, denoted $D \models F[\nu]$ ($\texttt{Const}'$ is understood), if one of the following holds:

- $F = R(w_1, \ldots, w_m)$ and $R(\nu(w_1), \ldots, \nu(w_m))$ is a fact of D.
- $F = w_1\ op\ w_2$, with $op \in \{>, <, \geq, \leq, =, \neq\}$, and $\nu(w_1)\ op\ \nu(w_2)$ is `true`.
- $F = G \wedge H$, $D \models G[\nu|_{free(G)}]$, and $D \models H|_{free(H)}]$.
- $F = G \vee H$. In addition, $D \models G[\nu|_{free(G)}]$ or $D \models H|_{free(H)}]$.
- $F = \neg G$ and $D \models G[\nu|_{free(G)}]$ does not hold.
- $F = \exists x G$ and $\exists c \in \mathtt{Const}'$, $D \models G[\nu']$, where ν' is a valuation for x and the variables of ν such that $\nu'(x) = c$ and $\nu'(y) = \nu(y)$ for any other variable y.
- $F = \forall x G$ and $\forall c \in \mathtt{Const}'$, $D \models G[\nu']$, where ν' is a valuation for x and the variables of ν such that $\nu'(x) = c$ and $\nu'(y) = \nu(y)$ for any other variable y.

We define the semantics of a relational calculus query $Q = \{(u_1, \ldots, u_n) \mid F\}$ over a database D w.r.t. the evaluation domain $\mathtt{Const}'$, where $((adom(D) \cup adom(Q)) \subseteq \mathtt{Const}' \subseteq \mathtt{Const}$. The role of the $\mathtt{Const}'$ is to specify the constants over which variables can range. Notice that the supersets of $adom(D) \cup adom(Q)$ are the only domains with respect to which it makes sense to evaluate Q over D. We define the semantics of Q over D w.r.t. $\mathtt{Const}'$ as follows:

$$Q_{\mathtt{Const}'}(D) = \{(\nu(u_1) \ldots, \nu(u_n)) \mid D \models F[\nu] \text{ and } \nu \text{ is a valuation for } free(F) \text{ with range } \subseteq \mathtt{Const}'\}$$

When $\mathtt{Const}' = \mathtt{Const}$ the semantics above corresponds to the standard interpretation of predicate calculus. Note that if $\mathtt{Const}'$ is infinite, then $Q_{\mathtt{Const}'}(D)$ can be an infinite set of tuples.

We use the notation $\mathcal{RC}$ to denote the set of queries expressible in relational calculus

Domain Independent and Safe RC Queries

A relational calculus query Q is *domain independent* if for each database D, and for each couple $\mathtt{Const}', \mathtt{Const}''$ such that $(adom(D) \cup adom(Q)) \subseteq \mathtt{Const}', \mathtt{Const}'' \subseteq \mathtt{Const}$, it happens that $Q_{\mathtt{Const}'}(D) = Q_{\mathtt{Const}''}(D)$. Thus, for an arbitrary database, a domain independent relational calculus query gives the same result regardless of the domain w.r.t. which it is evaluated. This means that, if Q is domain independent,

then $Q_{\mathtt{Const}'}(D)$ does not change when $\mathtt{Const}'$ changes. Therefore $Q_{\mathtt{Const}'}(D)$ can be computed for $\mathtt{Const}' = adom(D) \cup adom(Q)$.

Example 1.5. Assume you have the relation schema $R(A, B)$ and the following relational calculus queries:

- $Q^1 = \{(x, y) \mid \exists u \, \exists v(R(u, v) \wedge R(x, y))\}$;
- $Q^2 = \{(x, y) \mid \neg R(x, y)\}$;
- $Q^3 = \{(x) \mid \forall y R(x, y)\}$.

All the queries above are not domain independent. To see why, consider a relation $r = \{(c_1, c_1), (c_1, c_2)\}$ and let $\mathtt{Const}'$ be a domain. It is easy to check that $Q^1_{\mathtt{Const}'}(\{r\}) = \{(x, y) \mid x \in \mathtt{Const}' \wedge y \in \mathtt{Const}'\}$ and $Q^2_{\mathtt{Const}'}(\{r\}) = \{(x, y) \mid x \in \mathtt{Const}' \wedge y \in \mathtt{Const}' \wedge (x, y) \notin r\}$. As the results of Q^1 and Q^2 contain values of $\mathtt{Const}'$, then their results clearly depend on $\mathtt{Const}'$. One can see that Q^3 will always contain values taken from the input relation; however, it is not domain dependent. Indeed, it is easy to see that if $\mathtt{Const}'$ is infinite, then $Q^3_{\mathtt{Const}'}(\{r\})$ is empty. The same holds if, for instance, $\mathtt{Const}' = \{c_1, c_2, c_3\}$. Nevertheless, if $\mathtt{Const}' = \{c_1, c_2\}$, then $Q^3_{\mathtt{Const}'}(\{r\}) = \{(c_1)\}$. Hence Q^3 may return different results over the same relation when different domains are considered. □

Theorem 1.6. *The problem of deciding whether a relational calculus query is domain independent is undecidable (cf. [81]).*

It is important to observe that the fact that we can express relational calculus queries which are not domain independent is not a positive aspect because, in the presence of an infinite database domain, we could get query answer whose number of tuples is infinite.

In the theorem 1.6 we saw that there does not exist an algorithm to determine whether a relational calculus query is domain independent. We said that a relational calculus query that is not domain independent is not desirable too. Therefore, we now present some syntactical restrictions that lead to a class of relational calculus queries, called $safe$, that are guaranteed to be domain independent. Safe relational calculus queries are a subset of the domain independent relational calculus queries.

Safe relational calculus (SRC) is derived from relational calculus by forcing the following restriction on formulas:

- the universal quantifier $\forall$ is not used. This does not invalidate the expressiveness of the language as expressions of the form $\forall x F$ may be written as $\neg(\exists x \neg F)$;

- the disjunctions operator is applied only to logical expressions having the same set of free variable;
- for any maximal sub-formulas F of the form $F_1 \wedge \cdots \wedge F_n$, all the free variables of F must be *limited* as explained in the following:
  - a variable is limited if it is free in some F_i and F_i is not an arithmetic comparison and is not negated;
  - if F_i is of the form $x = c$ or $c = x$, where x is a variable and c is a constant, then x is limited;
  - if F_i is of the form $x = y$ or $y = x$, where x, y are variables and y is limited, then x is limited;
- negation is just applied to an F_i in a maximal sub-formula F of the form $F_1 \wedge \cdots \wedge F_n$ where all free variables are limited.

The set of queries that can be expressed with safe relational calculus is denoted as $\mathcal{SRC}$.

Theorem 1.7. $\mathcal{RA} = \mathcal{SRC}$ *(cf. [26, 84]).*

Conjunctive Queries

Conjunctive queries are a natural class of queries frequently arising that enjoy different desirable properties (e.g. checking for equivalence and containment of conjunctive queries is decidable) [23]. They may be expressed in the languages seen before in the following sense:

- $\mathcal{RC}$: relational calculus expressions of the form $\{\mathbf{w} \mid \exists \mathbf{x} R_1(\mathbf{u}_1) \wedge \cdots \wedge R_k(\mathbf{u}_k)\}$, where $\mathbf{w}$ is the tuple of variables (that must appear in the conjunction) and constants, $\mathbf{x}$ is the tuple of variables in the conjunction that are not in $\mathbf{w}$, the R_i's are relation names, and the $\mathbf{u}_i$'s are tuples of terms (i.e. variables and constants).
- $\mathcal{RA}$: relational algebra expressions using only positive selection (i.e. selection conditions are restricted to be conjunctions of equalities), projection and cartesian product.

1.2 Incomplete Databases

In this section we talk about the syntax and semantics of incomplete databases and the principled semantics for querying them. Then

we discuss about the most common representation systems used to model incomplete information. Finally, we illustrate the behaviour of SQL in the presence of null values.

1.2.1 Syntax and Semantics

We assume the existence of the following disjoint countably infinite sets: a set $\texttt{Const}$ of constants and a set $\texttt{Null}$ of *labelled nulls*. $\texttt{Const}$ is the database domain, as introduced in Subsection 1.1.1. The set $\texttt{Null}$ is defined as follows: $\texttt{Null} = \{\bot_i \mid i \in \mathbb{N}\}$ (thus, nulls are denoted by the symbol $\bot$ subscripted). A *valuation* is a mapping $\nu : \texttt{Null} \cup \texttt{Const} \to \texttt{Const}$, such that $\nu(c) = c$ for every $c \in \texttt{Const}$. Hence, a valuation maps each constant to itself and every labelled null to a constant.

A tuple t of arity k is an element of $(\texttt{Const} \cup \texttt{Null})^k$, where k is a non-negative integer. The i-th element of t is denoted as $t[i]$, where $1 \leq i \leq k$. Given a possibly empty ordered sequence Z of integers $i_1, \ldots, i_h$ in the range $[1..k]$, $t[Z]$ denotes the tuple $\langle t[i_1], \ldots, t[i_h] \rangle$. For simplicity, a tuple $\langle u \rangle$ of arity 1 is simply written as u, where u is a constant or a null. A relation of arity k is a finite set of tuples of arity k. A relational schema is a set of relation names, each associated with a non-negative arity. A database D associates a relation R^D of arity k with each relation name R of arity k. A relation (resp. database) is *complete* if it does not contain nulls. Databases as defined above have been called also *naive tables*, *V-tables* and *e-tables* [61, 1, 43]. Valuations can be applied to tuples, relations, and databases in the obvious way. For instance, the result of applying ν to a database D, denoted $\nu(D)$, is the complete database obtained from D by replacing every null $\bot_i$ with $\nu(\bot_i)$. The semantics of an incomplete database D is given by the set of complete databases $poss(D) = \{\nu(D) \mid \nu \text{ is a valuation}\}$, which are also called *possible worlds*. Indeed, this is the semantics under the missing value interpretation of nulls (i.e., every null stands for a value that exists but is unknown), and is referred to as the closed-world semantics of incompleteness (CWA for short, which stands for Closed-World Assumption). Under the open-world semantics of incompleteness (OWA for short), which we do not consider, the semantics of D is $\{\nu(D) \cup D' \mid \nu \text{ is a valuation and } D' \text{ is a complete database}\}$. In the following of the thesis we will consider just the CWA.

1.2.2 Query answering

We consider queries expressed in relational algebra, that is, by means of the following operators: selection σ, projection π, Cartesian product $\times$, union $\cup$, intersection $\cap$, and difference $-$. We introduced these operators in Section 1.1.2. Thus, a query is a relational algebra expression built from relation names and the operators above. A query Q returning k-tuples is said to be of arity k, and $ar(Q)$ denotes its arity.

As an incomplete database provides different states of the real world, a query can return a set of answers for each possible world.

Definition 1.8. Given a query Q and an incomplete database D, the result of evaluating Q over D is $Q(D) = \{Q(D') \mid D' \in D\}$.

Therefore, $Q(D)$ contains a set of query answers for every possible world of D. However, there may be some tuples that are answers to Q regardless of the possible world which is the true state of the world. On the other hands, there may be tuples which are answers to Q w.r.t. some, but not necessarily each possible world.

A tuple is a *possible* answer to Q w.r.t. D if it is an answer to Q in some possible world of D.

Definition 1.9. The set of *possible answers* to a query Q w.r.t. an incomplete database D is defined as follows:

$$\texttt{possible}(Q, D) = \bigcup_{D' \in D} Q(D')$$

The evaluation of a query Q on a database D, treating nulls as standard constants (i.e., every labeled null or constant is equal to itself and different from every other element of $\texttt{Const} \cup \texttt{Null}$) is called *naive evaluation*, and it has been studied in [61].

A widely accepted semantics of query answering relies on the notion of *certain answers*. A tuple is a *certain* answer to a query Q w.r.t. an incomplete database D if it is an answer to Q in each possible world of D.

Definition 1.10. The set of *certain answers* to a query Q w.r.t. an incomplete database D is defined as follows:

$$\texttt{certain}(Q, D) = \bigcap_{D' \in D} Q(D').$$

As an alternative to the definition given above, an equivalent definition is: $\texttt{certain}(Q, D) = \cap\{Q(\nu(D))|\nu \text{ is a valuation}\}$

Computing certain answers is coNP-hard (data complexity), even in the case of Codd tables, that is, databases where the same null cannot occur multiple times [3] (for more details about Codd tables, see later).

For query answering, there exists a more general notion first proposed in [75] and called *certain answers with nulls* in [72], which avoids some anomalies of certain answers (we provide an example below). The certain answers with nulls to a query Q on a database D, denoted by $\texttt{cert}(Q, D)$, are all tuples t such that $\nu(t) \in Q(\nu(D))$ for every valuation ν. Comparing the standard notion of certain answers with the notion of certain answers with nulls, we have that the latter always include the former and may additionally include tuples with nulls (certain answers never contain nulls). Indeed, the certain answers are exactly the null-free tuples in the set of certain answers with nulls [72]. Certain answers with nulls avoid some anomalies of certain answers, such as returning a subset of a relation when the query is the "identity query". As an example, consider the following database D:

P
john
mary

E
john
$\perp_1$

S
mary
bob

and the queries $Q = P - E$ and $Q' = E$ (identity query). We have that: (i) the naive evaluation gives $Q(D) = \{\texttt{mary}\}$ and $Q'(D) = \{\texttt{john}, \perp_1\}$, as $\perp_1$ is treated as a constant; (ii) the certain answers with nulls are $\texttt{cert}(Q, D) = \emptyset$ and $\texttt{cert}(Q', D) = \{\texttt{john}, \perp_1\}$; and (iii) the certain answers to Q and Q' are $\emptyset$ and $\{\texttt{john}\}$, respectively. Notice that $Q(D)$ contains mary, which is neither a certain answer nor a certain answer with nulls. So the naive evaluation may return false positives. Moreover, notice that $\texttt{cert}(Q', D) = \{\texttt{john}, \perp_1\}$ while the certain answers to Q' consist only of john, which is indeed the only tuple in $\texttt{cert}(Q', D)$ without nulls. (We refer the interested reader to [68, 70] for a discussion of other drawbacks of the standard notion of certain answers.)

A query evaluation algorithm is said to have correctness guarantees for a query Q if for every database D it returns a subset of

$\texttt{cert}(Q, D)$. When a query evaluation algorithm has correctness guarantees for every query, we say that it has correctness guarantees. Notice that dealing with $\texttt{cert}(Q, D)$ allows us to deal with certain answers too: as recalled before, certain answers can be obtained from $\texttt{cert}(Q, D)$ by deleting tuples containing nulls.

1.2.3 Representation Systems

A number of approaches, called *representation systems*, have been proposed to represent incomplete databases compactly. In the following we present representation systems based on unknown null values [7, 15, 25, 44, 60, 59, 67, 73, 74, 88, 57, 56, 55]. Some representation systems such as c-tables may specify some restrictions on how nulls can be replaced.

We now recall the important notions of *strong* and *weak* representations system. Then, we illustrate a number of representation systems and show how the different ones behave w.r.t. such properties.

Given a representation T (in a particular representation system) of an incomplete database, we use $rep(T)$ to denote the set of complete databases represented by T (or, in other words, by the incomplete database represented by T). Give a query Q, we would always like to be able to find a representation of the answers to Q over the incomplete database represented by T, using the same representation system of T. More precisely, for every query Q and representation T of an incomplete database, we would like to compute a representation T' (from T and Q) such that $rep(T') = Q(rep(T))$. If a representation system has this property for a query language $\mathcal{L}$, then it is said to be a *strong* representation system for $\mathcal{L}$.

We now present the notion of a *weak* representation system by relaxing the requirements of a strong representation system.

Given a query language $\mathcal{L}$, we say that two incomplete databases I and J are $\mathcal{L}$-*equivalent*, denoted $I \equiv_{\mathcal{L}} J$, if $\texttt{certain}(Q, I) = \texttt{certain}(Q, J)$ for each query Q of $\mathcal{L}$. A representation system is *weak* for a query language $\mathcal{L}$ if for each representation T of an incomplete database and query Q of $\mathcal{L}$ there exists a representation T' such that $rep(T') \equiv_{\mathcal{L}} Q(rep(T))$. Unlike a strong representation system, a weak representation system is not required to be able to represent $Q(rep(T))$ for each query Q and representation T; nevertheless, a weak representation system must be able to provide a representation T' s.t. $rep(T')$ is $\mathcal{L}$-equivalent to $Q(rep(T))$ - this means that $rep(T')$

and $Q(rep(T))$ are indistinguishable as long as we are interested only in the certain answers to queries in $\mathcal{L}$.

Now we present four different representation systems. To simplify the presentation, we restrict our discussion to unirelational databases, and assume that each attribute domain coincides with the database domain *Const*. Generalization is easy to derive.

Codd tables

A *Codd table* is a relation possibly containing labelled nulls from `Null`, where each labelled null can occur at most once. The incomplete database represented by a Codd table T is defined as follows:

$$rep(T) = \{\nu(T) \mid \nu \text{ is a valuation}\}.$$

Therefore, the possible worlds represented by T are the complete databases which can be derived from T by replacing every labelled null in T with a constant. It is important to underline that the previous definition of $rep(T)$ assumed the Closed World Assumption (CWA) because every tuple in a possible world of $rep(T)$ must be derived from a tuple of T. Instead, in the presence of the Open World Assumption (OWA), the possible worlds represented by T include $rep(T)$ and any other complete database that contains a database in $rep(T)$.

Example 1.11. Assume $\perp_1$, $\perp_2$, $\perp_3$, $\perp_4$ are labelled nulls in `Null`. The following is a Codd table:

A_1	A_2	A_3
2	2	$\perp_1$
$\perp_2$	$\perp_3$	0
0	1	$\perp_4$

The following relations are some of the possible worlds represented by the previous Codd table:

A_1	A_2	A_3
2	2	1
1	1	0
0	1	2

A_1	A_2	A_3
2	2	2
1	2	0
0	1	1

A_1	A_2	A_3
2	2	2
4	2	0
0	1	0

as an example, the first relation above is obtained from Codd table by means of a valuation ν s.t. $\nu(\perp_1) = 1$, $\nu(\perp_2) = 1$, $\nu(\perp_3) = 1$, $\nu(\perp_4) = 2$. Under the OWA, the following is also one of the possible worlds (because it is a superset of the first possible world reported above):

A_1	A_2	A_3
2	2	1
1	1	0
0	1	2
0	0	1

As illustrated in the following example, Codd tables are not a strong representation system even for restricted subsets of relational algebra. □

Example 1.12. To give an idea of why Codd tables are not a strong representation system for different subsets of relational algebra, consider the Codd table of Example 1.11, call it T and the simple query Q defined as $\sigma_{A_1=5}(T)$. Clearly, $Q(rep(T))$ contains an empty relation (this is obtained, for instance, by evaluating Q over the first possible world reported in Example 1.11) and at least one non-empty relation (e.g. the one obtained by evaluating Q over the third possible world reported in Example 1.11). It may be easily verified that there in no Codd table T' whose possible worlds contain the two aforementioned relations. □

Codd tables are a weak representation system for the subset of relational algebra just consisting of selection (involving equalities and inequalities) and projection. If we consider a language that allows join or union too, the Codd tables are no longer a weak representation system for such a language.

Naive tables

One of the limitations of Codd tables is that a labelled null can appear in the database at most once. *Naive tables* remove this constraint and are defined as Codd tables except that labelled nulls are allowed to occur more than once.

The set of possible worlds represented by a naive table T is defined similarly as done for Codd tables, that is:

$$rep(T) = \{\nu(T) \mid \nu \text{ is a valuation}\}.$$

Notice that if a naive table contains multiple occurrences of the same labelled null, the possible worlds are obtained by replacing the different occurrences of the same labelled null with the same constant.

Example 1.13. Assume $\perp_1$, $\perp_2$, $\perp_3$ are labelled nulls in `Null`. The following is a naive table but not a Codd table, because of the two occurrences of $\perp_1$:

A_1	A_2	A_3
2	2	$\perp_1$
$\perp_2$	$\perp_3$	0
0	1	$\perp_1$

Notice that the naive table above says that even if the A_3-values of the first and third tuples are unknown, we know that they are the same.

The following relations are some of the possible worlds represented by the previous naive table:

A_1	A_2	A_3
2	2	1
1	1	0
0	1	1

A_1	A_2	A_3
2	2	3
1	2	0
0	1	3

A_1	A_2	A_3
2	2	1
3	2	0
0	1	1

As an example, the first relation above is obtained from the naive table by means of a valuation ν s.t. $\nu(\perp_1) = 1, \nu(\perp_2) = 1, \nu(\perp_3) = 1$. We remark again that for each possible world of the previous naive table the C-value of the first tuple is equal to the C-value of the third tuple. □

Naive tables are a weak representation system for $\mathcal{RA}$ queries using selection (where only equalities are allowed), projection, join and union. For a query Q in this class, the certain answers to Q w.r.t. the incomplete database represented by a naive table T may be computed as follows: first, Q is evaluated over T in the standard way by treating labelled nulls as new constants different from any constant in the database domain; then, tuples in the result containing labelled nulls are discarded and the remaining tuples are certain answers.

Conditional tables

Until now we have seen that neither Codd nor naive tables are strong representation system for full $\mathcal{RA}$. We now present a much more

powerful representation system, called *conditional tables*, that forms a strong representation system for $\mathcal{RA}$. Conditional tables have been proposed in [61].

A *condition* is a conjunction of atoms of the form $\perp_i = \perp_j$, $\perp_i = c$, $\perp_i \neq \perp_j$ or $\perp_i \neq c$, where $\perp_i$ and $\perp_j$ are labelled nulls and c is a constant. A valuation ν *satisfies* a condition ϕ iff by replacing each occurrence of labelled null $\perp_i$ in ϕ with $\nu(\perp_i)$, the resulting logical expression is true. A *conditional table* (*c-table* for short) is a triple $\langle T, \Phi, \Psi \rangle$ where T is a naive table, Φ is a condition (called *global condition*) and Ψ is a function mapping each tuple of T to a condition (conditions associated with tuples by means of function Φ are called *local conditions*). Global and local conditions may contain labelled nulls not occurring in T.

The set of possible worlds represented by a conditional table $\langle T, \Phi, \Psi \rangle$ is defined as follows:

$$rep(\langle T, \Phi, \Psi \rangle) = \{r \mid \text{there exists a valuation } \nu \text{ such that } \nu \text{ satisfies } \Psi \text{ and } r = \{\nu(t) \mid t \in T \text{ and } \nu \text{ satisfies } \Psi(t)\}\}.$$

Notice that the previous definition of *rep* adopts the CWA.

Example 1.14. Suppose we know that Bill is taking chemistry or physics, but not both, and another course that is not known. Tom takes biology if Bill takes chemistry, and chemistry or computer science (CS), but not both, if Bill takes CS. This can be represented by the following c-table:

$\perp_1 \neq$ `chemistry` $\wedge$ $\perp_1 \neq$ `physics`

Student	Course	
Bill	`chemistry`	$\perp_2 = 0$
Bill	`physics`	$\perp_2 \neq 0$
Bill	$\perp_1$	
Tom	`biology`	$\perp_2 = 0$
Tom	`chemistry`	$\perp_1 =$ `CS` $\wedge$ $\perp_3 = 0$
Tom	`CS`	$\perp_1 =$ `CS` $\wedge$ $\perp_3 \neq 0$

In the previous c-table $\perp_1$, $\perp_2$ and $\perp_3$ are labeled nulls. The global conditions Φ is $\perp_1 \neq$ `chemistry` $\wedge$ $\perp_1 \neq$ `physics`. For each tuple the condition associated by Ψ is reported in the last column (a missing condition for a tuple t means that $\Psi(t) =$ `true`).

The following relations are some of the possible worlds represented by the previous c-table:

Student	Course
Bill	chemistry
Bill	CS
Tom	biology
Tom	chemistry

Student	Course
Bill	chemistry
Bill	biology
Tom	biology

Student	Course
Bill	physics
Bill	biology

As an example, the first relation above is obtained from the c-table by means of a valuation ν s.t. $\nu(\perp_1) = \texttt{CS}$, $\nu(\perp_2) = 0$, $\nu(\perp_3) = 0$.

A valuation ν such that $\nu(\perp_1) = \texttt{chemistry}$, $\nu(\perp_2) = 1$ would lead to the following relation:

Student	Course
Bill	physics
Bill	chemistry

Furthermore this relation is not a possible world because ν does not satisfy the global condition. □

C-tables are a strong representation for $\mathcal{RA}$ [2].

Both local and global conditions may be `true`. From here on we will consider conditional tables with no global condition (that is with a global condition equal to `true`: for convenience, if the global condition is `true`we can omit it). We denote such a conditional table simply with T, in place of the triple $\langle T, \Phi, \Psi \rangle$.

Thus, essentially a c-table is a relation extended by one additional special column (which cannot be used inside queries) containing logical formulae, specifying under which conditions tuples are true. Formally, let $\mathcal{E}$ be the set of all expressions (or conditions) that can be built using the standard logical connectives $\wedge$, $\vee$, and $\neg$ with expressions of the form `true`, `false`, $(\alpha = \beta)$, and $(\alpha \neq \beta)$, where $\alpha, \beta \in \texttt{Const} \cup \texttt{Null}$. We say that a valuation ν satisfies a condition ϕ, denoted $\nu \models \phi$, if its assignment of constants to nulls makes ϕ true.

We can define a *conditional tuple* (c-tuple for short) $\mathbf{t}$ of arity k ($k \geq 0$) as a pair $\langle t, \phi \rangle$, where t is a tuple of arity k and $\phi \in \mathcal{E}$. Notice that ϕ may involve nulls and constants not necessarily appearing in t, e.g., t is the tuple $\langle a, \perp_1 \rangle$ and ϕ is the condition $(\perp_2 = c) \wedge (\perp_1 \neq \perp_3)$. So a conditional table of arity k is a finite set of c-tuples of arity k. A conditional database C associates a c-table R^C of arity k with each relation name R of arity k.

So the result of applying a valuation ν to a c-table T is $\nu(T) = \{\nu(T) \mid \langle t, \phi \rangle \in \text{t and } \nu \models \phi\}$. Thus, $\nu(T)$ is the complete relation obtained from T by keeping only the c-tuples in T whose condition

is satisfied by ν, and applying ν to such c-tuples. As an example, consider the conditional table $T = \{\langle \bot_1, \bot_1 = \texttt{mary} \rangle\}$ and two valuations ν_1 and ν_2 such that $\nu_1(\bot_1) = \texttt{mary}$ and $\nu_2(\bot_1) = \texttt{john}$. Then, $\nu_1(\mathrm{t}) = \{\texttt{mary}\}$, as by replacing $\bot_1$ with `mary` the condition $\texttt{mary} = \texttt{mary}$ is true, whereas $\nu_2(\mathrm{t}) = \emptyset$, as by replacing $\bot_1$ with `john` the condition $\texttt{john} = \texttt{mary}$ is false. The set of complete relations represented by T is $rep(\mathrm{t}) = \{\ \nu(T) \mid \nu \text{ is a valuation}\}$. Likewise, a conditional database $C = \{T_1, \ldots, T_m\}$ represents the following set of complete databases: $rep(\mathrm{C}) = \{\ \{\nu(T_1), \ldots, \nu(T_m)\} \mid \nu \text{ is a valuation}\}$.

Conditional evaluation We now illustrate the conditional evaluation of a query over a conditional database ([61, 43]). Basically, it consists in evaluating relational algebra operators so that they can take c-tables as input and return a c-table as output. The conditional evaluation of a query over a conditional database is then obtained by applying the conditional evaluation of each operator. Following [61, 43], we provide definitions of base operators only (i.e., projection, selection, union, difference, and Cartesian product). Other operators can be expressed in terms of the base ones. Let T_1 and T_2 be c-tables of arity n and m, respectively. In the definitions below, for the union and difference operators it is assumed that $n = m$. For projection, Z is a possibly empty ordered sequence of integers in the range $[1..n]$. For selection, θ is a Boolean combination of expressions of the form $(\$i = \$j)$, $(\$i = c)$, $(\$i \neq \$j)$, $(\$i \neq c)$, where $1 \leq i, j \leq n$, and $c \in \texttt{Const}$. In the following, given two tuples t_1 and t_2 of arity n, we use $(t_1 = t_2)$ as a shorthand for the condition $\bigwedge_{i \in [1..n]} (t_1[i] = t_2[i])$. The conditional evaluation of a relational algebra operator op is denoted as $\dot{op}$ and is defined as follows.

- Projection: $\dot{\pi}_Z(T_1) = \{\langle t[Z], \phi \rangle \mid \langle t, \phi \rangle \in T_1\}$.
- Selection: $\dot{\sigma}_\theta(T_1) = \{\langle t, \phi' \rangle \mid \langle t, \phi \rangle \in T_1 \text{ and } \phi' = \phi \wedge \theta(t)\}$, where $\theta(t)$ is the condition obtained from θ by replacing every $\$i$ with $t[i]$.
- Union: $T_1 \dot{\cup} T_2 = \{\langle t, \phi \rangle \mid \langle t, \phi \rangle \in T_1 \text{ or } \langle t, \phi \rangle \in T_2\}$.
- Difference: $T_1 \dot{-} T_2 = \{\langle t_1, \phi' \rangle \mid \langle t_1, \phi_1 \rangle \in T_1 \text{ and } \phi' = \phi_1 \wedge \phi_{t_1, T_2}\}$, where $\phi_{t_1, T_2} = \bigwedge \langle t_2, \phi_2 \rangle \in T_2 \neg(\phi_2 \wedge (t_1 = t_2))$.
- Cartesian product: $T_1 \dot{\times} T_2 = \{\langle t_1 \circ t_2, \phi_1 \wedge \phi_2 \rangle \mid \langle t_1, \phi_1 \rangle \in \mathrm{T}_1, \langle t_2, \phi_2 \rangle \in T_2\}$, where $t_1 \circ t_2$ is the tuple obtained as the concatenation of t_1 and t_2.

The result of the conditional evaluation of a query Q over a conditional database C is denoted as $\dot{Q}(C)$. Notice that $\dot{Q}(C)$ is a c-table. For a fixed query Q and a conditional database C, $\dot{Q}(C)$ can be evaluated in polynomial time in the size of C (see [43]). The size of a conditional table T is $\|T\| = |T| + \sum_{\langle t,\phi\rangle \in T} \|\phi\|$, where $|T|$ is the number of c-tuples in T and $\|\phi\|$ is the length in symbols of condition ϕ. The size of a conditional database C is $\|C\| = \sum_{T \in C} \|T_i\|$. Recall that c-tables are a strong representation system for relational algebra [2], that is, for every relational algebra query Q and conditional database C, it is possible to compute a c-table T s.t. $rep(T) = \{Q(D) \mid D \in rep(C)\}$. Indeed, such a c-table is computed precisely by conditional evaluation, that is, $T = Q(C)$. From here on we assume that every selection condition is a conjunction of expressions of the form ($\$i = \$j$), ($\$i = c$), ($\$i \neq \$j$), and ($\$i \neq c$). There is no loss of generality, as an arbitrary selection $\sigma\theta(\mathrm{R})$ can be rewritten as follows to comply with our assumption. First, θ is rewritten in disjunctive normal form (DNF), that is, into a formula $\theta_1 \vee \cdots \vee \theta_m$, where each θ_i is a conjunction of expressions of the form ($\$i = \$j$), ($\$i = c$), ($\$i \neq \$j$), and ($\$i \neq c$). Then, $\sigma_\theta(R)$ is replaced by $\sigma_{\theta_1}(R) \cup \cdots \cup \sigma_{\theta_m}(R)$. Even though the conversion to DNF can lead to an exponential blow-up in size, this does not affect data complexity (as the query is fixed).

Horn tables

A *Horn table* is a c-table where conditions are constrained to assume a restricted form. Particularly, *Horn conditions* have one of the following forms:

- $\perp_i = c$ and $\perp_i = \perp_j$ are (atomic) Horn conditions. Here $\perp_i$ and $\perp_j$ are labelled nulls whereas c is a constant;
- $\neg F_1 \vee \cdots \vee \neg F_n$ is a Horn condition - here the F_i's are atomic Horn conditions;
- $F_1 \wedge \cdots \wedge F_n \rightarrow F_{n+1}$ is a Horn condition. Here the F_i's are atomic Horn conditions.

A c-table is a Horn table if the global condition is a Horn condition and local conditions are of the form $F_1 \wedge \cdots \wedge F_n$.

Example 1.15. The c-table of Example 1.14 is not a Horn table. One reason is that the global condition is not Horn. Moreover, the local conditions of the second and last tuples are not Horn too. □

1.2.4 Nulls in SQL

The SQL standard provides one single constant NULL to represent a missing value. Generally, the precise behaviour of the NULL value in SQL is not described in detail, as the SQL rules surrounding NULL may be ambiguous, often not intuitive and in some case astonishing. The way NULLs should be handled in SQL, in each scenarios, is not fully explained in the standard documentation.

In SQL, a NULL denotes that the value is unknown. Notice that a NULL occurrence is different from the value zero, the empty string and even from other NULL occurrences, namely two occurrences of NULL are not necessarily "hiding" the same value. In fact, any comparison between a NULL and any other value - a constant or another NULL - produces the unknown truth value because the value of a NULL is unknown. Therefore, in the presence of NULLs, SQL considers a three-valued logic where the truth values are false, unknown and true. As an example, given two relations $r_1 = \{(c_1, \texttt{NULL}), (c_2, 1), (c_3, 2)\}$ and $r_2 = \{(c_1, \texttt{NULL})\}$ with schemas $R_1(A_1, A_2)$ and $R_2(A_3, A_4)$, the join of r_1 and r_2 with join condition $A_2 = A_4$ gives in output an empty relation, while the union of r_1 and r_2 is equal to $\{(c_1, \texttt{NULL}), (c_2, 1), (c_3, 2)\}$. Furthermore, the selection of the tuples of r_1 satisfying the condition $A_2 = 1$ gives the relation $\{(c_2, 1)\}$, whereas if the condition is $A_2 \neq 1$ we get the relation $\{(c_3, 2)\}$, as NULL cannot be considered to be equal to 1, but cannot even be considered to be different from 1, hence both comparisons yield the truth value unknown and the first tuple of r_1 is not incorporated in the result. Indeed, only tuples whose comparison yields true are incorporated in the result.

Under the linear ordering false $<$ unknown $<$ true defined over the truth values, the semantics of the logical operators $\wedge$ and $\vee$ does not change as $A_1 \wedge A_2 = \min\{A_1, A_2\}$ and $A_1 \vee A_2 = \max\{A_1, A_2\}$; the meaning of the negation operators must be extended assuming that $\neg$unknown $=$ unknown is true. Furthermore, arithmetic operators involving Nulls return as result a NULL. Although it is reasonable that NULL $+ 100 =$ NULL, surprisingly NULL $\times 0 =$ NULL. Even more astonishing is the following scenarios: if we count the tuples in the above relation r_1, the result is 3 (SELECT COUNT(*) FROM R_1), but if we count the tuples in the relation obtained by projecting r_1 over attribute A_2, the result is 2 (SELECT COUNT(A_2) FROM R_1) and if we sum the values in the second column of r_1 the result is 3 (SELECT SUM(A_2) FROM R_1).

Moreover, [72] has evidenced that SQL's evaluation may produce unexpected results w.r.t. the semantics of certain answers. Firstly, SQL can miss some certain answers, thus producing false negatives

(a false negative answer is a certain answer that should be returned by SQL evaluation but it does not); secondly, SQL may return some tuples that are not certain answers, thus producing false positives (a false positive answer is a tuple that is returned by SQL evaluation and yet is not certain). This behaviour has been reported in Example 2.2.

A formal semantics for SQL null values that exactly captures the behaviors of SQL queries in the presence of nulls has been proposed in [38, 39].

A knowledge base (KB) is a pair (D, Σ), where D is a database and Σ is an acyclic set of EGDs.

Chapter 2

Approximate query answering over incomplete databases: state of the art

Incomplete databases have been investigated for several decades. The seminal paper of Imielinski and Lipski [61] introduced the formal foundations for databases with nulls representing missing values and the notion of representation system. This work laid the foundation for the development of an exciting new research field, where several problems have been investigated over the years, such as query answering, dependency satisfaction, and handling updates [61, 1, 85].

Certain answers are a principled manner to define the semantics of query answering [3, 2]. However, their computational complexity is high (coNP-hard in data complexity [3]).

Relational DBMSs rely on the three-valued semantics of SQL, but unfortunately this can yield incorrect answers, as both false positives and false negatives to certain answers can be produced (see [54]).

A renewed interest in the field has recently arisen [70] due to the relevance of incomplete and uncertain data in applications such as data integration [66], data exchange [6, 32, 33], inconsistent databases [9, 48, 53, 62, 63, 78, 86], data cleaning and ontology-based query answering [12, 21, 22, 8, 14, 31, 79].

Recently, a number of approximation algorithms for query answering have been proposed: [72, 71] proposed the *L-approach*, [54] proposed the *GL-approach*, [49] proposed the *GMT-approaches*. We will discuss all these approaches later in this chapter.

The L-approach consists in rewriting a query Q into queries Q^t and Q^f computing approximate sets of certainly true and certainly false answers, respectively. Even if this technique has good theoretical complexity (AC^0), it suffers from a number of problems that

severely hinder its practical implementation [54]. In particular, it requires the computation of active domains and, even worse, their Cartesian products. While expressible in relational algebra, the execution of the Q^f translations for selection, Cartesian product, projection, and even base relations becomes prohibitively expensive, as they require computing Cartesian products of the active domain.To overcome these problems, the GL-approach, which has correctness guarantees too, produces queries that can be computed more efficiently. As we will see next, for a query Q, GL-approach computes approximations of certainly true answers and possibly true answers, rather than computing certainly true and certainly false answers. As shown in the following, the simplest of GMT-approaches is equivalent to the GL-approach, while the other ones provide better approximations (i.e., they return strictly more certain answers) and retain polynomial time data complexity.

The central tools of GMT-approaches are conditional tables and the conditional evaluation of queries. Thus, different strategies to evaluate conditions are presented, leading to different approximation algorithms [49]. More accurate evaluation strategies have higher running times, but they pay off with more certain answers being returned.

2.1 Introduction

Incomplete information arises naturally in many database applications, such as data integration [50, 30], data exchange [66, 6], inconsistency management [32, 5, 9, 47], data cleaning [42, 58], ontological reasoning [64, 42, 58], and many others. A principled way of answering queries over incomplete databases is to compute certain answers [68, 61], which are query answers that can be obtained from all the complete databases represented by an incomplete database. This notion is illustrated below.

Example 2.1. Consider the database D consisting of the following three unary relations, whose names are P (Person), S (Student) and E (Employee), where $\perp_1$ is a null value.

P
john
mary

E
john
$\perp_1$

S
mary
bob

Under the missing value interpretation of nulls (i.e., a value for $\perp_1$ exists but is unknown), D represents all the databases obtained by

replacing $\perp_1$ with an actual value (in replacing nulls, we assume the CWA). A certain answer to a query is a tuple that is an answer to the query for every database represented by D. For instance, consider the query asking for the people who are not employed students, which can be expressed in relational algebra as $P - (E \cap S)$. The certain answers to the query are $\{\langle \texttt{john} \rangle\}$, because no matter how $\perp_1$ is replaced, $\langle \texttt{john} \rangle$ is always a query answer. □

Notice that `mary` is not a certain answer because when $\perp_1$ takes the value `mary`, she is not a query answer. Notice that a certain answer never contains a labeled null. For databases containing (labeled) nulls, certain answers to positive queries can be easily computed in polynomial time by using the so-called *naive evaluation* [61], which works as follows: first, the query is evaluated in the usual way treating nulls as standard constants, and then tuples containing nulls are discarded from the result of the query evaluation. However, for more general queries with negation the problem becomes coNP-hard in data complexity [3]. In fact, the naive evaluation does not work with queries involving negation. As an example, consider the query and the database of Example 2.1 above. Clearly, the evaluation of $(E \cap S)$ gives the empty set and thus $P - (E \cap S)$ gives both `john` and `mary`, but the latter is not a certain answer to the query. To make query answering feasible in practice, one might resort to SQL's evaluation, but unfortunately, the way SQL behaves in the presence of nulls may result in wrong answers. Notice that for SQL evaluation every labeled null in the database should be replaced by the usual `NULL` of SQL. Specifically, as evidenced in [72], there are two ways in which certain answers and SQL's evaluation can differ: (i) SQL can miss some certain answers, thus producing false negatives; or (ii) SQL can return some tuples that are not certain answers, that is, false positives. An example of the second case is illustrated below.

Example 2.2. Consider again the database D of Example 2.1. There are no certain answers to the query $P - E$, as the query answers are the empty set when $\perp_1$ is replaced with `mary`. Assuming that P and E's attribute is called name, the same query can be expressed in SQL as follows:

```
SELECT P.name
FROM P
WHERE NOT EXISTS (
        SELECT *
        FROM E
```

```
WHERE P.name = E.name )
```

Unfortunately, the evaluation of the SQL query above returns $\langle$`mary`$\rangle$, which is not a certain answer. The problem with the SQL semantics is that every comparison involving at least one null evaluates to the truth value unknown, then 3-valued logic is used to evaluate the classical logical connectives (*AND, OR, NOT*), and eventually only those tuples whose condition evaluates to true are kept. Going back to the query above, for the first tuple of P, namely `john`, the nested subquery finds the same tuple in E, and thus `john` is not returned. For the second tuple of P, namely `mary`, the nested subquery first compares `mary` with `john` and the comparison evaluates to false, it then compares `mary` with $\perp_1$ and the comparison evaluates to unknown (because a null is involved), and thus the result of the nested subquery is empty. As a consequence, $\langle$`mary`$\rangle$ is returned in the final result of the query. □

While missing some certain answers can be seen as an under-approximation of certain answers (a sound but possibly incomplete set of certain answers is returned), false positives must be avoided, as the result might contain plain incorrect answers, that is, tuples that are not certain. The experimental analysis in [54] showed that false positive are a real problem for queries involving negation; they were always present and sometimes they constitute almost 100% of the answers. Thus, SQL's evaluation is efficient but flawed. On the other hand, certain answers provide a principled semantics, but with high complexity. One way of dealing with this issue is to develop polynomial time approximation algorithms for computing (approximate) certain answers. In this regard, there has been recent works on evaluation algorithms with correctness guarantees, that is, techniques providing a sound but possibly incomplete set of certain answers [72, 54, 71].

In particular, two techniques consisting in rewriting a query Q into a pair of queries have been proposed. The first technique, namely the L-approach, was introduced in [72, 71] and rewrites Q into two queries Q^t and Q^f of computing approximations of certainly true and certainly false answers, respectively. The second technique, called GL-approach, was introduced in [54] and rewrites Q into two queries Q^+ and $Q^?$ computing approximations of certainly true and possibly true answers, respectively. [49] introduced the GMT-approaches which are new approximation algorithms able to return (strictly) more certain answers than L-approach and GL-approach. Here conditional

tables and the conditional evaluation of queries [61] are the central tools. This approach allows to keep track of useful information that can be profitably used to determine if a tuple is a certain answer. The very basic idea is illustrated in the following example.

Example 2.3. Consider again the database and the query of Example 2.1. The conditional evaluation of the query is carried out by applying the "conditional" counterpart of each relational algebra operator. Rather than returning a set of tuples, the conditional evaluation of a relational algebra operator returns "conditional tuples", that is, pairs of the form $\langle t, \phi \rangle$, where t is a standard tuple and ϕ is an expression stating under which conditions t can be derived. Regarding the query of Example 2.1, namely $P - (E \cap S)$, first the conditional evaluation of $E \cap S$ is performed, which gives the conditional tuples $\langle \bot_1, \phi_1 \rangle$ and $\langle \bot_1, \phi_2 \rangle$, where ϕ_1 is the condition $(\bot_1 = \texttt{mary})$ and ϕ_2 is the condition $(\bot_1 = \texttt{bob})$. This intuitively means that the tuple $\langle \bot_1 \rangle$ is derived when $\bot_1$ is `mary` or `bob`. Then, the conditional evaluation of the difference operator is carried out, yielding the conditional tuples $\langle \texttt{john}, \phi' \rangle$ and $\langle \texttt{mary}, \phi'' \rangle$, where ϕ' is the condition

$$\neg((\texttt{john} = \bot_1) \wedge (\bot_1 = \texttt{mary})) \wedge \neg((\texttt{john} = \bot_1) \wedge (\bot_1 = \texttt{bob})),$$

and ϕ'' is the condition

$$\neg((\texttt{mary} = \bot_1) \wedge (\bot_1 = \texttt{mary})) \wedge \neg((\texttt{mary} = \bot_1) \wedge (\bot_1 = \texttt{bob})).$$

Intuitively, ϕ' expresses that `john` is derived when it is not in the result of $E \cap S$. In order for `john` to be in the result of $E \cap S$, $\bot_1$ must be equal to `john` and equal to either `mary` or `bob` (indeed, this can never happen, and in fact `john` is a certain answer). Likewise, ϕ'' expresses that `mary` is derived when it is not in the result of $E \cap S$. In order for `mary` to be in the result of $E \cap S$, $\bot_1$ must be equal to `mary`, which might be the case (and indeed, `mary` is not a certain answer). The conditional tuples $\langle \texttt{john}, \phi' \rangle$ and $\langle \texttt{mary}, \phi'' \rangle$ are the result of the conditional evaluation of the whole query. Conditions are valuable information that can be exploited to determine which tuples are certain answers. □

As already mentioned, for a conditional tuple $\langle t, \phi \rangle$, the expression ϕ says under which condition t can be derived. By "condition evaluation", a way of associating ϕ with a truth value (true, false, or unknown) is meant. The aim is to ensure that if ϕ evaluates to

true, then t is a certain answer. A condition ϕ is always a propositional formula whose atomic formulas are comparisons involving actual values and labeled nulls, and the evaluation does not access the database. For instance, from an analysis of ϕ' in Example 2.3 above, one can realize that the condition is always true (i.e., it holds for every possible value $\perp_1$ stands for), and thus $\langle \texttt{john} \rangle$ is a certain answer. There might be different ways of evaluating conditions and at different stages of the query evaluation process. It'll be shown that a simple "eager" way consisting in evaluating conditions right after each operator leads to the approximation algorithm of [54]. Then, more refined evaluation strategies will be illustrated. Intuitively, postponing condition evaluation allows to keep more information and have a more global view, and thus perform more accurate analyses. These novel evaluation algorithms have correctness guarantees. As said, conditional tables and the conditional evaluation of relational algebra are leveraged, in order to keep track of information that can be profitably used to determine certainly true (or false or possible) answers. Tuples' conditions can be evaluated in different ways, with the aim of providing a sound (but possibly incomplete) set of certain answers. Different such strategies lead to different approximation algorithms, which provide better approximations than [54], in that strictly more certain answers can be found. Furthermore, all these algorithms are polynomial time in data complexity.

2.2 L-approach

We say that two tuples t_1 and t_2 *unify*, denoted $t_1 \Uparrow t_2$, if there exists a valuation v such that $v(t_1) = v(t_2)$.

The query computing the active domain (i.e. the set of all constants and nulls occurring in the database) is denoted as $\texttt{adom}$. Moreover, for a relation R of arity k, the *complement* is

$$R^{\ominus} = \{t \in \texttt{adom}^k \mid \nexists t' \in R \text{ s.t. } t \Uparrow t'\}.$$

The query evaluation algorithm proposed in [72, 71] works as follows:

$$
\begin{aligned}
&R^t = R\\
&(Q_1 \cup Q_2)^{\mathbf{t}} = Q_1^{\mathbf{t}} \cup Q_2^{\mathbf{t}}\\
&(Q_1 \cap Q_2)^{\mathbf{t}} = Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{t}}\\
&(Q_1 - Q_2)^{\mathbf{t}} = Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{f}}\\
&(Q_1 \times Q_2)^{\mathbf{t}} = Q_1^{\mathbf{t}} \times Q_2^{\mathbf{t}}\\
&(\sigma_\theta(Q))^{\mathbf{t}} = \sigma_{\theta^+}(Q^{\mathbf{t}})\\
&(\pi_Z(Q))^{\mathbf{t}} = \pi_Z(Q^{\mathbf{t}})
\end{aligned}
$$

$$
\begin{aligned}
&R^{\mathbf{f}} = R^{\ominus}\\
&(Q_1 \cup Q_2)^{\mathbf{f}} = Q_1^{\mathbf{f}} \cap Q_2^{\mathbf{f}}\\
&(Q_1 \cap Q_2)^{\mathbf{f}} = Q_1^{\mathbf{f}} \cup Q_2^{\mathbf{f}}\\
&(Q_1 - Q_2)^{\mathbf{f}} = Q_1^{\mathbf{f}} \cup Q_2^{\mathbf{f}}\\
&(Q_1 \times Q_2)^{\mathbf{f}} = Q_1^{\mathbf{f}} \times \mathtt{adom}^{ar(Q_2)} \cup \mathtt{adom}^{ar(Q_1)} \times Q_2^{\mathbf{f}}\\
&(\sigma_\theta(Q))^{\mathbf{f}} = Q^{\mathbf{f}} \cup \sigma_{(\neg\theta)^+}(\mathtt{adom}^{ar(Q)})\\
&(\pi_Z(Q))^{\mathbf{f}} = \pi_Z(Q^{\mathbf{f}}) - \pi_Z(\mathtt{adom}^{ar(Q)} - Q^{\mathbf{f}})
\end{aligned}
$$

where a selection condition θ is translated into θ^+ as follows:

$$
\begin{aligned}
&(\$i = \$j)^+ = (\$i = \$j)\\
&(\$i = c)^+ = (\$i = c)\\
&(\$i \neq \$j)^+ = (\$i \neq \$j) \wedge \mathtt{const}(\$i) \wedge \mathtt{const}(\$j)\\
&(\$i \neq c)^+ = (\$i \neq c) \wedge \mathtt{const}(\$i)\\
&(\theta_1 \vee \theta_2)^+ = \theta_1^+ \vee \theta_2^+\\
&(\theta_1 \wedge \theta_2)^+ = \theta_1^+ \wedge \theta_2^+
\end{aligned}
$$

where $\mathtt{const}(\$i)$ is a condition saying whether $t[i]$ is a constant for a tuple t.

The approach consists in translating a query Q into two queries $Q^{\mathbf{t}}$ and $Q^{\mathbf{f}}$ computing certainly `true` and certainly `false` answers, respectively. The ultimate goal is to compute $Q^{\mathbf{t}}$, which provides a sound but possibly incomplete set of certain answers. Its definition requires the computation of $Q^{\mathbf{f}}$, as one can see taking a look at the translation of the difference operator.

2.3 GL-approach

Given two relations R and S of the same arity, the following operators are defined:

- the *left unification semijoin*:

$$R \ltimes_{\Uparrow} S = \{t \in R \mid \exists t' \in S \text{ s.t. } t \Uparrow t'\};$$

- the *left unification anti-semijoin*:

$$R \overline{\ltimes}_{\Uparrow} S = \{t \in R \mid \nexists t' \in S \text{ s.t. } t \Uparrow t'\}.$$

The evaluation algorithm of [54] consist in translating a query Q into two queries Q^+ and $Q^?$, where the evaluation of Q^+ over a database gives a subset of certain answers. The translation is reported here:

$$\begin{aligned}
&R^+ = R\\
&(Q_1 \cup Q_2)^+ = Q_1^+ \cup Q_2^+\\
&(Q_1 \cap Q_2)^+ = Q_1^+ \cap Q_2^+\\
&(Q_1 - Q_2)^+ = Q_1^+ \overline{\ltimes}_{\Uparrow} Q_2^+\\
&(Q_1 \times Q_2)^+ = Q_1^+ \times Q_2^+\\
&(\sigma_\theta(Q))^+ = \sigma_{\theta^+}(Q^+)\\
&(\pi_Z(Q))^+ = \pi_Z(Q^+)
\end{aligned}$$

$$\begin{aligned}
&R^? = R\\
&(Q_1 \cup Q_2)^? = Q_1^? \cup Q^?\\
&(Q_1 \cap Q_2)^? = Q_1^? \ltimes_{\Uparrow} Q_2^?\\
&(Q_1 - Q_2)^? = Q_1^? - Q_2^+\\
&(Q_1 \times Q_2)^? = Q_1^? \times Q_2^?\\
&(\sigma_\theta(Q))^? = \sigma_{\theta^?}(Q^?)\\
&(\pi_Z(Q))^? = \pi_Z(Q^?)
\end{aligned}$$

where a selection condition θ is translated into θ^+ as defined before for the first approach and is translated into $\theta^?$ as follows:

$$
\begin{aligned}
(\$i \neq \$j)^? &= (\$i \neq \$j)\\
(\$i \neq c)^? &= (\$i \neq c)\\
(\$i = \$j)^? &= (\$i = \$j) \vee \texttt{null}(\$i) \vee \texttt{null}(\$j)\\
(\$i = c)^? &= (\$i = c) \vee \texttt{null}(\$i)\\
(\theta_1 \vee \theta_2)^? &= \theta_1^? \vee \theta_2^?\\
(\theta_1 \wedge \theta_2)^? &= \theta_1^? \wedge \theta_2^?
\end{aligned}
$$

where $\texttt{null}(\$i)$ is a condition indicating whether $t[i]$ is a null for a tuple t.

Thus, given a query Q and a database D, the evaluation of Q^+ over D gives a subset of $\texttt{certain}(Q, D)$, whereas the evaluation of $Q^?$ over D gives "possible" answers. Notice that the definition of $(Q_1 - Q_2)^+$ uses $Q_2^?$, while the definition of $(Q_1 - Q_2)^?$ uses Q_2^+; however, when recursively applying the Q^+ and $Q^?$ translations, both of them eventually yield standard relational algebra queries. The query evaluation of the second approach improves on the one of the first approach in terms of efficiency.

The major drawback of the latter is that it needs to compute expensive Cartesian products of the active domain (i.e. the set of all constants and nulls occurring in the database), which may be a large set.

The two evaluation algorithms are incomparable in terms of the approximations they provide, as shown in the following example (none of the two approaches uses conditional tables).

Example 2.4. Let D be the database consisting of the two unary relations $R = \{a\}$ and $S = \{\bot_1\}$.

Consider the query $Q_1 = R - (S - R)$. It is easy to see that $\texttt{certain}(Q_1, D) = \{a\}$. Since

$$
\begin{aligned}
Q_1^{\mathbf{t}} &= (R - (S - R))^{\mathbf{t}} = (R^{\mathbf{t}} \cap (S - R)^{\mathbf{f}})\\
&= (R^{\mathbf{t}} \cap (S^{\mathbf{f}} \cup R^{\mathbf{t}})) = (R \cap (S^{\ominus} \cup R))
\end{aligned}
$$

then $Q_1^{\mathbf{t}}(D) = \{a\} \cap (\emptyset \cup \{a\}) = \{a\}$.

Moreover, as

$$\begin{aligned} Q_1^+ &= (R - (S - R))^+ = (R^+ \overline{\ltimes}_{\Uparrow} (S - R)^?) \\ &= (R^+ \overline{\ltimes}_{\Uparrow} (S^? - R^+)) = (R \overline{\ltimes}_{\Uparrow} (S - R)) \end{aligned}$$

then $Q_1^+(D) = \{a\} \overline{\ltimes}_{\Uparrow} (\{\bot_1\} - \{a\}) = \{a\} \overline{\ltimes}_{\Uparrow} \{\bot_1\} = \emptyset$. Hence, $Q_1^+(D) \subset Q_1^{\mathbf{t}}(D)$.

Consider now the query $Q_2 = R - (S - S)$. It is easy to see that $\texttt{certain}(Q_2, D) = \{a\}$. Since

$$Q_2^{\mathbf{t}} = (R - (S - S))^{\mathbf{t}} = R^{\mathbf{t}} \cap (S - S)^{\mathbf{f}} = R^{\mathbf{t}} \cap (S^{\mathbf{f}} \cup S^{\mathbf{t}}) = R \cap (S^{\ominus} \cup S),$$

then $Q_2^{\mathbf{t}}(D) = \{a\} \cap (\emptyset \cup \{\bot_1\}) = \emptyset$. Moreover, as

$$\begin{aligned} Q_2^+ &= (R - (S - S))^+ = R^+ \overline{\ltimes}_{\Uparrow} (S - S)^? \\ &= R^+ \overline{\ltimes}_{\Uparrow} (S^? - S^+) = R \overline{\ltimes}_{\Uparrow} (S - S), \end{aligned}$$

then $Q_2^+(D) = \{a\} \overline{\ltimes}_{\Uparrow} (\{\bot_1\} - \{\bot_1\}) = \{a\} \overline{\ltimes}_{\Uparrow} \emptyset = \{a\}$. Hence, $Q_2^{\mathbf{t}}(D) \subset Q_2^+(D)$.

Thus, [72, 71] provides a better approximation for Q_1, while [54] provides a better approximation for Q_2. □

The intersection operator is not commutative (cf. [54]), because $(Q_1 \cap Q_2)^?$ is translated into $Q_1^? \overline{\ltimes}_{\Uparrow} Q_2^?$ and the left unification semijoin takes tuples from $Q_1^?$. Furthermore, correctess guarantee hold if we replace the left unification semijoin with the right one, which keeps unifiable tuples from the second argument. However, as shown in the following example, non-commutativity can be an issue, as one order of the arguments can yield better results than the other (with either the left unification semijoin or the right one), and this depends on the contents of the database.

Example 2.5. Consider the database D consisting if the relations $R = \{a\}$, $S = \{\bot_1\}$ and $P = \{b\}$. Consider now the query $Q_1 = R - (S \cap P)$. It is easy to see that $\texttt{certain}(Q_1, D) = \{a\}$. Since

$$\begin{aligned} Q_1^+ &= (R - (S \cap P))^+ = (R^+ \overline{\ltimes}_{\Uparrow} (S \cap P)^?) \\ &= (R^+ \overline{\ltimes}_{\Uparrow} (S^? \overline{\ltimes}_{\Uparrow} P^?)) = (R \overline{\ltimes}_{\Uparrow} (S \overline{\ltimes}_{\Uparrow} P)), \end{aligned}$$

then $Q_1^+(D) = \emptyset$. However, if we consider the equivalent query $Q_2 = R - (P \cap S)$, since $Q_2^+ = (R \overline{\ltimes}_{\Uparrow} (P \overline{\ltimes}_{\Uparrow} S))$, then $Q_2^+(D) = \{a\}$.

Clearly, the right unification semijoin yields a symmetric scenario, and thus the same problem occurs. □

The previous example shows that the order of the arguments of intersections matters, but cannot know which one is better a priori (notice that scenarios symmetric to the ones shown in the example above can be obtained by swapping the content of S and P).

2.4 GML-approaches

In this section, we illustrate novel evaluation algorithms with correctness guarantees. All of them rely on the conditional evaluation of relational algebra operators, and apply a specific strategy to evaluate conditions, so that each tuple is eventually associated with a truth value (i.e., `true`, `false` or `unknown`). Tuples that are associated with the condition true are guaranteed to be certain answers with nulls. Each approach improves on the previous one by refining its evaluation of conditions. The simplest of our techniques, which evaluates conditions simply right after the application of each relational algebra operator, is equivalent to the approach of [54]. Even though all of the novel evaluation algorithms have polynomial time data complexity, we will show that as we move to more powerful techniques, the conditions that have to be managed become more complex. We will experimentally assess this aspect. Firstly, an explicit conditional evaluation for intersection is introduced. In Section 1.2.3, we illustrated the conditional evaluation of base relational algebra operators. Clearly, even though intersection was not reported, it can be expressed in terms of the other operators. Nevertheless, to ease presentation and the comparison with the evaluation algorithm in [54], a direct conditional evaluation for intersection is reported ([61, 43] do not provide an explicit definition of it). Let T_1 and T_2 be c-tables of arity n. Then, $\mathrm{T}_1 \widehat{\cap} \mathrm{T}_2 = \{\langle t_1, \phi' \rangle \mid \langle t_1, \phi_1 \rangle \in \mathrm{T}_1, \langle t_2, \phi_2 \rangle \in \mathrm{T}_2, \phi' = \phi_1 \wedge \phi_2 \wedge (t_1 = t_2)\}$. The other relational algebra operators enjoy the following property: the conditional evaluation of a query over a conditional database C yields a c-table representing the answers that we would obtain by querying each database represented by C. The following simple proposition states that this property continues to hold in the presence of the above conditional evaluation of intersection.

Proposition 2.6. *For every conditional database C and query Q possibly using selection, projection, Cartesian product, union, intersection, and difference, it holds that $rep(\dot{Q}(C)) = \{Q(D) \mid D \in rep(C)\}$.*

A generalization of c-tables is considered in order to allow also unknown as a condition. Thus, from now on, $\mathcal{E}$ is the set of all expressions that can be built using the standard logical connectives with expressions (called atomic conditions) of the form true, false, unknown, $(\alpha = \beta)$ and $(\alpha \neq \beta)$, where $\alpha, \beta \in \texttt{Const} \cup \texttt{Null}$. We will illustrate different strategies to "evaluate" conditions, that is, to reduce them to either true or false or unknown - as shown in the following, tuples having condition true are certain answers with nulls. We assume the following strict ordering $\texttt{false} < \texttt{unknown} < \texttt{true}$ and recall that $\neg\texttt{unknown} = \texttt{unknown}$. The three-valued evaluation of a condition $\phi \in \mathcal{E}$, denoted $\texttt{eval}(\phi)$, is defined inductively as follows:

- $\texttt{eval}(\alpha = \beta) = \begin{cases} \texttt{true} & \text{if } \alpha = \beta, \\ \texttt{false} & \text{if } \alpha \neq \beta \text{ and } \alpha, \beta \in \texttt{Const}, \\ \texttt{unknown} & \text{otherwise.} \end{cases}$
- $\texttt{eval}(\alpha \neq \beta) = \begin{cases} \texttt{true} & \text{if } \alpha \neq \beta \text{ and } \alpha, \beta \in \texttt{Const}, \\ \texttt{false} & \text{if } \alpha = \beta, \\ \texttt{unknown} & \text{otherwise.} \end{cases}$
- $\texttt{eval}(\phi_1 \wedge \phi_2) = \min\{\texttt{eval}(\phi_1), \texttt{eval}(\phi_2)\}$.
- $\texttt{eval}(\phi_1 \wedge \phi_2) = \max\{\texttt{eval}(\phi_1), \texttt{eval}(\phi_2)\}$.
- $\texttt{eval}(\neg\phi) = \begin{cases} \texttt{true} & \text{if } \texttt{eval} = \texttt{false}, \\ \texttt{false} & \text{if } \texttt{eval} = \texttt{true}, \\ \texttt{unknown} & \text{otherwise.} \end{cases}$
- $\texttt{eval}(v) = v$ for $v \in \{\texttt{true}, \texttt{false}, \texttt{unknown}\}$.

For convenience, in the following examples we use T, F and U in place of true, false and unknown, respectively, and simplify conditions of the form $\texttt{T} \wedge \phi$ or $\texttt{F} \vee \phi$ into ϕ. Moreover, for ease of presentation, in our examples, "useless" conditional tuples will be discarded, that is, conditional tuples of the form $\langle t, \texttt{false}\rangle$ will always be discarded, whereas conditional tuples of the form $\langle t, \texttt{unknown}\rangle$ will be discarded only if there is a conditional tuple $\langle t, \texttt{true}\rangle$ belonging to the same c-table.

2.4.1 Eager evaluation

The basic idea of the first evaluation algorithm, which is called *eager evaluation*, is to perform the three-valued evaluation of conditions after each relational algebra operator is applied, that is, after its conditional evaluation. To define eager evaluation, some auxiliary definitions are needed. Given a c-tuple $\mathbf{t} = \langle t, \phi \rangle$, with a slight abuse of notation, we use $\texttt{eval}(t)$ to denote $\langle t, \texttt{eva}(\phi) \rangle$. Likewise, given a conditional table T , $\texttt{eval}(T)$ denotes $\{\texttt{eval}(\mathbf{t}) \mid \mathbf{t} \in T\}$. Given a relation r, we define the c-table $\overline{r} = \{\langle t, \texttt{true} \rangle \mid t \in r\}$. Analogously, given a database D, we define $\overline{D}$ as the conditional database obtained from D by replacing every relation r with $\overline{r}$. The basic idea is to convert a database D into a simple conditional database $\overline{D}$ where all conditions are `true`, so that $\overline{D}$ can be used as the starting point for the conditional evaluation of queries. To define eager evaluation, we first provide the definitions below, where R is a relation name, D is a database, and Q, Q_1, and Q_2 are queries:

- $\texttt{Eval}^e(R, D) = \overline{R^D}$
- $\texttt{Eval}^e(Q_1 \cap Q_2, D) = \texttt{eval}(\texttt{Eval}^e(Q_1, D) \dot{\cap} \texttt{Eval}^e(Q_2, D))$
- $\texttt{Eval}^e(Q_1 - Q_2, D) = \texttt{eval}(\texttt{Eval}^e(Q_1, D) \dot{-} \texttt{Eval}^e(Q_2, D))$
- $\texttt{Eval}^e(Q_1 \times Q_2, D) = \texttt{eval}(\texttt{Eval}^e(Q_1, D) \dot{\times} \texttt{Eval}^e(Q_2, D))$
- $\texttt{Eval}^e(\sigma_\theta(Q), D) = \texttt{eval}(\overline{\sigma}_\theta(\texttt{Eval}^e(Q_2, D)))$
- $\texttt{Eval}^e(\pi_Z(Q), D) = \texttt{eval}(\overline{\pi}_Z(\texttt{Eval}^e(Q_2, D)))$

Finally, we define:

- $\texttt{Eval}^e_t(Q, D) = \{t \mid \langle t, \texttt{true} \rangle \in \texttt{Eval}^e(Q, D)\}$
- $\texttt{Eval}^e_p(Q, D) = \{t \mid \langle t, \phi \rangle \in \texttt{Eval}^e(Q, D) \text{ and } \phi \neq \texttt{false}\}$

Thus, $\texttt{Eval}^e_t(Q, D)$ are the tuples in $\texttt{Eval}^e(Q, D)$ associated with `true`, while $\texttt{Eval}^e_p(Q, D)$ are the tuples in $\texttt{Eval}^e(Q, D)$ associated with `unknown` or `true`. As stated more precisely in *Theorem 2.8*, $\texttt{Eval}^e_t(Q, D)$ and $\texttt{Eval}^e_p(Q, D)$ coincide with Q^+ and $\mathrm{Q}^?$, respectively, where Q^+ and $\mathrm{Q}^?$ are the translations of [54]. However, only $\texttt{Eval}^e_t(Q, D)$ is an under-approximation of certain answers with nulls.

Example 2.7. Consider the database D of Example 2.1 and the query $Q = P - E$ asking for the people who are not employees. The conditional database D includes the two c-tables $\overline{P^D} = \{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\}$ and $\overline{E^D} = \{\langle \texttt{john}, \texttt{T}\rangle, \langle \bot_1, \texttt{T}\rangle\}$. Then,

$$
\begin{aligned}
&\texttt{Eval}^e(Q, D)\\
&= \texttt{eval}(\texttt{Eval}^e(P, D) \dot{-} \texttt{Eval}^e(\texttt{E}, D))\\
&= \texttt{eval}(\overline{P^D} \dot{-} \overline{\texttt{E}^D})\\
&= \texttt{eval}(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-} \{\langle \texttt{john}, \texttt{T}\rangle, \langle \bot_1, \texttt{T}\rangle\})\\
&= \texttt{eval}(\{\langle \texttt{john}, \texttt{T} \wedge \neg(\texttt{T} \wedge \texttt{john} = \texttt{john}) \wedge \neg(\texttt{T} \wedge \texttt{john} = \bot_1)\rangle,\\
&\quad \langle \texttt{mary}, \texttt{T} \wedge \neg(\texttt{T} \wedge \texttt{mary} = \texttt{john}) \wedge \neg(\texttt{T} \wedge \texttt{mary} = \bot_1)\rangle\})\\
&= \{\langle \texttt{john}, \texttt{eval}(\texttt{T} \wedge \neg(\texttt{T} \wedge \texttt{john} = \texttt{john}) \wedge \neg(\texttt{T} \wedge \texttt{john} = \bot_1))\rangle,\\
&\quad \langle \texttt{mary}, \texttt{eval}(\texttt{T} \wedge \neg(\texttt{T} \wedge \texttt{mary} = \texttt{john}) \wedge \neg(\texttt{T} \wedge \texttt{mary} = \bot_1))\rangle\}\\
&= \{\langle \texttt{john}, \texttt{eval}(\texttt{T} \wedge \neg(\texttt{T} \wedge \texttt{T}) \wedge \neg(\texttt{T} \wedge \texttt{u}))\rangle,\\
&\quad \langle \texttt{mary}, \texttt{eval}(\texttt{T} \wedge \neg(\texttt{T} \wedge \texttt{F}) \wedge \neg(\texttt{T} \wedge \texttt{U}))\rangle\}\\
&= \{\langle \texttt{john}, \texttt{F}\rangle, \langle \texttt{mary}, \texttt{U}\rangle\}\}
\end{aligned}
$$

□

Notice that the three-valued evaluation of $\texttt{john} = \texttt{john}$ (resp. $\texttt{mary} = \texttt{john}$) is $\texttt{T}$ (resp. $\texttt{F}$) as the equality is always $\texttt{true}$ (resp. $\texttt{false}$), while the three-valued evaluation of $\texttt{john} = \bot_1$ and $\texttt{mary} = \bot_1$ is $\texttt{U}$ as the equality might hold or not depending on the value $\bot_1$ takes. Then, $\texttt{Eval}^e_t(Q, D) = \emptyset$ and $\texttt{Eval}^e_p(Q, D) = \{\texttt{mary}\}$.

Clearly, eager evaluation does not leverage much the power of c-tables, as conditions are little exploited. We will illustrate more powerful techniques, which better exploit conditions, in the following subsections. However, one interesting fact about eager evaluation is that it is equivalent to the evaluation algorithm of [54]. We refer to Section 2.3 for the formal definition of Q^+ and $Q^?$, whose aim is to compute certain and possible query answers, respectively, which is also the intended goal of $\texttt{Eval}^e_t()$ and $\texttt{Eval}^e_p()$.

Theorem 2.8. *$Q^+(D) = \mathit{Eval}^e_t(Q, D)$ and $Q^?(D) = \mathit{Eval}^e_p(Q, D)$, for every query Q and database D.*

Union. By definition, $(Q_1 \cup Q_2)^+(D) = Q_1^+(D) \cup Q^+2(D)$. By the induction hypothesis, $Q_1^+(D) = \texttt{Eval}^e_t(Q_1, D)$ and $Q_2^+(D) = \texttt{Eval}^e_t(Q_2, D)$. It can be easily verified that $\texttt{Eval}^e_t(Q_1 \cup Q_2, D) = \texttt{Eval}^e_t(Q_1, D) \cup \texttt{Eval}^e_t(Q_2, D)$, and thus the claim follows.

Intersection. By definition, $(Q_1 \cap Q_2)^+(D) = Q_1^+(D) \cap Q_2^+(D)$. By the induction hypothesis, it holds that $Q_1^+(D) = \texttt{Eval}_t^e(Q_1, D)$ and $Q_2^+(D) = \texttt{Eval}_t^e(Q_2, D)$. By definition, tuples in $\texttt{Eval}_t^e(Q_1 \cap Q_2, D)$ are those tuples in $\texttt{Eval}^e(Q_1, D)$ whose condition is `true` and for which there is the same tuple in $\texttt{Eval}^e(Q_2, D)$ with condition `true`. Thus, $\texttt{Eval}_t^e(Q_1 \cap Q_2, D) = \texttt{Eval}_t^e(Q_1, D) \cap \texttt{Eval}_t^e(Q_2, D)$.

Difference. $(Q_1 Q_2)^+(D) = Q_1^+(D) \ltimes_{\Uparrow} Q_2^?(D)$ by definition. By the induction hypothesis, $Q_1^+(D) = \texttt{Eval}_t^e(Q_1, D)$ and $Q_2^?(D) = \texttt{Eval}_p^e(Q_2, D)$. It is easy to see that the definition of $\dot{}$ implies that $\texttt{Eval}_t^e(Q_1 Q_2, D)$ contains the tuples in $\texttt{Eval}^e(Q_1, D)$ whose condition is `true` and s.t. there is no tuple in $\texttt{Eval}^e(Q_2, D)$ that unifies and with condition `true` or `unknown`. Thus, $\texttt{Eval}_t^e(Q_1 Q_2, D) = \texttt{Eval}_t^e(Q_1, D) \ltimes_{\Uparrow} \texttt{Eval}_p^e(Q_2, D)$.

Selection. By definition, $(\sigma_\theta(Q))^+(D) = \sigma_\theta + (Q^+(D))$. By the induction hypothesis, $\sigma_{\theta^+}(Q^+(D)) = \sigma_{\theta^+}(\texttt{Eval}_t^e(Q, D))$. It can be easily verified by induction on the structure of + that a tuple t satisfies θ^+ iff $\texttt{eval}(\theta(t)) = \texttt{true}$. Thus, $\sigma_{\theta^+}(\texttt{Eval}_t^e(Q, D)) = \texttt{Eval}_t^e(\dot{\sigma}_\theta(\texttt{Eval}^e(Q, D)))$, which is equal to $\texttt{Eval}_t^e(\sigma_\theta(Q), D)$.

Cartesian product. $(Q_1 \times Q_2)^+(D) = Q_1^+(D) \times Q_2^+(D)$ by definition. By the induction hypothesis, it holds that $Q_1^+(D) = \texttt{Eval}_t^e(Q_1, D)$ and $Q_2^+(D) = \texttt{Eval}_t^e(Q_2, D)$. It is easy to see that $\texttt{Eval}_t^e(Q_1 \times Q_2, D) = \texttt{Eval}_t^e(Q_1, D) \times \texttt{Eval}_t^e(Q_2, D)$.

Projection. By definition, $(\pi_Z(Q))^+(D) = \pi_Z(Q^+(D))$. By the induction hypothesis, $\pi_Z(Q^+(D)) = \pi_Z(\texttt{Eval}_t^e(Q, D))$. It can be easily verified that $\texttt{Eval}_t^e(\pi_Z(Q), D) = \pi_Z(\texttt{Eval}_t^e(Q, D))$.

We now show $Q^?(D) = \texttt{Eval}_p^e(Q, D)$.

Identity. By definition, $R^? = R$. It is straightforward to check that $\texttt{Eval}_p^e(R, D) = R$.

Union. By definition, $(Q_1 \cup Q_2)^?(D) = Q_1^?(D) \cup Q_2^?(D)$. By the induction hypothesis, $Q_1^?(D) = \texttt{Eval}_p^e(Q_1, D) and Q_2^?(D) = \texttt{Eval}_p^e(Q_2, D)$. It can be easily verified that $\texttt{Eval}_p^e(Q_1 \cup Q_2, D) = \texttt{Eval}_p^e(Q_1, D) \cup \texttt{Eval}_p^e(Q_2, D)$, and thus the claim follows.

Intersection. $(Q_1 \cap Q_2)^?(D) = Q_1^?(D) \ltimes_{\Uparrow} Q_2^?(D)$ by definition. By the induction hypothesis, $Q_1^?(D) = \texttt{Eval}_p^e(Q_1, D) and Q_2^?(D) = \texttt{Eval}_p^e(Q_2, D)$. It can be easily verified that the tuples in $\texttt{Eval}_p^e(Q_1 \cap Q_2, D)$ are the tuples in $\texttt{Eval}^e(Q_1, D)$ whose condition is `true` or `unknown` and for which there is a unifying tuple in $\texttt{Eval}^e(Q_2, D)$ with condition `true` or `unknown`. As a consequence, $\texttt{Eval}_p^e(Q_1 \cap Q_2, D) = \texttt{Eval}_p^e(Q_1, D) \ltimes_{\Uparrow} \texttt{Eval}_p^e(Q_2, D)$.

Difference. $(Q_1 Q_2)^?(D) = Q_1^?(D) Q_2^+(D)$ by definition. By the induction hypothesis, $Q_1^?(D) = \texttt{Eval}_p^e(Q_1, D) and Q_2^+(D) = \texttt{Eval}_t^e(Q_2, D)$.

The definition of $\dot{}$ implies that $\texttt{Eval}^e p(Q_1 Q_2, D)$ contains the tuples in $\texttt{Eval}^e(Q_1, D)$ whose condition is `true` or `unknown` and the tuple does not appear in $\texttt{Eval}^e(Q_2, D)$ with condition `true`. Thus, $\texttt{Eval}^e_t(Q_1 Q_2, D) = \texttt{Eval}^e_p(Q_1, D)\texttt{Eval}^e_t(Q_2, D)$.

Selection. By definition, $(\sigma_\theta(Q))^?(D) = \sigma_{\theta^?}(Q^?(D))$. By the induction hypothesis, $\sigma_{\theta^?}(Q^?(D)) = \sigma_{\theta^?}(\texttt{Eval}^e_p(Q, D))$. It can be easily verified by induction on the structure of $\theta^?$ that a tuple t satisfies $\theta^?$ iff $\texttt{eval}(\theta(t)) = \texttt{true} \vee \mathit{eval}(\theta(t)) = \texttt{unknown}$. Thus, $\sigma_{\theta_?}(\texttt{Eval}^e_p(Q, D)) = \texttt{Eval}^e_p(\dot{\sigma}_\theta(\texttt{Eval}^e(Q, D)))$, which equals $\texttt{Eval}^e_p(\sigma_\theta(Q), D)$.

Cartesian product. $(Q_1 \times Q_2)^?(D) = Q_1^?(D) \times Q_2^?(D)$ by definition. By the induction hypothesis, $Q_1^?(D) = \texttt{Eval}^e_p(Q_1, D)$ and $Q_2^?(D) = \texttt{Eval}^e_p(Q_2, D)$. It is easy to see that $\texttt{Eval}^e_p(Q_1 \times Q_2, D) = \texttt{Eval}^e_p(Q_1, D) \times \texttt{Eval}^e_p(Q_2, D)$.

Projection. By definition, $(\pi_Z(Q))^?(D) = \pi_Z(Q^?(D))$. By the induction hypothesis, $\pi_Z(Q^?(D)) = \pi_Z(\texttt{Eval}^e_p(Q, D))$. It can be easily verified that $\texttt{Eval}^e_p(\pi_Z(Q), D) = \pi_Z(\texttt{Eval}^e_p(Q, D))$. □

We point out that, in the previous claim, D is a database (thus, possibly containing nulls), but without conditions. However, the eager evaluation first converts D into a conditional database D with all conditions being `true`, and then performs the evaluation $\texttt{Eval}^e()$ over $\overline{D}$, so that eventually tuples associated with `true` are certain answers with nulls. It immediately follows from the equivalence with [54] that eager evaluation has correctness guarantees.

Corollary 2.9. *$\mathit{Eval}^e_t()$ has correctness guarantees.*

In the following theorem, hardness easily follows from the fact that the eager evaluation over complete databases coincides with the classical evaluation of relational algebra queries.

Theorem 2.10. *For any database D and query Q, the evaluation of $\mathit{Eval}^e_t(Q, D)$ is complete for AC^0.*

We point out that the eager evaluation makes the intersection operator non-commutative. For instance, considering the database D of Example 2.1, we have that $\texttt{Eval}^e(E \cap S, D) = \{\langle \bot_1, \texttt{U} \rangle\}$, whereas $\texttt{Eval}^e(S \cap E, D) = \{\langle \texttt{mary}, \texttt{U} \rangle, \langle \texttt{bob}, \texttt{U} \rangle\}$. Obviously, the technique proposed in [54], being equivalent to the eager evaluation, has the same problem (see Section 2.3 for a more comprehensive discussion and an example of this issue for [54]). The problem with non-commutativity is that one order of the arguments can yield better results than

the other, and this depends on the contents of the database. Thus, the order of the arguments of intersection matters, but we cannot know which one is better a priori. As shown in the following, the evaluation algorithms we illustrate next do not suffer from this problem.

2.4.2 Semi-eager evaluation

The eager evaluation presented in the previous subsection is somehow limited, as shown in the following example.

Example 2.11. Consider the query $Q = P - (E \cup S)$ and the database D of Example 2.1. It is easy to see that $\mathtt{cert}(Q, D) = \{\mathtt{john}\}$. However, the eager evaluation behaves as follows:

$$
\begin{aligned}
&\mathtt{Eval}^{e}(Q, D)\\
&= \mathtt{eval}(\overline{P^D} \mathbin{\dot{-}} \mathtt{eval}(\overline{\mathrm{E}^D} \mathbin{\dot{\cap}} \overline{\mathrm{S}^D}))\\
&= \mathtt{eval}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \mathtt{mary}, \mathtt{T}\rangle\} \mathbin{\dot{-}}\\
&\quad \mathtt{eval}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \perp_1, \mathtt{T}\rangle\} \mathbin{\dot{\cap}} \{\langle \mathtt{mary}, \mathtt{T}\rangle, \langle \mathtt{bob}, \mathtt{T}\rangle\}))\\
&= \mathtt{eval}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \mathtt{mary}, \mathtt{T}\rangle\} \mathbin{\dot{-}}\\
&\quad \mathtt{eval}(\{\langle \mathtt{john}, \mathtt{john} = \mathtt{mary}\rangle, \langle \mathtt{john}, \mathtt{john} = \mathtt{bob}\rangle,\\
&\qquad \langle \perp_1, \perp_1 = \mathtt{mary}\rangle, \langle \perp_1, \perp_1 = \mathtt{bob}\rangle\}))\\
&= \mathtt{eval}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \mathtt{mary}, \mathtt{T}\rangle\} \mathbin{\dot{-}} \{\langle \mathtt{john}, \mathtt{F}\rangle, \langle \perp_1, \mathtt{U}\rangle\})\\
&= \mathtt{eval}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \mathtt{mary}, \mathtt{T}\rangle\} \mathbin{\dot{-}} \{\langle \perp_1, \mathtt{U}\rangle\})\\
&= \mathtt{eval}(\{\langle \mathtt{john}, \neg(\mathtt{john} = \perp_1 \wedge \mathtt{U})\rangle, \langle \mathtt{mary}, \neg(\mathtt{mary} = \perp_1 \wedge \mathtt{U})\rangle\})\\
&= \{\langle \mathtt{john}, \mathtt{U}\rangle, \langle \mathtt{mary}, \mathtt{U}\rangle\}
\end{aligned}
$$

Thus, $\mathtt{Eval}^{e}_{t}(Q, D) = \emptyset$. □

The problem of the eager evaluation in the previous example is that the three-valued evaluation of the c-tuples $\langle \perp_1, \perp_1 = \mathtt{mary}\rangle$ and $\langle \perp_1, \perp_1 = \mathtt{bob}\rangle$ yields (for both c-tuples) the c-tuple $\langle \perp_1, \mathtt{U}\rangle$, which loses some information of the original ones. Specifically, for the first (resp. second) c-tuple we forget that when $\perp_1$ is equal to $\mathtt{mary}$ (resp. $\mathtt{bob}$) the tuple $\mathtt{mary}$ (resp. $\mathtt{bob}$) is obtained, whereas when $\perp_1$ takes any other value $c \neq \mathtt{mary}$ (resp. $c \neq \mathtt{bob}$) no tuple is derived. Obviously, preserving more information allows us to perform more accurate analyses; so, the previous example suggests we might try to keep the information coming from equalities in conditions, and exploit equalities to refine the evaluation of conditions. For instance, in the scenario above, the evaluation of the c-tuple $\langle \perp_1, \perp_1 = \mathtt{mary}\rangle$ might

evaluate the condition $\perp_1 =$ mary into U and replace $\perp_1$ with mary in the c-tuple, thereby yielding $\langle$mary, U$\rangle$ (instead of $\langle \perp_1$, U$\rangle$, which is less informative). The same approach can be applied to the c-tuple $\langle \perp_1, \perp_1 =$ bob$\rangle$. This is the basic idea of our second algorithm, called *semi-eager evaluation*, which behaves like the eager evaluation in that conditions are evaluated right after each relational algebra operator, but it propagates equalities into tuples and conditions. To introduce the semi-eager evaluation, we first need some auxiliary notions.

Given a condition ϕ of the form $(\alpha = \beta)$, where $\alpha, \beta \in$ Null $\cup$ Const, we define ν_ϕ to be the identity on Null $\cup$ Const except that if at least one of α and β is a null, say α, then $\nu_\phi(\alpha) = \beta$. When both α and β are nulls, the choice of whether $\nu_\phi(\alpha) = \beta$ or $\nu_\phi(\beta) = \alpha$ is irrelevant for our purposes.

With a slight abuse of notation, we apply ν_ϕ also to any condition (resp. tuple) ψ; the result of the application, denoted $\nu_\phi(\psi)$, is the condition (resp. tuple) derived from ψ by replacing every null $\perp_i$ in ψ with $\nu_\phi(\perp_i)$.

To exploit equalities we need to manipulate conditions in c-tuples, and we do so by applying the commutativity and associativity properties of logical formulae to conditions. Given conditions ϕ_1, ϕ_2 and ϕ_3, we state the following syntactical rules to transform a condition into another equivalent one:

- Commutativity: $\phi_1 \star \phi_2 \equiv \phi_2 \star \phi_1$, for $\star \in \{\wedge, \vee\}$.
- Associativity: $(\phi_1 \star \phi_2) \star \phi_3 \equiv \phi_1 \star (\phi_2 \star \phi_3)$, for $\star \in \{\wedge, \vee\}$.

Each of the rules above says that the condition on the left-hand side of $\equiv$ can be replaced with the one on the right-hand side, and vice versa. For instance, $(\phi_1 \star \phi_2) \star \phi_3$ can be rewritten into $\phi_1 \star (\phi_2 \star \phi_3)$, and $\phi_1 \star (\phi_2 \star \phi_3)$ can be rewritten into $(\phi_1 \star \phi_2) \star \phi_3$.

Given two conditions ϕ and ϕ' , we write $\phi \equiv_{AC} \phi'$, iff ϕ' can be obtained from ϕ by iteratively applying the commutativity and associativity rules zero or more times. Essentially, the two rules above say that the conjuncts (resp. disjuncts) in a conjunction (resp. disjunction) can be rearranged in an arbitrary order. For every condition ϕ, we define $[\phi] = \{\phi' \mid \phi' \equiv_{AC} \phi\}$.

We now introduce a new function $\texttt{eval}^s(\cdot)$, which will be used in our next algorithm, called semi-eager evaluation, to evaluate conditions (here the symbol s is used to recall semi-eager). As we will see in the definition of the semi-eager evaluation reported in the following, $\texttt{eval}^s(\cdot)$ will replace function $\texttt{eval}^e(\cdot)$ used in the eager evaluation. This is essentially how the definition of semi-eager differs from

the one of eager evaluation. Given a c-tuple $\mathbf{t} = \langle t, \phi \rangle$, we define $\texttt{eval}^s(\mathbf{t})$ as follows:

- if ϕ is of the form $(\alpha = \beta)$, then $\texttt{eval}^s(\mathbf{t}) = \langle \nu_{(\alpha=\beta)}(t), \texttt{eval}((\alpha = \beta)) \rangle$;
- if $\phi \equiv_{AC} (\alpha = \beta) \wedge \phi'$ for some $(\alpha = \beta)$, then $\texttt{eval}^s(\mathbf{t}) = \langle t'', \texttt{eval}(\alpha = \beta) \wedge \phi'' \rangle$, where $\langle t'', \phi'' \rangle = \texttt{eval}^s(\langle \nu_{(\alpha=\beta)}(t), \nu_{(\alpha=\beta)}(\phi') \rangle)$;
- otherwise $\texttt{eval}^s(\mathbf{t}) = \texttt{eval}(\mathbf{t})$.

The first item above says that when ϕ is of the form $(\alpha = \beta)$, then $(\alpha = \beta)$ is propagated into t and the condition $(\alpha = \beta)$ is evaluated using $\texttt{eval}()$. So, for instance, if ϕ is $(\perp_1 = a)$, then every occurrence of $\perp_1$ in t is replaced with a and $(\perp_1 = a)$ is evaluated using $\texttt{eval}()$, which yields $\texttt{unknown}$. The second item above says that when ϕ is equivalent (using the commutativity and associativity rules) to a condition of the form $(\alpha = \beta) \wedge \phi'$, then (i) $(\alpha = \beta)$ is propagated into both t and ϕ'; (ii) $\texttt{eval}^s()$ is recursively applied to the resulting c-tuple, yielding a c-tuple $\langle t'', \phi'' \rangle$; (iii) the final result is the c-tuple consisting of t'' and the evaluation of $(\alpha = \beta) \wedge \phi''$ using $\texttt{eval}()$. The last item above addresses the case where no equality can be propagated. In such a case, $\texttt{eval}()$ is applied.

There might be different equalities for which the equivalence in the second item above holds (in such a case, one choice is made nondeterministically), and there could be equalities where both terms are nulls (in such a case we can choose nondeterministically to replace one null with the other). All alternative results are equivalent in that one can be derived from the other by simply renaming nulls. For instance, consider the conditional tuple $\langle \perp_1, \perp_1 = \perp_2 \wedge \perp_1 = \perp_3 \rangle$. By applying the $\texttt{eval}^s()$ function to this c-tuple, we can choose to exploit the first or the second equality. By exploiting the first equality $\perp_1 = \perp_2$ and replacing $\perp_1$ with $\perp_2$, we get $\langle \perp_2, \texttt{U} \wedge \perp_2 = \perp_3 \rangle$. Then, by replacing $\perp_2$ with $\perp_3$ we obtain $\langle \perp_3, \texttt{U} \rangle$. Alternatively, equivalent results that can be obtained by performing different choices are $\langle \perp_1, \texttt{U} \rangle$ and $\langle \perp_2, \texttt{U} \rangle$. Notice that all results differ only in terms of the index given to the null value.

For every c-table T , $\texttt{eval}^s(T) = \{\texttt{eval}^s(\mathbf{t}) \mid t \in T\}$.

Semi-eager evaluation is defined as follows:

- $\texttt{Eval}^s(Q, D) = \overline{R^D}$

- $\texttt{Eval}^s(Q_1 \cup Q_2, D) = \texttt{eval}^s(\texttt{Eval}^s(Q_1, D) \dot{\cup} \texttt{Eval}^s(Q_2, D))$
- $\texttt{Eval}^s(Q_1 \cap Q_2, D) = \texttt{eval}^s(\texttt{Eval}^s(Q_1, D) \dot{\cap} \texttt{Eval}^s(Q_2, D))$
- $\texttt{Eval}^s(Q_1 - Q_2, D) = \texttt{eval}^s(\texttt{Eval}^s(Q_1, D) \dot{-} \texttt{Eval}^s(Q_2, D))$
- $\texttt{Eval}^s(Q_1 \times Q_2, D) = \texttt{eval}^s(\texttt{Eval}^s(Q_1, D) \dot{\times} \texttt{Eval}^s(Q_2, D))$
- $\texttt{Eval}^s(\sigma_\theta(Q), D) = \texttt{eval}^s(\dot{\sigma}_\theta(Q, D)))$
- $\texttt{Eval}^s(\pi_Z(Q), D) = \texttt{eval}^s(\dot{\pi}_Z(\texttt{Eval}^s(Q, D)))$

Notice that the difference between the semi-eager evaluation (performed by the function $\texttt{Eval}^s()$) and the eager one (performed by function $\texttt{Eval}^e()$), is that $\texttt{eval}^s()$ is used in place of $\texttt{eval}()$.

Finally, we define:

$$\texttt{Eval}^s_t(Q, D) = \{t \mid \langle t, \texttt{true}\rangle \in \texttt{Eval}^s(Q, D)\}.$$

We have called this technique *semi-eager* as it behaves similar to the eager technique (conditions are evaluated as soon as possible), but before performing evaluations, equalities occurring in conditions are exploited. Below is an example.

Example 2.12. Consider again the database D of Example 2.1 and the query Q of Example 2.11. The semi-eager evaluation behaves as follows:

$$\begin{aligned}
&\texttt{Eval}^s(Q, D)\\
&= \texttt{eval}^s(\overline{P^D} \dot{-} \texttt{eval}(\overline{\mathrm{E}^D} \dot{\cap} \overline{\mathrm{S}^D}))\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \mathrm{T}\rangle, \langle \texttt{mary}, \mathrm{T}\rangle\} \dot{-}\\
&\quad \texttt{eval}^s(\{\langle \texttt{john}, \mathrm{T}\rangle, \langle \bot_1, \mathrm{T}\rangle\} \dot{\cap} \{\langle \texttt{mary}, \mathrm{T}\rangle, \langle \texttt{bob}, \mathrm{T}\rangle\}))\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \mathrm{T}\rangle, \langle \texttt{mary}, \mathrm{T}\rangle\} \dot{-}\\
&\quad \texttt{eval}^s(\{\langle \texttt{john}, \texttt{john} = \texttt{mary}\rangle, \langle \texttt{john}, \texttt{john} = \texttt{bob}\rangle,\\
&\qquad \langle \bot_1, \bot_1 = \texttt{mary}\rangle, \langle \bot_1, \bot_1 = \texttt{bob}\rangle\}))\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \mathrm{T}\rangle, \langle \texttt{mary}, \mathrm{T}\rangle\} \dot{-} \{\langle \texttt{john}, \mathrm{F}\rangle, \langle \texttt{mary}, \mathrm{U}\rangle, \langle \texttt{bob}, \mathrm{U}\rangle\})\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \mathrm{T}\rangle, \langle \texttt{mary}, \mathrm{T}\rangle\} \dot{-} \{\langle \texttt{mary}, \mathrm{U}\rangle, \langle \texttt{bob}, \mathrm{U}\rangle\})\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \neg(\texttt{john} = \texttt{mary} \wedge \mathrm{U}) \wedge \neg(\texttt{john} = \texttt{bob} \wedge \mathrm{U})\rangle,\\
&\qquad \langle \texttt{mary}, \neg(\texttt{mary} = \texttt{mary} \wedge \mathrm{U}) \wedge \neg(\texttt{mary} = \texttt{bob} \wedge \mathrm{U})\rangle\})\\
&= \{\langle \texttt{john}, \mathrm{T}\rangle, \langle \texttt{mary}, \mathrm{U}\rangle\}.
\end{aligned}$$

Thus, $\texttt{Eval}^s_t(Q, D) = \{\langle \texttt{john} \rangle\}$. Recall that $\texttt{cert}(Q, D) = \{\langle \texttt{john} \rangle\}$ and $\texttt{Eval}^e_t(Q, D) = \emptyset$. Thus, in this case, the semi-eager evaluation returns more certain tuples than the eager one.

Comparing the evaluations performed in Example 2.11 and Example 2.12 by $\texttt{Eval}^e()$ and $\texttt{Eval}^s$, respectively, they start to differ at the fourth step, where functions $\texttt{eval}()$ and $\texttt{eval}^s()$, respectively, are applied to the same set of c-tuples $S = \{\langle \texttt{john}, \texttt{john} = \texttt{mary} \rangle, \langle \texttt{john}, \texttt{john} = \texttt{bob} \rangle, \langle \bot_1, \bot_1 = \texttt{mary} \rangle, \langle \bot_1, \bot_1 = \texttt{bob} \rangle\}$, giving different results. Specifically,

$$\texttt{eval}(S) = \{\langle \texttt{john}, \texttt{F} \rangle, \langle \bot_1, \texttt{U} \rangle\}$$

while

$$\texttt{eval}^s(S) = \{\langle \texttt{john}, \texttt{F} \rangle, \langle \texttt{mary}, \texttt{U} \rangle, \langle \texttt{bob}, \texttt{U} \rangle\}$$

because the equalities $\bot_1 = \texttt{mary}$ and $\bot_1 = \texttt{bob}$ are propagated by $\texttt{eval}^s()$ into the c-tuples, and this makes $\texttt{Eval}^s(S)$ more informative than $\texttt{eval}(S)$, as equalities have been better exploited. The subsequent steps of $\texttt{Eval}^s()$ are executed in the same way as $\texttt{Eval}^e()$, but starting from more informative c-tuples, which allow the semi-eager evaluation to determine that $\texttt{john}$ is a certain answer. In fact, $\texttt{eval}^s(S)$ says that the only tuples that might be derived after the intersection are $\texttt{mary}$ and $\texttt{bob}$, which allows the semi-eager evaluation to determine that $\texttt{john}$ is certainly not in the result of the intersection, and thus $\texttt{john}$ is a certain answer. In contrast, $\texttt{eval}(S)$ says that any value $\bot_1$ might be derived after the intersection (including $\texttt{john}$), which does not allow the eager evaluation to determine that $\texttt{john}$ is a certain answer with nulls. □

As stated in the following simple proposition, the eager and semi-eager evaluations behave in the same way as long as only union, difference, Cartesian product, and projection are used. They might differ when selection and intersection are used, because the conditional evaluation of such operators introduce equalities in conditions, which can be better exploited by the semi-eager evaluation.

Proposition 2.13. *For every database D and query Q using only union, difference, Cartesian product, and projection, $\mathit{Eval}^e_t(Q, D) = \mathit{Eval}^s_t(Q, D)$.*

For arbitrary queries, exploiting equalities enables the semi-eager evaluation to provide strictly better approximations, that is, to return more tuples, while retaining correctness guarantees and polynomial time complexity.

Theorem 2.14. *For every query Q and database D, $\mathit{Eval}_t^e(Q, D) \subseteq \mathit{Eval}_t^s(Q, D)$. There exist a query Q and a database D such that $\mathit{Eval}_t^e(Q, D) \subset \mathit{Eval}_t^s(Q, D)$.*

Theorem 2.15. *$\mathit{Eval}_t^s()$ has correctness guarantees.*

Theorem 2.16. *$\mathit{Eval}_t^s(Q, D)$ can be computed in polynomial time in the size of D, for every query Q and database D.*

Observe that, as opposed to the eager evaluation, for the database D of *Example 1*, $Es(\overline{E} \cap \overline{S}, D) = \mathtt{Eval}^s((S \cap \overline{E}, D) = \{\langle \mathtt{mary}, \mathtt{U} \rangle, \langle \mathtt{bob}, \mathtt{U} \rangle\}$. The next proposition states that the semi-eager evaluation makes the intersection operator commutative, which is not the case for the eager evaluation and the approach of [54].

Proposition 2.17. *For every database D, and for all queries Q_1 and Q_2 of the same arity, $\mathit{Eval}_t^s(Q_1 \cap Q_2, D)$ and $\mathit{Eval}_t^s(Q_2 \cap Q_1, D)$ are isomorphic (i.e., equal up to renaming of nulls).*

Notice that in the previous proposition $\mathtt{Eval}_t^s(Q_1 \cap Q_2, D)$ and $\mathtt{Eval}_t^s(Q_2 \cap Q_1, D)$ are equal modulo renaming of labeled nulls, that is, one argument can be derived from the other by simply renaming nulls. In fact, as previously discussed, when applying $\mathtt{eval}^s()$ there might be nondeterministic choices about which equality is exploited and which null should replace another null. Different choices might lead to different results, which are nonetheless all equivalent in that one can be obtained from another by renaming nulls.

2.4.3 Lazy evaluation

The semi-eager evaluation improves on the eager one by better exploiting equalities. However, like the eager evaluation, conditions are evaluated right after each relational algebra operator is applied. This can be a limitation, because when collapsing conditions into `true`, `false` or `unknown`, some information might be lost, as shown in the example below. On the contrary, postponing conditions' evaluation yields longer conditions, which provide a more "global" view of how tuples are derived during query evaluation, and thus allow better analyses. We illustrate this aspect in the following example.

Example 2.18. Consider the database D of Example 2.1 and the query $Q = P - (P \cap (\sigma_{\$1 \neq \mathtt{john}}(E)))$. It can be easily verified that $\mathtt{cert}(Q, D) = \{\mathtt{john}\}$. Intuitively, one can see that `john` is a certain answer with nulls by doing a global analysis of the query: regardless of which

value $\perp_1$ takes, $\texttt{john}$ is never in the result of $\sigma_{\$1 \neq \texttt{john}}(E)$, and thus he is never in the result of $P \cap (\sigma_{\$1 \neq \texttt{john}}(E))$, and as a consequence he is always in result of $P - (P \cap (\sigma_{\$1 \neq \texttt{john}}(E)))$.

However, $\texttt{Eval}^s_t(Q, D) = \emptyset$, as

$$
\begin{aligned}
&\texttt{Eval}^s(Q, D)\\
&= \texttt{eval}^s(\overline{P^D} \dot{-} \texttt{eval}^s(\overline{P^D} \dot{\cap} \texttt{eval}^s(\dot{\sigma}_{\$1 \neq \texttt{john}}(\overline{E^D}))))\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-}\\
&\quad \texttt{eval}^s(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{\cap}\\
&\qquad \texttt{eval}^s(\dot{\sigma}_{\$1 \neq \texttt{john}}\{\langle \texttt{john}, \texttt{T}\rangle, \langle \perp_1, \texttt{T}\rangle\})))\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-}\\
&\quad \texttt{eval}^s(\{\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{\cap} \texttt{eval}^s(\{\texttt{john}, \texttt{john} \neq \texttt{john}\rangle,\\
&\quad \langle \perp_1, \perp_1 \neq \texttt{john}\rangle\})))\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-}\\
&\qquad \texttt{eval}^s\{\langle \texttt{john}, \texttt{john} \neq \texttt{john}\rangle, \langle \perp_1, \perp_1 \neq \texttt{john}\rangle\}))\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-} \texttt{eval}^s(\{\langle \texttt{john}, \texttt{john} = \perp_1 \wedge \texttt{U}\rangle,\\
&\quad \langle \texttt{mary}, \texttt{mary} = \texttt{mary} \wedge \texttt{U}\rangle\}))\\
&= \texttt{eval}^s(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-} \{\langle \texttt{john}, \texttt{U}\rangle, \langle \texttt{mary}, \texttt{U}\rangle\})\\
&= \{\langle \texttt{john}, \texttt{U}\rangle, \langle \texttt{mary}, \texttt{U}\rangle\}.
\end{aligned}
$$

Here, some information is lost at the fourth step (before applying the $\cap$ operator): the condition $\texttt{john} \neq \texttt{john}$ is $\texttt{false}$, and the tuple $\langle \texttt{john}, \texttt{F}\rangle$ is discarded from the result, whereas the condition $\perp_1 \neq \texttt{john}$ is $\texttt{U}$ and, therefore, the result of the application of the $\texttt{eval}^s()$ function gives the c-tuple $\langle \perp_1, \texttt{U}\rangle$. This means that the information that $\perp_1$ is never equal to $\texttt{john}$ (i.e., the condition $\perp_1 \neq \texttt{john}$), which would be useful in the next steps to understand that $\texttt{john}$ cannot belong to the result of the subquery $P \cap (\sigma_{\$1 \neq \texttt{john}}(E))$, is lost altogether. We are left with the c-tuple $\langle \perp_1, \texttt{U}\rangle$, which does not carry anymore that useful piece of information.

By postponing the evaluation of conditions, it is possible to have a more complete view of how tuples are derived, and combine the different pieces of information to perform better analyses: in our case, we could understand that $\texttt{john}$ is not in the result of the selection, and thus he is not in the result of the intersection, and thus he is in the result of the difference. □

The previous example suggests that delaying the evaluation of conditions can yield benefits in that more accurate analyses can be

performed. However, there is a trade-off here: while postponing conditions' evaluation enables for better analyses, this implies that longer conditions are kept and manipulated during the evaluation of a query, which incurs in higher processing costs (indeed, we will show this aspect both theoretically in Theorem 2.31 and experimentally in Section 4.2). In this regards, among all operators, the difference operator is the most critical one, as it yields much longer conditions. The idea of the *lazy evaluation* is the following compromise: in order to do better analyses, we postpone the evaluation of conditions as long as all operators but difference are encountered (because they produce conditions of "reasonable" size), and when the difference operator is encountered conditions are evaluated (and thus collapsed into `true`, `false`, `unknown`) because otherwise they would become too large. The formal definitions are reported below:

- $\texttt{Eval}^{\ell}(R, D) = \overline{R^D}$
- $\texttt{Eval}^{\ell}(Q_1 \cup Q_2, D) = \texttt{Eval}^{\ell}(Q_1, D) \dot{\cup} \texttt{Eval}^{\ell}(Q_2, D)$
- $\texttt{Eval}^{\ell}(Q_1 \cap Q_2, D) = \texttt{Eval}^{\ell}(Q_1, D) \dot{\cap} \texttt{Eval}^{\ell}(Q_2, D)$
- $\texttt{Eval}^{\ell}(Q_1 - Q_2, D) = \texttt{eval}^{\ell}(\texttt{Eval}^{\ell}(Q_1, D) \dot{-} \texttt{eval}^{s}(\texttt{Eval}^{\ell}(Q_2, D)))$
- $\texttt{Eval}^{\ell}(Q_1 \times Q_2, D) = \texttt{Eval}^{\ell}(Q_1, D) \dot{\times} \texttt{Eval}^{\ell}(Q_2, D)$
- $\texttt{Eval}^{\ell}(\sigma_{\theta}(Q), D) = \dot{\sigma}_{\theta}(\texttt{Eval}^{\ell}(Q, D))$
- $\texttt{Eval}^{\ell}(\pi_{Z}(Q), D) = \dot{\pi}_{Z}(\texttt{Eval}^{\ell}(Q, D))$

where for a c-tuple $\mathbf{t} = \langle t, \phi \rangle$ s.t. ϕ is of the form:

$$\phi = \phi' \wedge \bigwedge_{\langle t_j, v_j \rangle} \neg(v_j \wedge t = t_j)$$

with $v_j \in \{\texttt{true}, \texttt{false}, \texttt{unknown}\}$, $\texttt{eval}^{\ell}(\mathbf{t})$ is defined as follows:

$$\texttt{eval}^{\ell}(\mathbf{t}) = \langle t, \phi' \wedge \texttt{eval}(\bigwedge_{\langle t_j, v_j \rangle} \neg(v_j \wedge t = t_j)) \rangle$$

and for a c-table T , $el(\mathrm{T}) = \{\texttt{eval}^{\ell}(\mathbf{t}) \mid \mathbf{t} \in \mathrm{T}\}$.

Observe that function $el()$ is applied after the difference operator is encountered and thus the conditions to which it is applied have

exactly the form ϕ reported above. Given a query Q and a database D, we define

$$\texttt{Eval}_t^{\ell}(Q, D) = \{t \mid \langle t, \texttt{true}\rangle \in \texttt{eval}^s(\texttt{Eval}^{\ell}(Q, D))\}$$

that is, the true answers are computed by (i) first, evaluating $El(Q, D)$, yielding a c-table T , and (ii) then, evaluating $\texttt{eval}^s$)T).

Example 2.19. Consider again the query Q of Example 2.18 and the database D of Example 2.1. The lazy evaluation is performed as follows:

$$\begin{aligned}
&\texttt{Eval}^{\ell}(Q, D)\\
&= \texttt{eval}^{\ell}(\overline{P^D} \dot{-} \texttt{eval}^s(\overline{P^D} \dot{\cap} \texttt{eval}^s(\dot{\sigma}_{\$1 \neq \texttt{john}}(\overline{E^D}))))\\
&= \texttt{eval}^{\ell}(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-}\\
&\quad \texttt{eval}^s(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{\cap} \texttt{eval}^s(\dot{\sigma}_{\$1 \neq \texttt{john}}\{\langle \texttt{john}, \texttt{T}\rangle, \langle \bot_1, \texttt{T}\rangle\})))\\
&= \texttt{eval}^{\ell}(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-}\\
&\quad \texttt{eval}^s(\{\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{\cap} \texttt{eval}^s(\{\texttt{john}, \texttt{john} \neq \texttt{john}\rangle,\\
&\quad \langle \bot_1, \bot_1 \neq \texttt{john}\rangle\})))\\
&= \texttt{eval}^{\ell}(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-}\\
&\quad \texttt{eval}^s\{\langle \texttt{john}, \texttt{john} = \texttt{john} \wedge \texttt{john} \neq \texttt{john}\rangle,\\
&\quad \langle \texttt{john}, \texttt{john} = \bot_1 \wedge \bot_1 \neq \texttt{john}\rangle\}))\\
&\quad \langle \texttt{mary}, \texttt{mary} = \texttt{john} \wedge \texttt{john} \neq \texttt{john}\rangle, \langle \texttt{mary}, \texttt{mary} = \bot_1 \wedge \bot_1 \neq \texttt{john}\rangle\}))\\
&= \texttt{eval}^{\ell}(\{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{T}\rangle\} \dot{-} \texttt{eval}^s(\{\langle \texttt{mary}, \texttt{U}\rangle\})\\
&= \{\langle \texttt{john}, \texttt{T}\rangle, \langle \texttt{mary}, \texttt{U}\rangle\}.
\end{aligned}$$

Observe that, at the fourth step, by applying the $\dot{\cap}$ operator we get four c-tuples, whose conditions take into account both the selection and the intersection operators. After that, by applying the $\texttt{eval}^s()$ function, we have that

1. the first c-tuple, namely $\langle \texttt{john}, \texttt{john} = \texttt{john} \wedge \texttt{john} \neq \texttt{john}\rangle$, becomes $\langle \texttt{john}, \texttt{F}\rangle$, as condition $\texttt{john} = \texttt{john} \wedge \texttt{john} \neq \texttt{john}$ evaluates to `false`, and thus the c-tuple is discarded from the result.

2. The second c-tuple, namely $\langle \texttt{john}, \texttt{john} = \bot_1 \wedge \bot_1 \neq \texttt{john}\rangle$, becomes $\langle \texttt{john}, \texttt{F}\rangle$ and is also discarded from the result. The

reason is that $\texttt{john} = \perp_1$ evaluates to unknown, then john is substituted for $\perp_1$ in the second conjunct, which becomes $\texttt{john} \neq \texttt{john}$, which evaluates to fa, and finally $\texttt{unknown} \wedge \texttt{false}$ is false.

3. The third c-tuple, namely $\langle \texttt{mary}, \texttt{mary} = \texttt{john} \wedge \texttt{john} \neq \texttt{john} \rangle$, becomes $\langle \texttt{mary}, \texttt{F} \rangle$, because $\texttt{mary} = \texttt{john} \wedge \texttt{john} \neq \texttt{john}$ evaluates to false, and the c-tuple is discarded as well.

4. The fourth c-tuple, namely $\langle \texttt{mary}, \texttt{mary} = \perp_1 \wedge \perp_1 \neq \texttt{john} \rangle$, becomes $\langle \texttt{mary}, \texttt{U} \rangle$, because $\texttt{mary} = \perp_1$ is unknown, $\perp_1$ is substituted by mary in the second conjunct, which becomes $\texttt{mary} \neq \texttt{john}$ and evaluates to true, and $\texttt{unknown} \wedge \texttt{true}$ results in unknown.

Therefore, the result of the intersection operator contains only the c-tuple $\langle \texttt{mary}, \texttt{U} \rangle$. It is worth noting that, while the semi-eager evaluation gives john as unknown in the result of the intersection, the lazy evaluation gives john as certainly false, which in turn allows the lazy evaluation to realize that john is certainly true (i.e., a certain answer) in the result of the query. In fact, $\texttt{Eval}_t^{\ell}(Q, D) = \{\langle \texttt{john} \rangle\}$, whereas $\texttt{Eval}_t^{s}(Q, D) = \emptyset$. □

The following theorems state that the lazy evaluation provides strictly better approximations than the semi-eager one, has correctness guarantees, and can be computed in polynomial time.

Theorem 2.20. *For every query Q and database D, $\mathit{Eval}_t^{s}(Q, D) \subseteq \mathit{Eval}_t^{\ell}(Q, D)$. There exist a query Q and a database D such that $\mathit{Eval}_t^{s} \subset \mathit{Eval}_t^{\ell}(Q, D)$.*

Theorem 2.21. *$\mathit{Eval}_t^{\ell}()$ has correctness guarantees.*

Theorem 2.22. *$\mathit{Eval}_t^{\ell}(Q, D)$ can be computed in polynomial time in the size of D, for every query Q and database D.*

The following proposition says that also for the lazy evaluation the intersection operator is commutative.

Proposition 2.23. *For every database D, and for all queries Q_1 and Q_2 of the same arity, $\mathit{Eval}_t^{\ell}(Q_1 \cap Q_2, D)$ and $\mathit{Eval}_t^{\ell}(Q_2 \cap Q_1, D)$ are isomorphic (i.e., equal up to renaming of nulls).*

2.4.4 Aware evaluation

The main reason why the lazy evaluation improves on previous evaluation algorithms is that it delays the evaluation of conditions. Indeed, the evaluation is delayed until the difference operator is encountered, and when this happens, conditions are evaluated. A natural question now is if we can do better and postpone conditions' evaluation even further, that is, after the application of the difference operator. This is the basic question that leads to our last evaluation algorithm, which is called *aware evaluation*. The basic idea is to postpone conditions' evaluation until the very end: after the conditional evaluation of the whole query has been performed, conditions are rewritten into a "better" form through a set of rewriting rules (which keep the length of the conditions polynomial), and finally $\texttt{eval}^s()$ is applied. Before introducing the aware evaluation, we show some limitations of the lazy evaluation in the following example.

Example 2.24. Consider the database D of Example 2.1 and the query $Q = P - (E - S)$. It is easy to see that $\texttt{cert}(Q, D) = \{\langle \texttt{mary} \rangle\}$. Intuitively, the reason is that, regardless of the value $\bot_1$ takes, `mary` is never in the result of $E - S$ because she is in S, and therefore she is always in the result of $P - (E - S)$. Such kind of reasoning requires keeping track of the fact that `mary` cannot be in the result of $E - S$, and using this piece of information when the outer difference in $P - (E - S)$ is performed. However, as the query involves two difference operators and the lazy evaluation evaluates conditions when the difference operator is encountered, this information

is lost. Specifically, the lazy evaluation is performed as follows:

$$
\begin{aligned}
&\mathtt{Eval}^{\ell}(Q, D)\\
&= \mathtt{eval}^{\ell}(\overline{P^D} \mathbin{\dot{-}} \mathtt{eval}^{s}(\mathtt{eval}^{\ell}(\overline{E^D} \mathbin{\dot{-}} \mathtt{eval}^{s}(\overline{S^D}))))\\
&= \mathtt{eval}^{s}(\mathtt{eval}^{\ell}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \mathtt{mary}, \mathtt{T}\rangle\} \mathbin{\dot{-}}\\
&\quad \mathtt{eval}^{s}(\mathtt{eval}^{\ell}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \bot_1, \mathtt{T}\rangle\} \mathbin{\dot{-}} \mathtt{eval}^{s}(\{\langle \mathtt{mary}, \mathtt{T}\rangle, \langle \mathtt{bob}, \mathtt{T}\rangle\})))))\\
&= \mathtt{eval}^{s}(\mathtt{eval}^{\ell}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \mathtt{mary}, \rangle\} \mathbin{\dot{-}}\\
&\quad \mathtt{eval}^{s}(\mathtt{eval}^{\ell}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \bot_1, \mathtt{T}\rangle\} \mathbin{\dot{-}} \{\langle \mathtt{mary}, \mathtt{T}\rangle, \langle \mathtt{bob}, \mathtt{T}\rangle\}))))\\
&= \mathtt{eval}^{s}(\mathtt{eval}^{\ell}(\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \mathtt{mary}, \mathtt{T}\rangle\} \mathbin{\dot{-}} \{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \bot_1, \mathtt{U}\rangle\}))\\
&= \mathtt{eval}^{s}(\mathtt{eval}^{\ell}(\{\langle \mathtt{john}, \neg(\mathtt{john} = \mathtt{john} \wedge \mathtt{T}) \wedge \neg(\mathtt{john} = \bot_1 \wedge \mathtt{U})\rangle,\\
&\quad \langle \mathtt{mary}, \neg(\mathtt{mary} = \mathtt{john} \wedge \mathtt{T}) \wedge \neg(\mathtt{mary} = \bot_1 \wedge \mathtt{U})\rangle\}))\\
&= \mathtt{eval}^{s}(\{\langle \mathtt{john}, \mathtt{F}\rangle, \langle \mathtt{mary}, \mathtt{U}\rangle\})\\
&= \{\langle \mathtt{mary}, \mathtt{U}\rangle\}.
\end{aligned}
$$

Thus, $\mathtt{Eval}^{\ell}_{\mathtt{t}}(Q, D) = \emptyset$.

It is worth noting that at the third step, after the inner $\dot{-}$ operator is applied, we get the set of c-tuples $\{\langle \mathtt{john}, \mathtt{T} \wedge \neg(\mathtt{john} = \mathtt{mary} \wedge \mathtt{T}) \wedge \neg(\mathtt{john} = \mathtt{bob} \wedge \mathtt{T})\rangle, \langle \bot_1, \mathtt{T} \wedge \neg(\bot_1 = \mathtt{mary} \wedge \mathtt{T}) \wedge \neg(\bot_1 = \mathtt{bob} \wedge \mathtt{T})\rangle\}$. By evaluating condition, since both equalities $\mathtt{john} = \mathtt{mary}$ and $\mathtt{john} = \mathtt{bob}$ are false, whereas both equalities $\bot_1 = \mathtt{mary}$ and $\bot_1 = \mathtt{bob}$ are unknown, we get the set of tuples $\{\langle \mathtt{john}, \mathtt{T}\rangle, \langle \bot_1, \mathtt{U}\rangle\}$. Thus, the fact that mary is certainly false in the result of $E - S$ is lost, which in turn prevents us from concluding that she is certainly true in $P - (E - S)$. □

As mentioned before, besides delaying conditions' evaluation, the aware evaluation relies on a set of rules to syntactically manipulate conditions. The aim is to rewrite conditions into simpler equivalent ones (equivalence under 3-valued logic) so that a better analysis can be performed. Simplified conditions are eventually evaluated. The set of rewriting rules is reported next:

1. De Morgan: $\neg(\phi_1 \wedge \phi_2) \vdash (\neg\phi_1 \vee \neg\phi_2)$ and $\neg(\phi_1 \vee \phi_2) \vdash (\neg\phi_1 \wedge \neg\phi_2)$
2. Negation: $\neg(\neg\phi) \vdash \phi$, $\neg(\phi_1 = \phi_2) \vdash (\phi_1 \neq \phi_2)$, $\neg(\phi_1 \neq \phi_2) \vdash (\phi_1 = \phi_2)$, $\neg\mathtt{unknown} \vdash \mathtt{unknown}$, $\neg\mathtt{true} \vdash \mathtt{false}$, and $\neg\mathtt{false} \vdash \mathtt{true}$.
3. Middle excluded: $(\alpha = \beta) \vee (\alpha \neq \beta) \vdash \mathtt{true}$; $(\alpha \neq \beta) \vee (\alpha \neq \beta') \vdash \mathtt{true}$, where $\beta, \beta' \in \mathtt{Const}$ and $\beta \neq \beta'$;

4. Contradiction: $(\alpha = \beta) \wedge (\alpha \neq \beta) \vdash \texttt{false}$; $(\alpha = \beta) \wedge (\alpha = \beta') \vdash \texttt{false}$, where $\beta, \beta' \in \texttt{Const}$ and $\beta \neq \beta'$.

5. Or-simplification: $\phi \vee \phi \vdash \phi, \phi \vee \texttt{false} \vdash \phi$, and $\phi \vee \texttt{true} \vdash \texttt{true}$.

6. And-simplification: $\phi \wedge \phi \vdash \phi, \phi \wedge \texttt{true} \vdash \phi$, and $\phi \wedge \texttt{false} \vdash \texttt{false}$.

7. Equality: $(\alpha = \alpha) \vdash \texttt{true}$, for $\alpha \in \texttt{Const} \cup \texttt{Null}$ and $(\alpha \neq \alpha) \vdash \texttt{false}$, for $\alpha \in \texttt{Const} \cup \texttt{Null}$.

8. Inequality: $(\alpha = \beta) \vdash \texttt{false}$ if $\alpha, \beta \in \texttt{Const}$ and $\alpha \neq \beta$, and $(\alpha \neq \beta) \vdash \texttt{true}$ if $\alpha, \beta \in \texttt{Const}$ and $\alpha \neq \beta$.

Notice that the distributivity rule is not included in our set of rules, because this allows us to keep the length of conditions polynomial. The result of applying a rule $\phi' \vdash \phi''$ to a condition ϕ is the condition obtained from ϕ by replacing every occurrence of ϕ' with ϕ''. We write $\phi \overset{*}{\vdash} \phi'$, where ϕ and ϕ' are conditions, if (i) ϕ' can be derived from ϕ by iteratively applying rules 18 along with the commutativity and associativity rules, and (ii) none of the rules 1-8 is applicable to any of the conditions in $[\phi']$ (recall that $[\phi']$ is the set of all conditions that can be obtained from ϕ' by iteratively applying the commutativity and associativity rules zero or more times). If $\phi \overset{*}{\vdash} \phi'$, we say that ϕ' is a minimal condition for ϕ. Intuitively, a minimal condition ϕ' is obtained by iteratively applying rules 1-8 and the commutativity and associativity rules until none of the rules 1-8 can be applied to ϕ' or any other condition derivable from ϕ' by means of the commutativity and associativity rules.

There can be multiple minimal conditions of a condition ϕ, but they are all equivalent w.r.t. the commutativity and associativity rules (roughly speaking, they differ only w.r.t. the order of their terms), that is, if $\phi \overset{*}{\vdash} \phi'$ and $\phi \overset{*}{\vdash} \phi''$ then $\phi' \in [\phi'']$ and $\phi'' \in [\phi']$. Thus, we can talk about the minimal condition of ϕ, which we denote as $\texttt{minimal}(\phi)$.

Proposition 2.25. *Given a condition ϕ, computing a condition ϕ' such that $\phi \overset{*}{\vdash} \phi'$ can be done in polynomial time.*

For a c-tuple $\mathbf{t} = \langle t, \phi \rangle$, we define $\mathtt{eval}^a(\mathbf{t})$ as follows:
$\mathtt{eval}^a(\mathbf{t}) = \mathtt{eval}^s(\langle t, \mathtt{minimal}(\phi) \rangle)$
and for a c-table T, $\mathtt{eval}^a(\mathrm{T}) = \{\mathtt{eval}^a(\mathbf{t}) | \mathbf{t} \in \mathrm{T}\}$.

As minimal conditions differ only w.r.t. the order of their terms, two different minimal conditions (derived from the same original condition) cannot give different results in the evaluation above.

As said before, all minimal conditions (derived from the same original one) are equivalent and, therefore, the result of the evaluation above is independent from the chosen minimal condition, that is, for any two c-tuples $\langle t, \phi' \rangle$ and $\langle t, \phi'' \rangle$ derived from the same c-tuple $\langle t, \phi \rangle$, where ϕ', ϕ'' are minimal conditions for ϕ, then $\mathtt{eval}^a(\langle t, \phi' \rangle)$ and $\mathtt{eval}^a(\langle t, \phi'' \rangle)$ may differ only in indexes of nulls (i.e., one can be obtained from the other by simply renaming nulls).

Given a query Q and a database D, we define $\mathtt{Eval}^a(Q, D) = \mathtt{eval}^a(\dot{Q}(D))$. Moreover, we define:
$\mathtt{Eval}^a_{\mathtt{t}}(Q, D) = \{t \mid \langle t, \mathtt{true} \rangle \in \mathtt{Eval}^a(Q, D)\}$.

Example 2.26. Consider again the database D and the query Q of Example 2.24. The aware evaluation is performed as follows:

$$
\begin{aligned}
&\mathtt{Eval}^a(Q, D) \\
&= \mathtt{eval}^a(\overline{P^D} \mathbin{\dot{-}} (\overline{E^D} \mathbin{\dot{-}} \overline{S^D})) \\
&= \mathtt{eval}^a(\{\langle \mathtt{john}, \mathrm{T} \rangle, \langle \mathtt{mary}, \mathrm{T} \rangle\} \mathbin{\dot{-}} \mathtt{eval}^a(\{\langle \mathtt{john}, \mathrm{T} \rangle, \langle \bot_1, \mathrm{T} \rangle\} \mathbin{\dot{-}} \\
&\qquad \{\langle \mathtt{mary}, \mathrm{T} \rangle, \langle \mathtt{bob}, \mathrm{T} \rangle\})) \\
&= \mathtt{eval}^a(\{\langle \mathtt{john}, \mathrm{T} \rangle, \langle \mathtt{mary}, \mathrm{T} \rangle\} \mathbin{\dot{-}} \{\langle \mathtt{john}, \phi_1 \rangle, \langle \bot_1, \phi_2 \rangle\})
\end{aligned}
$$

where $\phi_1 = \neg(\mathtt{john} = \mathtt{mary}) \wedge \neg(\mathtt{john} = \mathtt{bob})$ and
$\phi_2 = \neg(\bot_1 = \mathtt{mary}) \wedge \neg(\bot_1 = \mathtt{bob})$

$$
\begin{aligned}
&= \mathtt{eval}^a(\{\langle \mathtt{john}, \neg(\phi_1 \wedge \mathtt{john} = \mathtt{john}) \wedge \neg(\phi_2 \wedge \mathtt{john} = \bot_1) \rangle, \\
&\qquad \langle \mathtt{mary}, \neg(\phi_1 \wedge \mathtt{mary} = \mathtt{john}) \wedge \neg(\phi_2 \wedge \mathtt{mary} = \bot_1) \rangle\}) \\
&= \mathtt{eval}^s(\{\langle \mathtt{john}, \mathrm{F} \rangle, \langle \mathtt{mary}, \mathrm{T} \rangle\}) \\
&= \{\langle \mathtt{mary}, \mathrm{T} \rangle\}.
\end{aligned}
$$

Thus, $\texttt{Eval}^{a}_{\texttt{t}}(Q, D) = \{\langle \texttt{mary} \rangle\}$, whereas $\texttt{Eval}^{\ell}_{\texttt{t}}(Q, D) = \emptyset$. Observe that when computing

$$\begin{aligned}&\texttt{eval}^{a}(\{\langle \texttt{john}, \neg(\phi_1 \wedge \texttt{john} = \texttt{john}) \wedge \neg(\phi_2 \wedge \texttt{john} = \bot_1)\rangle,\\ &\quad \langle \texttt{mary}, \neg(\phi_1 \wedge \texttt{mary} = \texttt{john}) \wedge \neg(\phi_2 \wedge \texttt{mary} = \bot_1)\rangle\})\end{aligned}$$

which is equal to

$$\begin{aligned}&\texttt{eval}^{a}(\{\langle \texttt{john}, \neg(\neg(\texttt{john} = \texttt{mary}) \wedge \neg(\texttt{john} = \texttt{bob}) \wedge \texttt{john} = \texttt{john}) \wedge\\ &\quad \neg(\neg(\bot_1 = \texttt{mary}) \wedge \neg(\bot_1 = \texttt{bob}) \wedge \texttt{john} = \bot_1)\rangle,\\ &\quad \langle \texttt{mary}, \neg(\neg(\texttt{john} = \texttt{mary}) \wedge \neg(\texttt{john} = \texttt{bob}) \wedge \texttt{mary} = \texttt{john}) \wedge\\ &\quad \neg(\neg(\bot_1 = \texttt{mary}) \wedge \neg(\bot_1 = \texttt{bob}) \wedge \texttt{mary} = \bot_1)\rangle\})\end{aligned}$$

rewriting rules are applied. Specifically, for the first c-tuple rewriting rules are applied as follows:

$$\begin{aligned}&\neg\big(\neg(\texttt{john} = \texttt{mary}) \wedge \neg(\texttt{john} = \texttt{bob}) \wedge \texttt{john} = \texttt{john}\big) \wedge\\ &\qquad \neg\big(\neg(\bot_1 = \texttt{mary}) \wedge \neg(\bot_1 = \texttt{bob}) \wedge \texttt{john} = \bot_1\big)\\ &\vdash \neg\big(\neg(\texttt{false}) \wedge \neg(\texttt{false}) \wedge \texttt{john} = \texttt{john}\big) \wedge\\ &\qquad \neg\big(\neg(\bot_1 = \texttt{mary}) \wedge \neg(\bot_1 = \texttt{bob}) \wedge \texttt{john} = \bot_1\big) && \text{rule 8}\\ &\vdash \neg\big(\neg(\texttt{false}) \wedge \neg(\texttt{false}) \wedge \texttt{true}\big) \wedge \neg\big(\neg(\bot_1 = \texttt{mary})\\ &\qquad \wedge \neg(\bot_1 = \texttt{bob}) \wedge \texttt{john} = \bot_1\big) && \text{rule 7}\\ &\vdash \texttt{false} \wedge \neg\big(\neg(\bot_1 = \texttt{mary}) \wedge \neg(\bot_1 = \texttt{bob}) \wedge \texttt{john} = \bot_1\big) && \text{rule 2, 6}\\ &\vdash \texttt{false} && \text{rule 6}\end{aligned}$$

For the second c-tuple, rewriting rules are applied as follows:

$$
\begin{array}{ll}
\neg\big(\neg(\texttt{john} = \texttt{mary}) \wedge \neg(\texttt{john} = \texttt{bob}) \wedge \texttt{mary} = \texttt{john}\big) \wedge & \\
\quad \neg\big(\neg(\bot_1 = \texttt{mary}) \wedge \neg(\bot_1 = \texttt{bob}) \wedge \texttt{mary} = \bot_1\big) & \\
\vdash \neg\big(\neg(\texttt{false}) \wedge \neg(\texttt{false}) \wedge \texttt{false}\big) & \\
\quad \wedge \neg\big(\neg(\bot_1 = \texttt{mary}) \wedge \neg(\bot_1 = \texttt{bob} \wedge \texttt{mary} = \bot_1\big) & \text{rule 8} \\
\vdash \neg\big(\texttt{false}\big) \wedge \neg\big(\neg(\bot_1 = \texttt{mary}) \wedge & \\
\quad \neg(\bot_1 = \texttt{bob}) \wedge \texttt{mary} = \bot_1\big) & \text{rule 6} \\
\vdash \texttt{true} \wedge \neg\big(\neg(\bot_1 = \texttt{mary}) \wedge \neg(\bot_1 = \texttt{bob}) \wedge \texttt{mary} = \bot_1\big) & \text{rule 2} \\
\vdash \neg\big(\neg(\bot_1 = \texttt{mary}) \wedge \neg(\bot_1 = \texttt{bob}) \wedge \texttt{mary} = \bot_1\big) & \text{rule 6} \\
\vdash \big((\bot_1 = \texttt{mary}) \vee (\bot_1 = \texttt{bob}) \vee (\texttt{mary} \neq \bot_1)\big) & \text{rule 1, 2} \\
\vdash \big(\texttt{true} \vee (\bot_1 = \texttt{bob})\big) & \text{rule 3} \\
\vdash \texttt{true} & \text{rule 5}
\end{array}
$$

The reason why the aware evaluation is able to give `john` as a certain answer, while the lazy does not, is that the former does not evaluate conditions in intermediate steps and thus it eventually ends up with conditions describing the entire query evaluation process, which in turn enable the global analysis discussed at the beginning of Example 2.24. □

The following theorems state that the aware evaluation provides strictly better approximations than the lazy one, has correctness guarantees, and can be computed in polynomial time.

Theorem 2.27. *For every query Q and database D, $\mathit{Eval}_t^{\ell}(Q, D) \subseteq \mathit{Eval}_t^{a}(Q, D)$. There exists a query Q and a database D such that $\mathit{Eval}_t^{\ell}(Q, D) \subset \mathit{Eval}_t^{a}(Q, D)$.*

Theorem 2.28. *$\mathit{Eval}_t^{a}()$ has correctness guarantees.*

Theorem 2.29. *$\mathit{Eval}_t^{a}(Q, D)$ can be computed in polynomial time in the size of D, for every query Q and database D.*

Proposition 2.30. *For every database D, and for all queries Q_1 and Q_2 of the same arity, $\mathit{Eval}_t^{a}(Q_1 \cap Q_2, D)$ and $\mathit{Eval}_t^{a}(Q_2 \cap Q_1, D)$ are isomoprhic (i.e., equal up to renaming of nulls).*

We now present a theorem that characterizes the size of the conditions handled by $\texttt{Eval}^{\star}()$, for $\star \in \{e, s, \ell, a\}$. The theorem highlights that, in general, algorithms providing better approximations have to handle more complex conditions. Specifically, consider a query Q and a database D. We use the following notations: n is the maximum number of tuples of relations in D, a is the maximum arity of

relations in D, k is the number of relational algebra operators in Q, and s is the maximum number of conjuncts of the selection conditions in Q - recall that we assume that every selection condition is a conjunction of equalities/inequalities (see Subsection 1.2.3). It is worth noting that when data complexity is considered, a, k, and s are all fixed.

In the following theorem, we use $||\phi||$ to denote the maximum length of a condition produced during the evaluation of $\texttt{Eval}^\star(Q, D)$.

Theorem 2.31.

1. *For the eager evaluation,* $||\phi|| = O(1)$.
2. *For the semi-eager evaluation,* $||\phi|| = O(max\{s, k \cdot a\})$.
3. *For the lazy evaluation,* $||\phi|| = O(k \cdot max\{s, a\})$.
4. *For the aware evaluation,* $||\phi|| = O(n^k \cdot max\{s, k \cdot a\})$.

Thus, the previous theorem shows that as we move from the eager evaluation (the simplest one) to the aware evaluation (the most complex one) the size of the conditions to be handled increases - however, this can be paid back by more certain answers with nulls being returned. We point out that the maximum length of a condition is polynomial for all evaluation strategies when data complexity is considered, as a, k, and s are all fixed.

We experimentally study this aspect in Chapter 4. The experimental results confirm what suggested by *Theorem* 2.31, that is, running times increase as we move from the eager evaluation to the aware one (but more certain answers with nulls are returned).

Example 2.32. Consider again the database D and the query Q of Example 2.24, namely $P - (E - S)$.

For the eager evaluation, conditions are always either true or false or unknown, and thus their size is $O(1)$. The reason is that $\texttt{eval}()$ can be evaluated "on the fly" by keeping conditions of constant size, that is, equalities/inequalities and logical connectives can be evaluated according to the three-valued evaluation as they are generated without materializing entire conditions.

Notice that, in this case, there are no equalities to be propagated by the semi-eager evaluation (in fact, there is no selection and no intersection), and thus the semi-eager evaluation behaves like the eager one.

The lazy evaluation was shown in Example 2.24 and produces conditions that are no more of constant size. The aware evaluation was shown in Example 2.26 and produces even longer conditions, as the conditional evaluation of the entire query has to be carried out. □

Chapter 3

A System Prototype for Approximate Query Answering over Incomplete Data

In this chapter we showcase ACID, a system to compute sound sets of certain answers [35]. The central tools of its underlying algorithms are conditional tables and the conditional evaluation of relation algebra. Different evaluation strategies can be applied, with more accurate ones having higher complexity, but returning more certain answers. These techniques have been presented in Chapter 2. We show how to query incomplete databases using the ACID system, which offers a suite of approximation algorithms enabling users to choose the technique that best meets their needs in terms of balance between efficiency and quality of the result's approximation.

3.1 Introduction

A principled way of answering queries over incomplete databases is to compute *certain answers*, which are query answers that can be obtained from all the complete databases represented by an incomplete database [28, 69, 68].

For basic notions and notations about certain query answers, the way SQL behaves in the presence of nulls, see Section 1.2.2.

In the previous chapter we have illustrated state-of-the-art evaluation algorithms with correctness guarantees leveraging conditional tables and the conditional evaluation of relational algebra (see Section 2.4). Conditional tables are standard tables where each tuple is

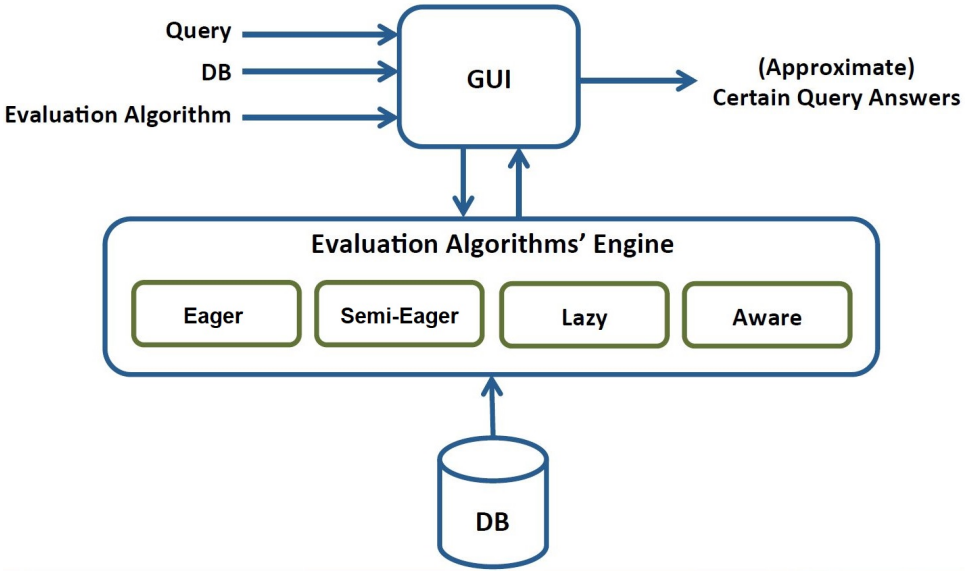

FIGURE 3.1: System Architecture.

associated with a condition and the conditional evaluation is a generalization of relational algebra that manipulate conditional tables. Conditions keep track of how tuples are derived and how nulls are used in comparison operators (see Subsection 1.2.3).

These algorithms are called *eager*, *semi-eager*, *lazy* and *aware* evaluations and we referred to them as the GMT-approaches. They have been implemented in the ACID system, which enables users to query incomplete databases and get under-approximations of the certain answers, choosing the evaluation strategy that is most suitable for the application at hand.

3.2 System Overview

The ACID system has been implemented in Java. The system architecture is depicted in Figure 3.1.

There are three main components: a graphical user interface (GUI), the evaluation algorithms' engine, and the database.

The GUI allows user to specify the query to be evaluated, the database, and the type of evaluation to be performed, that is, the approximation algorithm to be applied. The GUI displays the result of evaluating the specified query over the provided database according to the chosen evaluation algorithm. Different filters can be applied to the result (more details are discussed in the next section).

The system's engine supports the four evaluation algorithms mentioned in the previous section, with the eager algorithm being the most efficient but the least accurate one, and the aware algorithm being the most accurate but the least efficient one. The basic ideas of the approximation algorithms are as follows:

- The *eager evaluation* evaluates tuples' conditions right after each relational algebra operator has been applied, using three-valued logic.
- The *semi-eager evaluation* behaves like the naive one, but it better exploits equalities in conditions (by propagating values into tuples and conditions) to provide more accurate results.
- The *lazy evaluation* improves upon the semi-naive one by postponing conditions' evaluation until the set difference operator is encountered in the query.
- The *aware evaluation* provides even more accurate results and behaves as follows: it performs the conditional evaluation of the entire query, then it uses a set of axioms to "simplify" conditions, and eventually it evaluates (simplified) tuples' conditions.

The ACID system manages relational databases possibly containing labeled nulls (in the literature, they have been called *naive tables*, *V-tables*, and *e-tables* [1, 43, 59]). Thus, the same (labeled) null can occur multiple times - e.g., this can be used to express that there are two employees with the same unknown salary.

The GUI provides information on the query, the database, and the evaluation strategy to the engine, which computes the approximate certain answers accessing the database. After the evaluation has been carried out, the engine returns the result to the GUI.

The ACID system provides also an API which allow third party applications to interact with the system.

We go into the details of how to interact with the ACID system in the following section.

3.3 Demonstration

The ACID system will be demonstrated through interaction with a graphical user interface (cf. Figure 3.2). We will provide sample

queries over different incomplete datasets, even though users can experience querying their own databases with custom queries.

Our goal is to demonstrate the efficiency and effectiveness achieved by the different approximation algorithms when querying incomplete databases, showing that algorithms of increasing complexity have higher running times but provide better results. Users can thus choose the right balance between efficiency and quality of the results according to their needs.

A typical interaction with the system involves the following steps:

1. The user specifies the input databases. Specifically, for each table in the database, its location in the file system is provided. Tables are supposed to be in csv format.

2. The user specifies the query to be evaluated using standard SQL syntax. Queries can be loaded from and saved to files.

3. The user specifies the evaluation strategy that has to be applied to evaluate the query (indeed, the system supports also the "standard" evaluation mentioned in the introduction and the conditional evaluation of a query).

4. After the evaluation has been launched and has finished, the result and statistics are displayed. Specifically, the result is a set of a tuples, where each tuple is associated with a condition that is either true or unknown. Tuples associated with true are guaranteed to be certain answers to the input query. The result can be filtered with respect to the truth value of the tuples, thus displaying only true or only unknown tuples. The total number of true (resp. unknown) tuples is displayed as well as the execution time. Results can be saved to files.

With the same query and database, moving to more accurate strategies, that is, from the eager (resp. semi-eager, lazy) evaluation to the semi-eager (lazy, aware) one, users can see better results, that is, more tuples with condition `true` (i.e., more certain answers), but running times might get higher.

In general, using the system and analysing the query syntax, users can figure out the strategy that is best suited for their purposes.

As an example, Tables 3.1, 3.2, 3.3 report the execution time, the number of `true` and `unknown` answers for three sample queries over a database with the same schema of the one in Example 2.1, with 1000 tuples per relation and 10% of nulls (randomly generated).

The queries are:

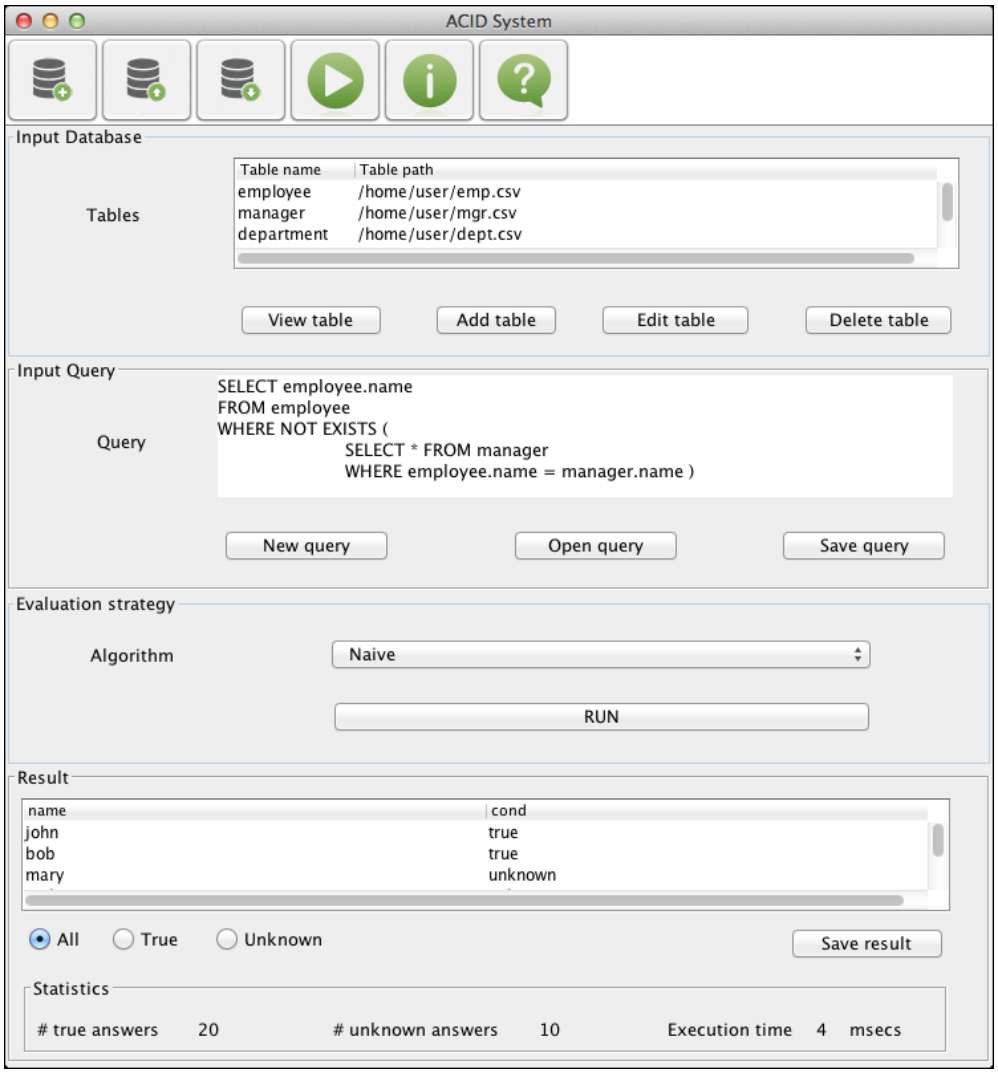

FIGURE 3.2: ACID system's GUI.

- $Q_{se} = E - \sigma_{\$1=c}(S)$,
- $Q_{lazy} = P - (E \cap (\sigma_{\$1 \neq c}(S)))$, and
- $Q_{aware} = P - (E - S)$,

where c is a value randomly chosen from those in S.

The purpose of the first scenario is to exhibit a query (namely, Q_{se}) that shows the benefits of going from the naive to the semi-naive evaluation - notice that, in this case, there is no benefit in applying the lazy or aware evaluation, as the structure of the query does not have features that can be exploited by them.

	Q_{se}		
	Time	#true	#unknown
Eager	518	741	167
Semi-eager	615	763	145
Lazy	661	763	145
Aware	3580	763	145

TABLE 3.1: Runtime (msecs), number of true and possible answers for the query $Q_{se} = E - \sigma_{\$1=c}(S)$.

	Q_{lazy}		
	Time	#true	#unknown
Eager	788	710	189
Semi-eager	1090	710	189
Lazy	1579	737	162
Aware	4350	737	162

TABLE 3.2: Runtime (msecs), number of true and possible answers for the query $Q_{lazy} = P - (E \cap (\sigma_{\$1 \neq c}(S)))$.

	Q_{aware}		
	Time	#true	#unknown
Eager	2163	100	731
Semi-eager	2347	100	731
Lazy	5542	100	731
Aware	10376	231	600

TABLE 3.3: Runtime (msecs), number of true and possible answers for the query $Q_{aware} = P - (E - S)$.

Likewise, the purpose of the second and third scenarios is to show the advantage of using the lazy (resp. aware) evaluation rather than the semi-eager (resp. lazy) one.

3.4 Discussion

Certain answers are a principled manner to answer queries on incomplete databases. Since their computation is a coNP-hard problem, recent research has focused on developing polynomial time algorithms providing under-approximations. [49] presented the GMT-approaches and we implemented them in the ACID system, which allows users to query incomplete information and get approximate

answers with the flexibility of choosing the technique that best meets their needs in terms of balance between efficiency and quality of the result's approximation.

Chapter 4

Optimizing the Computation of Approximate Certain Query Answers over Incomplete Databases

In this chapter, we propose a novel technique that allows us to improve the approximation algorithms presented in Chapter 2, obtaining a good balance between running time and quality of the results [34]. This new technique is placed between the lazy evaluation and the aware one (in terms of quality of the results), thus it is can be seen as an improvement over the lazy technique. First we present the behavior of this novel approach, then we report experimental results confirming its effectiveness.

4.1 Introduction

We will focus on the semantics of certain answers as the semantics of query answering over incomplete databases. In order to recall how the this semantics works, the reader may refer to Example 2.1.

For databases containing (labeled) nulls, certain answers to positive queries can be easily computed in polynomial time as follows: first a "standard" evaluation (that is, treating nulls as standard constants) is applied; then tuples with nulls in the result of the first step are discarded and the remaining tuples are the certain answers to the query. However, for more general queries with negation the problem of computing certain answers becomes coNP-hard. To make query answering feasible in practice, one might resort to SQL's evaluation, but unfortunately, the way SQL behaves in the presence of nulls may result in wrong answers. As evidenced in [68], there are two ways

in which certain answers and SQL's evaluation may differ: (i) SQL can miss some of the tuples that belong to certain answers, thus producing false negatives, or (ii) SQL can return some tuples that do not belong to certain answers, that is, false positives. While the first case can be seen as an under-approximation of certain answers (a sound but possibly incomplete set of certain answers is returned), the second scenario must be avoided, as the result might contain plain incorrect answers, that is, tuples that are not certain. The experimental analysis in [54] showed that false positive are a real problem for queries involving negation - they were always present and sometimes they constitute almost 100% of the answers. Thus, on the one hand, SQL's evaluation is efficient but flawed, on the other hand, certain answers are a principled semantics but with high complexity. To deal with this issue, there has been work on developing algorithms that compute a sound but possibly incomplete set of certain answers [35, 50, 49, 54, 69, 68]. Computing sound sets of consistent query answers over inconsistent databases has been addressed in [41], but databases are assumed to be complete, while in this paper we consider incomplete databases with no integrity constraints. In this chapter, we start with an experimental evaluation of the GMT-approaches proposed in Chapter 2. Experimental results confirm what suggested from the theoretical analysis carried out in [49]: moving from the eager to the aware algorithm, running times increase, but this paid back by more certain answers being found. While we observe a mild increase of running times when moving from eager to semi-eager, and from semi-eager to lazy, there is a much higher difference between the running times of the lazy and aware evaluations. This raised the question on whether we can devise a novel technique between lazy and aware, which achieves a good balance between running time and quality of the results.We answer this question positively by proposing a novel evaluation strategy, called *lazy*$^+$, which improves upon the lazy evaluation by drawing ideas of the aware evaluation, while keeping running times moderate. We then experimentally evaluate lazy$^+$ comparing it against the lazy and aware evaluations. Experimental results show the effectiveness of lazy$^+$: not only running times are much lower that those of the aware evaluation, but they are even lower than those of the lazy one, because of newly introduced optimizations. As for the quality of results, lazy$^+$ is placed between the lazy and aware algorithms, thereby achieving a good balance between computation time and quality of results.

4.2 Experimental Evaluation of Approximation Algorithms

In this section, we report on an experimental evaluation we conducted to evaluate approximation algorithms in terms of efficiency and quality of the results.

First, we recall the approximation algorithms we dealt with in Chapter 2. These algorithms leverage conditional tables and the conditional evaluation of relational algebra. The conditional evaluation returns conditional tuples $\langle t, \phi \rangle$, the expression ϕ says under which condition t can be derived. Conditions are valuable information that can be exploited to determine which tuples are certain answers. By condition evaluation we mean a way of associating ϕ with a truth value (`true`, `false`, or `unknown`). The aim is to ensure that if ϕ evaluates to `true`, then t is a certain answer. Tuples' conditions can be evaluated in different ways. The basic ideas of the strategies leading to algorithms are as follows:

- The *eager evaluation* evaluates tuples' conditions right after each relational algebra operator has been applied, using three-valued logic.
- The *semi-eager* evaluation behaves like the eager one, but it better exploits equalities in conditions (by propagating values into tuples and conditions) to provide more accurate results.
- The *lazy evaluation* improves upon the semi-eager one by postponing conditions' evaluation until the set difference operator is encountered in the query.
- The *aware evaluation* provides even more accurate results and behaves as follows: it performs the conditional evaluation of the entire query, then it uses a set of rewriting rules to "simplify" conditions, and eventually it evaluates (simplified) tuples conditions.

With the same query and database, moving to more accurate strategies, that is, from the eager (resp. semi-eager, lazy) evaluation to the semi-eager (lazy, aware) one, we can obtain more certain answers, but running times might get higher.

Thus, there is a trade-off in choosing one of the algorithms: moving from the eager to the aware evaluation the complexity increases but more certain answers can be returned (still, all algorithms have

	200		400		600		800		1000	
	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
Eager	27	136	101	291	230	437	380	587	623	741
Semi-eager	28	143	104	303	232	456	391	610	632	763
Lazy	29	143	106	303	237	456	395	610	636	763
Aware	220	143	837	303	2,522	456	4,166	610	6,743	763

TABLE 4.1: Runtime (msecs), number of certain answers to Q_{se} (10% of nulls).

	100		1000		10000	
	Time	#T	Time	#T	Time	#T
Eager	7	82	623	741	68,597	7,324
Semi-eager	7	86	632	763	68,963	7,610
Lazy	7	86	636	763	68,255	7,610
Aware	53	86	6,743	763	783,522	7,610

TABLE 4.2: Runtime (msecs), number of certain answers to Q_{se} (10% of nulls).

polynomial time complexity). The four evaluation strategies have been implemented in Java. All experiments were run on an Intel i7 3770K 3.5 GHz, 64GB of memory, running Linux Mint 17.1. Datasets were generated using the DBGen tool of the TPC-H benchmark [29]. As the generated databases are complete, nulls were randomly inserted.

Semi-eager. In order to assess the benefits of the semi-eager evaluation, we measured the running time and the number of certain answers to the query $Q_{se} = R - \sigma_{\$2=c}(S)$, where c is a value randomly chosen from those in the second column of S. We considered datasets having 200-1000 tuples per relation in steps of 200. Notice that Q_{se} is a query where the propagation of the equality in the selection condition can yield benefits, and thus it might be worth applying the semi-eager evaluation rather than the eager one (which is indeed the case, as shown by the experimental results below). Also, 10% of the values in the database are (randomly introduced) nulls. Experimental results are reported in Table 4.1.

As expected, running times increase as more powerful evaluation strategies are applied. We can see that the percentage of additional certain answers that the semi-eager evaluation yields w.r.t. to the eager one ranges from 3% to 5%. There is no benefit in applying evaluation strategies more accurate than the semi-eager one, as the

	2% **of nulls**		4% **of nulls**		6% **of nulls**		8% **of nulls**		10% **of nulls**	
	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
Eager	503	758	539	755	593	743	606	740	623	741
Semi-eager	513	763	550	763	606	763	613	763	632	763
Lazy	535	763	562	763	612	763	625	763	636	763
Aware	6,135	763	6,467	763	6,585	763	6,708	763	6,743	763

TABLE 4.3: Runtime (msecs), number of certain answers to Q_{se} (DB size: 1000).

structure of the query does not have features that can be exploited by them.

Table 4.2 reports results for databases having 100, 1000, and 10,000 tuples per relation. We can see a trend similar to the one previously discussed for Table 1 - again, the percentage of additional certain answers that the semi-eager evaluation yields w.r.t. to the eager one ranges from 3% to 5%.

We also ran experiments with databases having 1000 tuples per relation, varying the null rate from 2% to 10% in steps of 2, see Table 4.3. The advantage of the semi-eager evaluation w.r.t. the eager one (in terms of additional certain answers) ranges in 0.5-3%.

Lazy. In order to assess the benefits of the lazy evaluation, we measured the running time and the number of certain answers of the query $Q_{lazy} = P - (R \cap (\sigma_{\$2 \neq c}(S)))$, where c is a value randomly chosen from the second column of S. Once again, we considered datasets having 200-1000 tuples per relation in steps of 200 and 10% of nulls. Experimental results are reported in Table 4.4. Running times increase as more accurate evaluation strategies are applied. The benefits of the lazy evaluation w.r.t. to the semi-eager one (in terms of additional certain answers) ranges from 2.5% to 6.45%. There is no benefit in applying the aware evaluation, as the structure of the query does not have features that can be exploited by it.

Results for databases having 100, 1000, and 10,000 tuples per relation are shown in Table 4.5, exhibiting a similar behavior - here the lazy yields 1.5% to 3.8% more certain answers than the semi-eager (Table 4.6). We also ran experiments with relations having 1000 tuples, varying the null rate from 2% to 10% in steps of 2. The advantage of the lazy evaluation w.r.t. the semi-eager one (in terms of additional certain answers) ranges in 0.4-4%.

	200		400		600		800		1000	
	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
Eager	71	153	249	279	552	421	958	565	1,655	710
Semi-eager	76	153	293	279	640	421	1,095	565	1,736	710
Lazy	355	157	1,392	297	3,210	444	5,559	590	8,660	737
Aware	638	157	2,460	297	5,643	444	9,884	590	15,920	737

TABLE 4.4: Runtime (msecs), number of certain answers to Q_{lazy} (10% of nulls).

	100		1000		10000	
	Time	#T	Time	#T	Time	#T
Eager	18	63	1,655	710	150,881	7,149
Semi-eager	19	63	1,736	710	170,491	7,149
Lazy	97	64	8,660	737	858,233	7,390
Aware	176	64	15,920	737	1,605,264	7,390

TABLE 4.5: Runtime (msecs), number of certain answers to Q_{lazy} (10% of nulls).

	2% **of nulls**		4% **of nulls**		6% **of nulls**		8% **of nulls**		10% **of nulls**	
	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
Eager	1,310	734	1,421	721	1,481	723	1,550	714	1,655	710
Semi-eager	1,368	734	1,429	721	1,532	723	1,702	714	1,736	710
Lazy	7,221	737	7,569	737	7,741	737	8,238	737	8,660	737
Aware	12,395	737	13,166	737	13,657	737	14,292	737	15,920	737

TABLE 4.6: Runtime (msecs), number of certain answers to Q_{lazy} (DB size: 1000).

Aware. Finally, to assess the benefits of the aware evaluation, we measured the running time and the number of certain answers to the query $Q_{aware} = P - (R - S)$ over datasets having 200-1000 tuples per relation. Also, 10% of the values in the database were (randomly introduced) nulls. Experimental results are reported in Table 4.7. The aware evaluation has the highest running times but it returns significantly more certain answers than the other algorithms, as the number of certain answer is always (at least) doubled.

Results for databases with 100, 1000, and $10,000$ tuples per relations are shown in Table 4.8. While for the first two databases the trend is similar to the one previously discussed, for the largest database the aware evaluation ran out of memory. We also ran experiments with a database having 1000 tuples per relation, varying the null rate from 2% to 10% in steps of 2. Results are reported in

	200		400		600		800		1000	
	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
Eager	100	13	374	41	826	66	1,472	84	2,632	100
Semi-eager	103	13	378	41	835	66	1,479	84	2,797	100
Lazy	401	13	1,489	41	3,334	66	5,803	84	12,645	100
Aware	39,306	36	334,604	93	1,053,816	143	2,311,524	192	5,589,977	231

TABLE 4.7: Runtime (msecs), number of certain answers to Q_{aware} (10% of nulls).

	100		1000		10000	
	Time	#T	Time	#T	Time	#T
Eager	26	15	2,632	100	238,100	1,269
Semi-eager	28	15	2,797	100	241,600	1,269
Lazy	107	15	12,645	100	1,187,733	1,269
Aware	3,263	25	5,589,977	231	Out of memory	Out of memory

TABLE 4.8: Runtime (msecs), number of certain answers to Q_{aware} (10% of nulls).

	2% **of nulls**		4% **of nulls**		6% **of nulls**		8% **of nulls**		10% **of nulls**	
	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
Eager	1,761	252	1,956	217	2,148	180	2,285	153	2,632	100
Semi-eager	1,824	252	2,068	217	2,220	180	2,382	153	2,797	100
Lazy	8,646	252	9,705	217	10,412	180	11,110	153	12,645	100
Aware	4,018,252	307	4,629,472	288	5,285,823	280	5,794,665	267	6,663,986	231

TABLE 4.9: Runtime (msecs), number of certain answers to Q_{aware} (DB size: 1000).

	Q_{se}		
	Time	`#true`	`#unknown`
Eager	623	741	167
Semi-eager	632	763	145
Lazy	636	763	145
Aware	6,743	763	145

TABLE 4.10: Runtime (msecs), number of answers with condition `true`, and number of answers with condition `unknown` for the query Q_{se}.

Table 4.9. We can see that the aware evaluation is again the one returning the highest number of certain answers, but with much higher running time.

Tables 4.10, 4.11 and 4.12 show the number of unknown answers in addition to true answers. Table 4.10 (resp. 4.11 and 4.12) extends the informative content of Table 4.1 (resp. 4.4 and 4.7), in the part referring to databases having 1000 tuples per relation.

	Q_{lazy}		
	Time	#true	#unknown
Eager	1,655	710	189
Semi-eager	1,736	710	189
Lazy	8,660	737	162
Aware	15,920	737	162

TABLE 4.11: Runtime (msecs), number of answers with condition `true`, and number of answers with condition `unknown` for the query Q_{lazy}.

	Q_{aware}		
	Time	#true	#unknown
Eager	2,632	100	731
Semi-eager	2,797	100	731
Lazy	12,654	100	731
Aware	5,589,977	231	600

TABLE 4.12: Runtime (msecs), number of answers with condition `true`, and number of answers with condition `unknown` for the query Q_{aware}.

Discussion. The experimental evaluation has confirmed what we were expecting from the theory, that is, moving to more powerful techniques we can get more certain query answers, but running times become higher. However, while the gaps in running time between eager and semi-eager and between semi-eager and lazy are somewhat mild, the gap between lazy and aware is significant. The reason is that the aware evaluation performs the conditional evaluation of the entire query and collapses conditions only after that. This means that long conditions need to be kept and manipulated, which makes the technique requiring more time and space than simpler ones. However, this has advantages in terms of quality of the results: longer conditions allows the aware algorithm to perform more refined analyses and thus return more certain query answers. A natural question then arises: can we devise a technique with a behaviour in the middle of the lazy and aware evaluations? Can we improve the lazy evaluation so as to return more certain query answers, drawing from the ideas that characterize aware, but without incurring in the high running times of the latter? We address these questions in the next section, where we propose a novel evaluation algorithm, called $lazy^+$, which indeed achieves a good trade-off between runtime and

quality of the results.

4.3 Novel Approach

The two key features of the aware evaluation are postponing condition evaluation until the very end (i.e., after the conditional evaluation of the entire query), and applying a set of simplification rules to conditions. In this section, we augment the lazy evaluation with a set of simplification rules (to better analyze conditions), which are applied when the difference operator is encountered. The atomic conditions involving only constants can be evaluated immediately, and substituted by the obtained result, which can be true or false. The set of simplification rules for conjunctions of simple conditions involving labelled nulls is reported next:

1. *Negation*: $\neg(\neg\phi) \vdash \phi$, $\neg(\phi_1 = \phi_2) \vdash (\phi_1 \neq \phi_2)$, $\neg(\phi_1 \neq \phi_2) \vdash (\phi_1 = \phi_2)$, $\neg(\beta < \alpha) \vdash \alpha \leq \beta$, $\neg(\beta \leq \alpha \vdash \alpha < \beta$, $\neg\texttt{unknown} \vdash \texttt{unknown}$, $\neg\texttt{true} \vdash \texttt{false}$, and $\neg\texttt{false} \vdash \texttt{true}$.

2. *Middle excluded*: $(\alpha \leq \beta) \wedge (\beta \leq \alpha) \vdash (\alpha = \beta)$;

3. *Contradiction*:

   (i) $(\alpha < \beta) \wedge (\beta \star \alpha) \vdash \texttt{false}$, where $\star \in \{<, \leq, =\}$;

   (ii) $(\alpha = \beta) \wedge (\beta \neq \alpha) \vdash \texttt{false}$;

   (iii) $(\alpha \star_1 \beta) \wedge (\beta' \star_2 \alpha) \vdash \texttt{false}$, where $\beta, \beta' \in \texttt{Const}, \beta < \beta', \star_1, \star_2 \in \{<, \leq, =\}$;

   (iv) $(\alpha \star \beta) \wedge (\alpha = \beta') \vdash \texttt{false}$, where $\beta, \beta' \in \texttt{Const}, \beta < \beta', \star \in \{<, \leq, =\}$.

4. *And-simplification*: $\phi \wedge \phi \vdash \phi, \phi \wedge \texttt{true} \vdash \phi$, and $\phi \wedge \texttt{false} \vdash \texttt{false}$.

5. *Equality*: $(\alpha = \alpha) \vdash \texttt{true}$ and $(\alpha \neq \alpha) \vdash \texttt{false}$.

The result of applying a rule $\phi' \vdash \phi''$ to a condition ϕ is the condition obtained by replacing every occurrence of ϕ' with ϕ''. We write $\phi \overset{*}{\vdash} \phi'$, where ϕ and ϕ' are conditions, if (i) ϕ' can be derived from ϕ by iteratively applying rules 1-5 along with the commutativity and associativity rules, and (ii) none of the rules 1-5 is applicable to any of the conditions in $[\phi']$, where $[\phi']$ is the set of all conditions that

can be obtained from ϕ' by iteratively applying the commutativity and associativity rules zero or more times. if $\phi \overset{*}{\vdash} \phi'$, we say that ϕ' is a *minimal* condition for ϕ. Intuitively, a minimal condition ϕ' is obtained by iteratively applying rules 1-5 and commutativity and associativity rules until none of the rules 1-5 can be applied to ϕ' or any other condition derivable from ϕ' by means of the commutativity and associativity rules.

There can be multiple minimal conditions of a condition ϕ, but they are all equivalent w.r.t. the commutativity and associativity rules (roughly speaking, they differ only w.r.t. the order of their terms), that is, if $\phi \overset{*}{\vdash} \phi'$ and $\phi \overset{*}{\vdash} \phi''$ then $\phi' \in [\phi'']$ and $\phi'' \in [\phi']$. Thus, we can talk about *the* minimal condition of ϕ, which we denote as $\texttt{minimal}(\phi)$.

The lazy$^+$ evaluation is defined as follows:

- $\texttt{Eval}^{\ell^+}(R, D) = \overline{R^D}$
- $\texttt{Eval}^{\ell^+}(Q_1 \cup Q_2, D) = \texttt{Eval}^{\ell^+}(Q_1, D) \dot{\cup} \texttt{Eval}^{\ell^+}(Q_2, D)$
- $\texttt{Eval}^{\ell^+}(Q_1 \cap Q_2, D) = \texttt{Eval}^{\ell^+}(Q_1, D) \dot{\cap} \texttt{Eval}^{\ell^+}(Q_2, D)$
- $\texttt{Eval}^{\ell^+}(Q_1 - Q_2, D) = \texttt{eval}^{\ell}(\texttt{Eval}^{\ell^+}(Q_1, D) \dot{-} \texttt{eval}^{s}(\texttt{minimal}(\texttt{Eval}^{\ell^+}(Q_2, D))))$
- $\texttt{Eval}^{\ell^+}(Q_1 \times Q_2, D) = \texttt{Eval}^{\ell^+}(Q_1, D) \dot{\times} \texttt{Eval}^{\ell^+}(Q_2, D)$
- $\texttt{Eval}^{\ell^+}(\sigma_\theta(Q), D) = \dot{\sigma}_\theta(\texttt{Eval}^{\ell^+}(Q, D))$
- $\texttt{Eval}^{\ell^+}(\pi_Z(Q), D) = \dot{\pi}_Z(\texttt{Eval}^{\ell^+}(Q, D))$

where $\texttt{eval}^s()$ and $\texttt{eval}^\ell()$ are defined as in Sections 2.4.3 and 2.4.2 respectively.

Given a query Q and a database D, we define:

$$\texttt{Eval}_t^{\ell^+}(Q, D) = \{t \mid \langle t, \texttt{true}\rangle \in \texttt{eval}^s(\texttt{Eval}^{\ell^+}(Q, D))\},$$

that is, the true answers are computed by (i) first, evaluating $\texttt{eval}^\ell(Q, D)$, yielding a c-table T, and (ii) then, evaluating $\texttt{eval}^s(T)$.

	1000		2000		3000		4000		5000	
	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
Lazy	92	554	330	1,118	663	1,678	1,256	2,225	2,015	2,766
Lazy$^+$	69	568	299	1,136	592	1,706	1,123	2,269	1,786	2,836
Aware	111	581	403	1,161	721	1,726	1,432	2,314	2,260	2,880

TABLE 4.13: Runtime (msecs), number of certain answers to Q_{lazy^+} (10% of nulls).

	10000		20000		30000		40000		50000	
	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
Lazy	8,202	5,586	44,606	11,114	104,412	16,766	180,375	22,343	382,442	27,896
Lazy$^+$	7,871	5,710	32,369	11,334	81,819	17,077	158,252	22,742	247,352	28,385
Aware	8,758	5,795	41,996	11,542	121,299	17,370	254,284	23,112	391,270	28,908

TABLE 4.14: Runtime (msecs), number of certain answers to Q_{lazy^+} (10% of nulls).

4.4 Experimental Evaluation of Lazy$^+$

In this section, we report an experimental evaluation of the Lazy$^+$ algorithm.

We used a database consisting of the following three relations: $Person(\underline{person\_id})$, $Manager(\underline{manager\_id}, salary)$, and $Employee(\underline{emp\_id}, salary, manager)$, where *Person* and *Manager* are complete relations and *Employee* is an incomplete relation with null values occurring in the *salary* attribute.

We used the following query:

$$Q_{lazy^+} = Person \setminus \pi_{\$1}(\sigma_{\$1=\$2 \wedge \$2>\$5}(\sigma_{\$2<2000}(Employee) \times Manager).$$

The results of the experiments are shown in Tables 4.13, 4.14 and 4.15. As expected, given a certain database, the performances of the lazy$^+$ evaluation, in terms of the number of certain answers, are placed, in each test, in the middle compared to those of the lazy and the aware approaches. What is surprising (positively) is that the execution times of the lazy+ approach not only outperforms those of the aware evaluation (as expected), but they are even better than those of the lazy evaluation. This highlights that the computational overhead introduced by the reduction of the logical expressions through the application of the aforementioned axioms, facilitates the calculation of evaluating the same expressions, leading, overall, to a reduction in execution times.

	2% **of nulls**		4% **of nulls**		6% **of nulls**		8% **of nulls**		10% **of nulls**	
	Time	#T	Time	#T	Time	#T	Time	#T	Time	#T
Lazy	266,452	28,595	299,288	28,385	306,495	28,227	364,515	28,097	382,442	27,896
Lazy$^+$	220,864	28,701	275,353	28,618	280,050	28,534	255,678	28,498	247,352	28,385
Aware	284,279	28,789	319,760	28,827	331,558	28,834	358,013	28,875	391,270	28,908

TABLE 4.15: Runtime (msecs), number of certain answers to Q_{lazy^+} (DB size: 50, 000).

4.5 Discussion

Certain answers are a principled manner to answer queries on incomplete databases. Since their computation is a coNP-hard problem, recent research has focused on developing polynomial time algorithms providing under-approximations.

We have provided an experimental evaluation of recently proposed approximation algorithms. Results have shown some limits of more powerful techniques in terms of efficiency. To cope with this issue, we have introduced a novel optimized evaluation strategy and experimentally evaluated it, showing that it achieves a good balance between running time and quality of the results.

Chapter 5

Approximate Consistent Query Answering over Inconsistent Knowledge Bases

Consistent query answering is a principled approach for querying inconsistent knowledge bases. It relies on two central notions: a *repair*, that is, a maximal consistent subset of the facts in the knowledge base, and a *consistent query answer*, that is, a query answer entailed by every repair of the knowledge base. This chapter presents the ACQUA system, which allows users to query inconsistent knowledge bases [36]. Specifically, equality generating dependencies are considered. Different from the standard notion of repair, where entire facts are deleted to restore consistency (which might lead to loss of useful information), the repair strategy adopted by ACQUA performs value updates within facts, thereby preserving more information in the knowledge base. An inconsistent knowledge base can admit multiple repairs; the ACQUA system computes a compact representation of all of them, called *universal repair*, which is also leveraged for query answering. Since consistent query answering is intractable in the considered setting, ACQUA implements a polynomial time algorithm to compute a sound (but not necessarily complete) set of consistent query answers.

5.1 Introduction

In this chapter we will use the notion of *knowledge base*. Basically a knowledge base is a pair (D, Σ), where D is a database and the Σ is a set of integrity constraints. For our purposes, Σ is a set of equality-generating dependencies (EGDs), which are one of the two major

types of data dependencies - the other major type consists of tuple-generating dependencies (TGDs) - and can model several kinds of constraints commonly arising in practice, such as functional dependencies and thus also key dependencies.

Example 5.1. Consider the database D comprehensive of the only relation `works`

works

john	cs	nyc
bob	math	rome
mary	math	rome

and the ontology Σ consisting of the following equality-generating dependency σ:

$$\texttt{works}(E_1, D, C_1) \wedge \texttt{works}(E_2, D, C_2) \rightarrow C_1 = C_2.$$

The fact $\texttt{works}(\texttt{john}, \texttt{cs}, \texttt{nyc})$ states that `john` is an employee working in the `cs` department located in `nyc`. The other facts can be similarly read. The dependency σ says that every department must be located in a single city. Thus, the knowledge base (D, Σ) consists of the table `works` and the EGD σ. All the facts in D satisfy σ, since for each pair of tuples with the same constants in correspondence of the second column (department), they have the same constants also in correspondence of the third column (city). In the example we have only two facts with the same value in the second column, $\texttt{works}(\texttt{bob}, \texttt{math}, \texttt{rome})$ and $\texttt{works}(\texttt{mary}, \texttt{math}, \texttt{rome})$, and they also coincide in the third column.

□

Reasoning in the presence of inconsistent information is a problem that has attracted much interest in the last decades. Many inconsistency-tolerant semantics for query answering have been proposed, and most of them rely on the central notions of *consistent query answer* and *repair*.

A consistent answer to a query is a query answer that is entailed by every repair, where a repair is a "maximal" consistent subset of the facts of the knowledge base. Different maximality criteria have been investigated, but all the resulting notions of repair share the same drawback: a fact is either kept or deleted altogether, and deleting entire facts can cause loss of "reliable" information, as illustrated in the following example.

Example 5.2. Consider the knowledge base (D, Σ) where D contains the following facts:

works

john	cs	nyc
john	math	rome
mary	math	sidney

and Σ is an ontology consisting again of the following equality-generating dependency σ:

$$\texttt{works}(E_1, D, C_1) \wedge \texttt{works}(E_2, D, C_2) \rightarrow C_1 = C_2.$$

Clearly, the last two facts violate σ, so every repair would discard either of them.

If we pose a query asking for the employees' name, the only consistent answer is `john`. However, intuitively, we might consider reliable the information on `mary` being an employee, as the only uncertainty concerns the `math` department and its city - roughly speaking, the information in the first column of the `works` table can be considered "clean". However, dropping entire facts cause loss of useful information. □

To overcome the drawback illustrated above, [51] have recently proposed a notion of repair based on updating values within facts. Update-based repairing allows for rectifying errors in facts without deleting them altogether, thereby preserving consistent values.

Example 5.3. Consider again the knowledge base of Example 5.2. Using value updates as the primitive to restore consistency, and assuming that the only uncertain values are `math`'s cities, we get the following two repairs:

john	cs	nyc
john	math	rome
mary	math	rome

john	cs	nyc
john	math	sidney
mary	math	sidney

If we ask again for the employees' name, both `mary` and `john` are consistent answers. □

Thus, an inconsistent knowledge base can admit multiple repairs. They can be compactly represented by a "universal" repair, which turns out to be a valuable tool to compute approximate query answers. The basic idea is illustrated in the following example.

Example 5.4. A universal repair for the two repairs of Example 5.3 is reported below:

john	cs	nyc
john	math	$\perp_1$
mary	math	$\perp_1$

$\perp_1 =$ rome$\vee$ $\perp_1 =$ sidney

where $\perp_1$ is a labeled null, and the "global" condition at the bottom restricts the admissible values for $\perp_1$, which are rome and sydney. Every replacement of nulls with constants complying with the global condition yields a repair. In fact, it is easy to see that when $\perp_1$ is replaced with either rome or sydney the global condition is satisfied and the two repairs in Example 5.3 are obtained.

We exploit such a representation for query answering, by combining the condition of the universal repair with provenance information during query evaluation. For instance, if we ask for the departments that are not located in nyc, we get math with condition $\perp_1 \neq$ nyc, which combined with the global condition allows us to conclude that math is a consistent answer (as its city is either rome or sydney, and thus cannot be nyc for sure). □

Since consistent query answering in the considered setting is coNP-complete, [51] developed a polynomial time approximation algorithm to compute a sound but possibly incomplete set of consistent query answers

This chapter presents the ACQUA system, which implements the framework proposed in [51]. Specifically, the system allows users to provide a knowledge base and a query, and computes a universal repair as well as a sound set of consistent answers to the provided query.

Now we provide the definitions of *argument* of a set of EGDs Σ and *argument and dependency graphs*.

Definition 5.5. Let Σ be a set of EGDs. An *argument* of Σ is an expression of the form $p[i]$, where p is an n-ary predicate appearing in Σ and $1 \leq i \leq n$.

Definition 5.6. The *argument graph* of a set Σ of EGDs is a directed graph $G_\Sigma = (V, E)$, where V is the set of all arguments of Σ, and E contains a directed edge from $p[i]$ to $q[j]$ labeled σ iff there is an EGD $\sigma \in \Sigma$ such that:

- the body of σ contains an atom $p(t_1, \ldots, t_n)$ such that either t_i is a constant or t_i is a variable occurring more than once in the body of σ, and
- the body of σ contains an atom $q(u_1, \ldots, u_m)$ such that u_j is a variable also appearing in the head of σ.

The *dependency graph* of Σ is a directed graph $\Gamma_\Sigma = (\Sigma, \Omega)$, where Ω is the following set:

$$\{(\sigma_1, \sigma_2) | G_\Sigma = (V, E) \wedge (p([i], q[j], \sigma_1), (q[j], r[k], \sigma_2) \in E\}.$$

We say that Σ is *acyclic* if its dependency graph is acyclic. Clearly, the set consisting of the EGDs in Example 5.1 is acyclic. In the following we will consider acyclic set of EGDs only.

The rest of this chapter is organized as follows: first we discuss related work and illustrate the approximation algorithm proposed in [51]; then, we provide an overview of the ACQUA system, discussing its architecture and how to interact with the system; finally, we draw conclusions.

5.2 Related work

Reasoning in the presence of inconsistent information is a problem that has attracted a great deal of interest in the AI and database communities. Consistent query answering was first proposed in [5].

Query answering under various inconsistency-tolerant semantics for ontologies expressed in DL languages has been studied in [3, 9]. Query answering in the presence of inconsistent ontologies expressed in (different fragments of) Datalog+/ has been investigated in [77, 76].

Several notions of maximality for a repair have been considered in [11]. [13] proposed an approach for the approximation of consistent query answers from above and from below. [41] proposed an approach based on three-valued logic to compute a sound but possibly incomplete set of consistent query answers.

All the approaches above adopt the most common notion of repair, where whole facts are removed. This can cause loss of information. In real-life scenarios, it might well be the case that facts have much more attributes and only a few of them are involved in inconsistencies, leading to significant loss of useful data.

There have also been different proposals adopting a notion of repair that allows values to be updated [16, 10]. In [16, 45, 46] focus on FDs only. [10] allows only numerical attributes to be updated, one primary key per relation is allowed, but keys are assumed to be satisfied by the original database, and thus no repairing is possible w.r.t. keys. [37] considers numerical databases and a different class of (aggregate) constraints. The repair strategy of [87] is quite different in that any value in the database can be updated (including on the left-hand side of FDs), provided that the set of changes is "minimal" and yields a consistent database. None of the approaches above has investigated the approximate computation of consistent query answers - notice that [16, 10] have proposed approximation algorithms for computing a repair with minimum distance from the original database, and [45] consider approximate probabilistic query answers.

Approximation algorithms for computing sound but possibly incomplete sets of query answers in the presence of nulls have been proposed in [54, 35], but no dependencies are considered therein, and thus the database is assumed to be consistent.

5.3 Approximation Algorithm

In the introduction of this chapter, we have illustrated the repair strategy adopted by the ACQUA system and the notion of a universal repair, which is a compact way of representing all repairs. The universal repair can be leveraged for computing a sound set of consistent query answers, which is the approach adopted by the approximation algorithm proposed in [51]. The basic idea is illustrated in the following example.

Example 5.7. Consider the knowledge base of Example 5.2 and the query asking for the pairs of departments located in different cities.

The first step of our approach consists of computing a universal repair, which is shown in Example 5.4.

The second step consists of adding the condition true to every fact of the universal repair, which leads to the following "conditional" instance (i.e., an instance where every fact is associated with a condition).

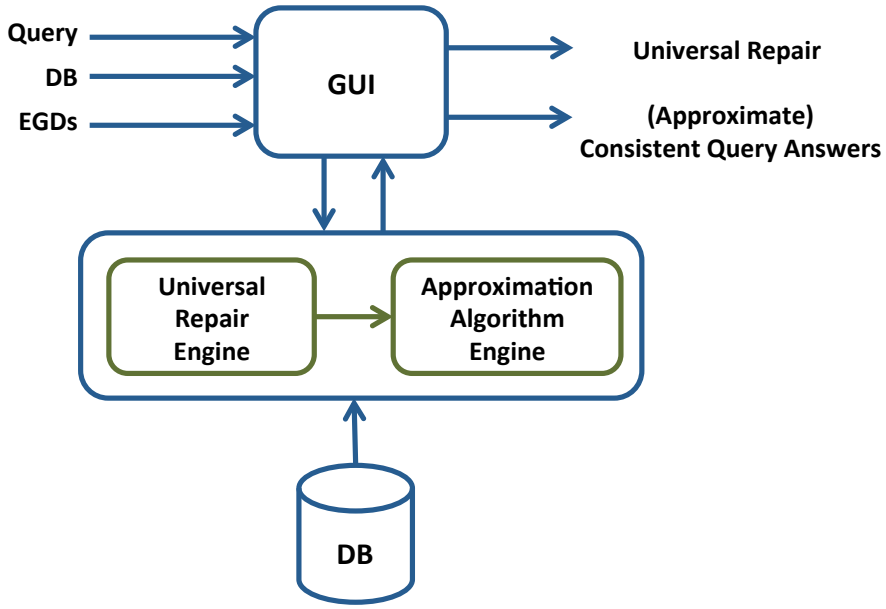

FIGURE 5.1: System Architecture.

john	cs	nyc	true
john	math	$\perp_1$	true
mary	math	$\perp_1$	true

$\perp_1 =$ rome$\vee$ $\perp_1 =$ sidney

The third step consists of conditionally evaluating the query over the conditional instance above. Intuitively, the conditional evaluation of a query yields a set of facts, each associated with a formula stating under which condition the fact can be derived. The result is as follows:

cs	cs	nyc $\neq$ nyc
cs	math	nyc $\neq \perp_1$
math	cs	$\perp_1 \neq$ nyc
math	math	$\perp_1 \neq \perp_1$

The last step tries to understand when a fact is certainly derived, that is, whether a fact can be derived no matter how nulls are replaced (under the constraints stated by the global condition of the universal repair). This is done by looking at the global condition of the universal repair along with the local conditions in the result of the conditional evaluation.

For instance, for the first fact above, it is easy to see that its condition, namely nyc $\neq$ nyc, is always false. Thus, (cs, cs) is not returned

as a consistent query answer. A similar argument applies to the last fact: $\perp_1 \neq \perp_1$ is always false no matter how $\perp_1$ is replaced.

Consider now the second and third facts. The global condition of the universal repair says that $\perp_1$ must be either `rome` or `sydney`. The local conditions of the two facts say that those facts are derived when $\perp_1$ is different from `nyc`. By combining the global and local conditions, we can conclude that the facts are always derived, as $\perp_1$ is always different from `nyc`. Thus, (`cs`, `math`) and (`math`, `cs`) are returned by the approximation algorithm - indeed, they are consistent query answers. □

5.4 System Overview

The ACQUA system has been implemented in Java. The system architecture is depicted in Figure 5.1. There are 4 main components: a graphical user interface (GUI), the algorithm to compute a universal repair, the algorithm to compute an approximate set of consistent query answers, and the database.

The GUI allows users to specify the query to be evaluated and the knowledge base (i.e., a database with a set of EGDs). It provides the knowledge base to the universal repair engine in order to compute a universal repair. It also provides the knowledge base and the query to the approximation algorithm engine in order to compute approximate consistent query answers. Also, the GUI displays a universal repair (which receives from the universal repair engine) and the query answers (received from the approximation algorithm engine). The approximation algorithm engine leverages the universal repair to compute a sound set of consistent answers to the query provided by the GUI. The DB component simply stores and handles access to the facts of the knowledge base.

The ACQUA system provides also an API which allows third party applications to interact with the system.

We now go into the details of how to interact with the system. The ACQUA's graphical user interface is shown in Figure 5.2. A typical interaction with the system involves the following steps:

1. The user specifies the input knowledge base, consisting of a database and an (acyclic) set of EGDs. Specifically, for each table in the database, its location in the file system is provided. Tables are supposed to be in csv format. EGDs can be specified in a text area, loaded from a file, and saved to a file.

2. The user specifies the query to be evaluated (this is given in Datalog syntax and has to be a stratified program). Queries can be loaded from and saved to files.

3. The user launches the computation of a universal repair (this does not involve query evaluation). After the computation has finished, a universal repair is displayed.

4. The user launches the query evaluation. If a universal repair has not been computed, first its computation is performed, and then the query is evaluated. After the evaluation has finished, the result (a sound set of consistent query answers) and the execution time are displayed.

5.5 Discussion

We presented ACQUA, a system for approximate query answering over inconsistent knowledge base consisting of EGDs. The main features that characterize the ACQUA system are three. It relies on a notion of repair allowing values within facts to be updated, which allow users to preserve more information of the original database. The system adopts a compact representation of all repairs (called universal repair), which can be leveraged for query answering. The ACQUA system implements a polynomial time approximation algorithm to compute an under-approximation of consistent query answers.

As directions for future work we plan to extend the ACQUA system by providing support for more general classes of dependencies, such as TGDs or arbitrary EGDs, and by integrating further approximation algorithms.

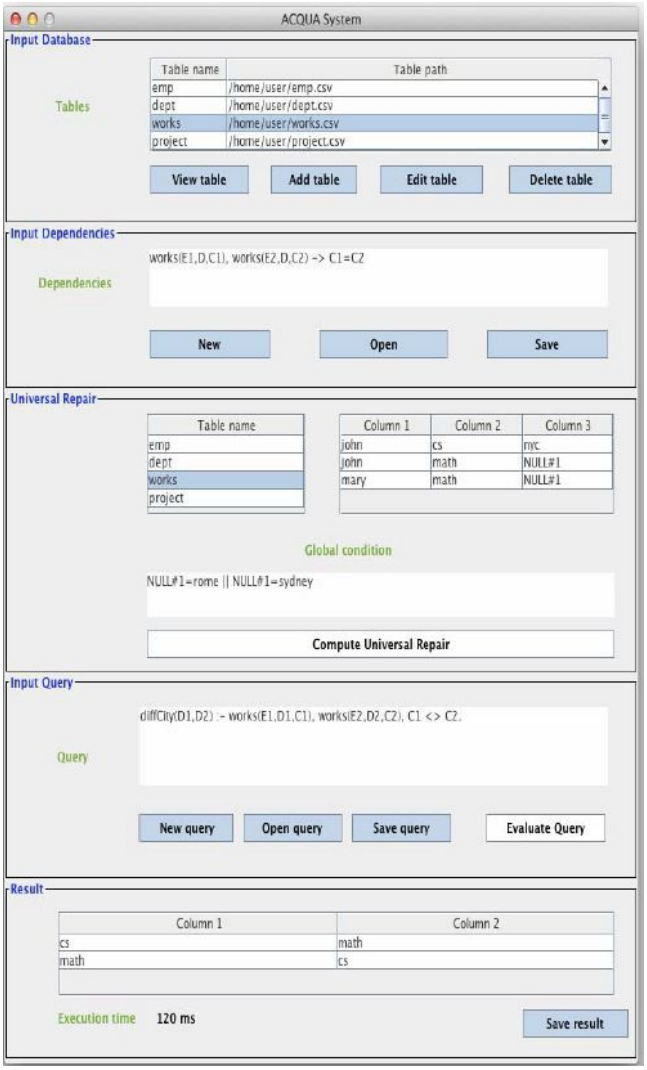

FIGURE 5.2: ACQUA system's GUI.

Chapter 6

Probabilistic Answers over Inconsistent Knowledge Bases

Consistent query answering is a generally accepted approach for querying inconsistent knowledge bases. A consistent answer to a query is a tuple entailed by every repair, where a repair is a consistent database that "minimally" differs from the original (possibly inconsistent) one. This is a somewhat coarse-grained classification of tuples into consistent and non-consistent which does not provide much information about the non-consistent tuples (e.g., a tuple entailed by 99 out of 100 repairs might be considered "almost consistent").

To overcome this limitation, we propose a probabilistic approach to querying inconsistent knowledge bases, which provides more informative query answers by associating a degree of consistency with each query answer by associating a probability to each repair, depending on the changes needed to obtain it [20].

6.1 Introduction

There has been a great deal of work on extracting reliable information from inconsistent data, in both the AI and database communities. To deal with this problem, many semantics of query answering have been proposed. Most of them are based on the *consistent query answering* framework [5]. The idea is that a tuple should be considered a consistent answer to a query posed over an inconsistent knowledge base if the tuple is a query answer in every repair, where a *repair* is a consistent database that "minimally" differs from the original one. Different variants of the consistent query answering problem have been investigated depending on the minimality

criterion - e.g., see [11] - or the employed repair primitive to restore consistency - e.g., see [9].

Regardless of the specific minimality criterion and repair primitive, all the resulting frameworks share the same drawback, which is a dichotomic classification of tuples in either consistent or non-consistent ones. This can provide very little information to users in many cases, as illustrated in the following example.

Example 6.1. Consider the knowledge base (D, Σ), where D contains the following facts:

employee

bob	cs	nyc
mike	cs	nyc
alice	cs	paris

and Σ is an ontology consisting of the following equality-generating dependency σ:

$$\texttt{employee}(E_1, D, C_1), \texttt{employee}(E_2, D, C_2) \rightarrow C_1 = C_2$$

As an example, the fact $\texttt{employee}(\texttt{bob}, \texttt{cs}, \texttt{nyc})$ states that bob is an employee working in the cs department located in nyc. The dependency σ says that every department must be located in a single city. Clearly, the first two facts are conflicting with the last one. There are two repairs for this inconsistent knowledge base, which are as follows (more details on the employed notion of repair will be provided in the following):

$R_1 =$

bob	cs	nyc
mike	cs	nyc
alice	cs	nyc

$R_2 =$

bob	cs	paris
mike	cs	paris
alice	cs	paris

Repair R_1 is obtained from the original database by replacing paris with nyc, while R_2 is obtained by replacing nyc with paris. The query asking for the city of the cs department has no consistent answers. Thus, a user trying to extract this information from the knowledge base is left with no answer altogether, all she/he gets is the empty set. □

To overcome the limitation illustrated in the example above, we propose a probabilistic approach to querying inconsistent knowledge bases whose aim is to provide more informative query answers than the classical consistent query answering framework. The main

idea is to give a "degree of consistency" to every repair, which is then taken into account to assign a degree of consistency to query answers.

Example 6.2. Consider again the scenario of Example 6.1. The degree of consistency of R_1 is $2/3$, as `nyc` is supported by two facts out of three in the original knowledge base, while the degree of consistency of R_2 is $1/3$, as `paris` is supported by only one fact out of three.

Consider again the query asking for the city of the `cs` department. Rather than providing no information (like classical consistent query answers do), our query answers are `nyc` with confidence $2/3$ (as this is entailed by R_1), and `paris` with confidence $1/3$ (as this is entailed by R_2), which is much more informative than returning nothing. □

The previous examples illustrate the limitation of providing only certain information and how this might be overcome by assigning a degree of consistency to query answers (notice that consistent query answers are still provided they have degree 1).

In this chapter, we consider knowledge bases where dependencies are expressed by means of *equality generating dependencies* (EGDs). Equality generating dependencies are one of the two major types of data dependencies - the other major type consists of *tuple-generating dependencies* (TGDs) - and can model several kinds of constraints arising in practice, such as functional dependencies and thus key dependencies.

In the presence of EGDs, the two main approaches to restore consistency are to perform fact deletions or fact updates. A drawback of the former is that entire facts are deleted to resolve inconsistency, even if they may still contain "reliable" information. Thus, in this chapter we adopt a notion of repair based on fact updates (we will provide a precise definition in the following). Nonetheless, the ideas developed here can be used also when a notion of repair based on fact deletions is adopted.

As we show, providing more informative query answers comes at a price: it is #P-hard. In light of this result, we discuss further developments providing polynomial time algorithms to compute approximate query answers.

Most inconsistency-tolerant approaches in the literature adopt the classical notion of repair (via fact deletions/insertions) and consistent query answer [65, 82, 13, 77, 76], while in this chapter we propose a probabilistic generalization where repairs use fact updates, akin to [37, 87, 16, 51], and query answers are probabilistic. Some

works on probabilistic query answering on inconsistent knowledge bases exist [4, 46, 17], but [4] and [46] only consider restricted classes of dependencies, while [17] deals with the classical notion of repair.

6.2 Preliminaries

We assume the existence of the following pairwise disjoint (countably infinite) sets: a set $\texttt{Const}$ of constants, a set $\texttt{Var}$ of variables, and a set $\texttt{Null}$ of *labeled nulls*. Nulls are denoted by the symbol $\perp$ subscripted with natural numbers. A *term* is a constant, variable, or null. We also assume a set of *predicates*, disjoint from the aforementioned sets, with each predicate being associated with an *arity*, which is a non-negative integer.

An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate and the t_i's are terms. We write an atom also as $p(\mathbf{t})$, where $\mathbf{t}$ is a sequence of terms. An atom without variables is also called a $fact$. An *instance* is a finite multiset of facts. A *database* is a finite set of facts containing constants only. An instance containing only constants is said to be *complete*. Notice that, as opposed to databases, instances can contain duplicates - as shown in the following, instances are used in intermediate steps during repairs' computation and it is needed to keep duplicates to count the number of modifications being made. We assume the existence of a function $\texttt{db}$ converting a complete instance to a database by eliminating duplicates.

A *homomorphism* is a mapping $h : \texttt{Const} \cup \texttt{Var} \cup \texttt{Null} \to \texttt{Const} \cup \texttt{Var} \cup \texttt{Null}$ that is the identity on $\texttt{Const}$. Homomorphisms are also applied to atoms and (multi) sets of atoms in the natural fashion, that is, $h(p(t_1, \ldots, t_n)) = p(h(t_1), \ldots, h(t_n))$, and $h(S) = \{h(A) | A \in S\}$ for every (multi) set S of atoms. A *valuation* is a homomorphism ν whose image is $\texttt{Const}$, that is, $\nu(t) \in \texttt{Const}$ for every $t \in \texttt{Const} \cup \texttt{Var} \cup \texttt{Null}$.

With a slight abuse of notation, we sometimes treat a conjunction of atoms as the set of its atoms. An instance J *satisfies* σ, denoted $J \models \sigma$, if whenever there exists a homomorphism h s.t. $h(\varphi(\mathbf{x})) \subseteq J$, then $h(x_i) = h(x_j)$. An instance J *satisfies* a set Σ of EGDs, denoted $J \models \Sigma$, if $I \models \sigma$ for every $\sigma \in \Sigma$. A *knowledge base* (KB) is a pair (D, Σ), where D is a database and Σ is a finite set of EGDs. We say that the knowledge base is *consistent* if $D \models \Sigma$, otherwise it is *inconsistent*.

In the rest of the chapter we consider acyclic sets of EGDs only. For acyclic sets of EGDs we can define a partial order $(\Sigma, <)$ where

EGDs in Σ are ordered by reachability in Γ_Σ. The result of evaluating a query Q over a database D is denoted $Q(D)$.

Repairs The notion of repair used here is based on updating facts and assumes that conflicting facts denote an attribute-level uncertainty in the data [51, 37, 10, 87, 16, 40]. Alternative approaches based on fact deletions assume that conflicting facts denote a *tuple-level uncertainty* [17, 83, 16]. The computation of repairs consists of a chase-like procedure that acts as follows: whenever an EGD $\varphi(\mathbf{x}) \rightarrow x_i = x_j$ is not satisfied by a database D, i.e. there exists an homomorphism h such that $D \models h(\varphi(\mathbf{x}))$ and $h(x_i) \neq h(x_j)$, we have to enforce it by making the two values equal, that is either $h(x_i)$ replaces $h(x_j)$ or vice versa. A repair is obtained through an exhaustive application of this repair step. Note that, although we follow the partial order $(\Sigma, <)$ to choose the EGD to enforce, there is a non-deterministic choice to be made when selecting an EGD and when updating values; this may lead to multiple repairs.

Example 6.3. Consider the set of EGDs

$$\Sigma = \{\sigma_1 : \texttt{employee}(X, Y_1, Z_1) \wedge \texttt{employee}(X, Y_2, Z_2) \rightarrow Y_1 = Y_2,$$
$$\sigma_2 : \texttt{employee}(X_1, Y, Z_1) \wedge \texttt{employee}(X_2, Y, Z_2) \rightarrow Z_1 = Z_2\}$$

and the database D:

bob	cs	rome
bob	math	rome
alice	math	nyc

By enforcing σ_1 into the first two facts, either `cs` or `math` can be chosen as `bob`'s department. If the latter is chosen, then the instance D' below is obtained.

Suppose now that σ_2 is enforced into the last two facts of D'. Then, either `rome` or `nyc` can be chosen as `math`'s city. If the former is chosen, we obtain the instance D'':

$D' =$

bob	math	rome
bob	math	rome
alice	math	nyc

$D'' =$

bob	math	rome
bob	math	rome
alice	math	rome

No further dependency enforcement is applicable at this point and thus the database derived from D'' by eliminating duplicates is a repair. □

The example 6.3 informally illustrated the basic idea of the repair strategy we adopt. In the next section, we formally define it and show how to associate probabilities to repairs on the basis of the modifications that yielded them.

Notice that it might be possible to restore consistency in other different ways. For instance, in the first step of the example 6.3, one may modify the employee names. However, we do not consider this option because it is unclear which (different) values should be assigned (any constant in `Const` is a candidate value). For instance, `bob` in the first fact might be replaced with `rome`, but this is somewhat arbitrary and indeed does not make much sense. In contrast, our repair strategy chooses candidate values that are somehow "justified" by the content of the database (e.g., in the example above, `bob` works for either the `cs` or the `math` department). Moreover, when EGDs are key dependencies, the aforementioned way of restoring consistency may lead to the introduction of entities that are not meaningful. Indeed, our choice has been made by different approaches relying on value updates - e.g., [16]. For special classes of EGDs, the repair strategy based on updating values coincides with the repair strategy based on fact deletions.

6.3 Probabilistic Repairs

One problem of the classical notion of consistent query answer is that query answers can provide little information in the presence of conflicting information. The alternative approach proposed in this chapter is to compute probabilistic answers by taking into account in to what extent information is updated to restore consistency.

In this chapter, we define probabilistic repairs and probabilistic query answers. In the next section, we show how to compute a compact representation of all probabilistic repairs. In both cases, we will use probabilistic instances, which are used to represent a single probabilistic repair in this section and are used to compactly represent all probabilistic repairs in the next section. Probabilistic instances are instances augmented with a set of probability assignments - intuitively, expressions stating conditions on nulls and probabilities on the values they can take.

More formally, let C be the set of all expressions, called *conditions*, that can be built using the standard logical connectives $\wedge$,$\vee$,$\neg$, and expressions of the form $t_i = t_j$, `true` and `false`, where $t_i, t_j \in$ `Const` $\cup$ `Null`. We will also use $t_i \neq t_j$ as a shorthand for $\neg(t_i = t_j)$. A

homomorphism h *satisfies* a condition φ, denoted $h \models \varphi$, if $h(\varphi)$ is true.

A *probability assignment* is an expression of the form $P(\varphi_1|\varphi_2) = p$, where φ_1, φ_2 are conditions and $p \in [0,1]$.

A homomorphism h *satisfies* a probability assignment $P(\varphi_1|\varphi_2) = p$ if the following holds true: if $h \models \varphi_2$ then $h \models \varphi_1$. Also, h *satisfies* a set Φ of probability assignments, denoted $h \models \Phi$, if h *satisfies* every probability assignment in Φ. For ease of presentation, a probability assignment of the form $P(\varphi_1|\texttt{true}) = p$ will be simply written as $P(\varphi_1) = p$.

A *probabilistic instance* (PI) is a pair $K = (J, \phi)$, where J is an instance and ϕ is a finite set of probability assignments. For any valuation ν, we define:

$$Pr(\nu, K) = \Pi\{p \mid P(\varphi_1 \mid \varphi_2) = p \in \Phi \text{ and } \nu \models \varphi_2\}.$$

The semantics of K is given by the set of its *probabilistic worlds*, that is, the set of pairs (instance, probability) defined as follows:

$$pw(K) = \{(\nu(J), Pr(\nu, K)) | \nu \text{ is a valuation s.t. } \nu \models \Phi \wedge Pr(\nu, K) > 0\}.$$

The probabilistic repairs of an inconsistent knowledge base (D, Σ) are computed starting from the probabilistic instance $(D, \emptyset)$ and iteratively enforcing EGDs (following a topological sorting of Γ_Σ). During the repair process we keep track of which values have been updated and how many modifications have been applied using labeled nulls and probability assignments. The enforcement of an EGD, which we call probabilistic repair step, is informally shown in the next example.

Example 6.4. Consider the knowledge base (D, Σ), where D and Σ are from Example 6.3. Starting from the probabilistic instance $(D, \emptyset)$, by enforcing σ_1 into the first two facts, either `cs` or `math` can be chosen as `bob`'s department. Therefore we obtain the following two (alternative) probabilistic instances K_1 (left) and K_2 (right):

bob	$\perp_1$	rome
bob	$\perp_1$	rome
alice	math	nyc

$P(\perp_1 = \texttt{cs}) = 1/2$

bob	$\perp_1$	rome
bob	$\perp_1$	rome
alice	math	nyc

$P(\perp_1 = \texttt{math}) = 1/2$

The only probabilistic world of K_1 is $(D_1, 1/2)$, while the only probabilistic world of K_2 is $(D_2, 1/2)$, where D_1 and D_2 are as follows (and are obtained by replacing $\perp_1$ with `cs` and `math`, respectively):

$D_1 =$

bob	cs	rome
bob	cs	rome
alice	math	nyc

$D_2 =$

bob	math	rome
bob	math	rome
alice	math	nyc

As $D_1 \models \Sigma$, $(\mathtt{db}(D_1), 1/2)$ is a probabilistic repair. On the other hand, $D_2 \not\models \Sigma$, and thus we need to keep applying pr-steps over K_2. By enforcing σ_2 over the first and third facts of K_2 we can get the following two (alternative) probabilistic instances K_3 (left) and K_4 (right):

bob	$\perp_1$	$\perp_2$
bob	$\perp_1$	rome
alice	math	$\perp_2$

$P(\perp_1 = \mathtt{math}) = 1/2$
$P(\perp_2 = \mathtt{rome}) = 1/2$

bob	$\perp_1$	$\perp_2$
bob	$\perp_1$	rome
alice	math	$\perp_2$

$P(\perp_1 = \mathtt{math}) = 1/2$
$P(\perp_2 = \mathtt{nyc}) = 1/2$

The only probabilistic world of K_3 is $(D_3, 1/4)$, while the only probabilistic world of K_4 is $(D_4, 1/4)$, where D_3 and D_4 are as follows:

$D_3 =$

bob	math	rome
bob	math	rome
alice	math	rome

$D_4 =$

bob	math	nyc
bob	math	rome
alice	math	nyc

Since $D_3 \models \Sigma$, $(\mathtt{db}(D_3), 1/4)$ is a probabilistic repair. On the other hand, $D_4 \not\models \Sigma$, and thus further pr-steps need to applied to K_4. By enforcing σ_2 over the first two facts of K_4 we can get the following two (alternative) probabilistic instances K_5 (left) and K_6 (right):

bob	$\perp_1$	$\perp_3$
bob	$\perp_1$	$\perp_3$
alice	math	$\perp_3$

$P(\perp_1 = \mathtt{math}) = 1/2$
$P(\perp_2 = \mathtt{nyc}) = 1/2$
$P(\perp_3 = \mathtt{rome}) = 1/3$

bob	$\perp_1$	$\perp_3$
bob	$\perp_1$	$\perp_3$
alice	math	$\perp_3$

$P(\perp_1 = \mathtt{math}) = 1/2$
$P(\perp_2 = \mathtt{nyc}) = 1/2$
$P(\perp_3 = \perp_2) = 2/3$

The only probabilistic world of K_5 is $(D_5, 1/12)$, while the only probabilistic world of K_6 is $(D_6, 1/6)$, where D_5 and D_6 are as follows:

$D_5 =$

bob	math	rome
bob	math	rome
alice	math	rome

$D_6 =$

bob	math	nyc
bob	math	nyc
alice	math	nyc

Since $D_5 \models \Sigma$ and $D_6 \models \Sigma$, both $(\mathtt{db}(D_5), 1/12)$ and $(\mathtt{db}(D_6), 1/6)$ are probabilistic repairs, and no further pr-step need to be applied.

Thus, the probabilistic repairs are $(\mathtt{db}(D_1), 1/2)$, $(\mathtt{db}(D_3), 1/4)$, $(\mathtt{db}(D_5), 1/12)$, and $(\mathtt{db}(D_6), 1/6)$. Notice that the sum of the probabilities of the probabilistic repairs is 1. Also, notice that D_3 and D_5 coincide. They might be replaced by a single probabilistic repair $(D', 1/4 + 1/12)$, where $D' = D_3 = D_5$. However, this does not affect query answering, that is, the probabilistic query answers are the same in both cases. □

6.4 Discussion

As a direction for future work, we planned to deepen the problem dealt with in this chapter. The idea is to present a polynomial time algorithm to compute approximate query answers where point probability are approximated by probability intervals. Thus, rather than providing query answers of the form (t, p), where t is a tuple and p is its probability, the approximation algorithm gives query answers of the form $(t, [\ell, u])$ guaranteeing that $p \in [\ell, u]$.

Further developments should consider more general frameworks with TGDs [17] and logic rules [52, 18, 19].

Conclusions

Query answering in the presence of missing and inconsistent information is a major problem which has been investigated through the years, and research has recently focused on a number of approaches with the objective to make the computation of certain and consistent query answers feasible in practice, settling for under-approximations.

In the first part of this thesis we dealt with incomplete information and we focused on polynomial time evaluation algorithms with correctness guarantees, that is, techniques computing a sound but possibly incomplete set of certain answers, being the exact computation of certain answers a coNP-hard problem. We illustrated existing approaches, pointing out that more accurate evaluation strategies have higher running times, but they pay off with more certain answers being returned. Thus, we presented the ACID system, a system prototype providing a suite of state-of-the-art approximation algorithms enabling users to choose the technique that best meets their needs in terms of balance between efficiency and quality of the results. The algorithms implemented and experimentally tested are those we called eager, semi-eager, lazy and aware evaluation strategies. Then, we showed how to query incomplete databases using the ACID system, allowing users to choose the technique to be run. Finally, we proposed a new strategy, called lazy$^+$ evaluation, which improves the lazy evaluation obtaining a good balance between running time and quality of the results. The experimental results we reported confirm the effectiveness of the technique.

In the second part of the thesis, the focus moved to consistent query answering, which is a principled approach for querying inconsistent knowledge bases. It relies on the notions of repair, that is, a maximal consistent subset of the facts in the knowledge base, and consistent query answer, that is, a query answer entailed by every repair of the knowledge base. We presented the ACQUA system, which allows users to query inconsistent knowledge bases. Specifically, the system considers equality generating dependencies. Different from the standard notion of repair, where entire facts are deleted

to restore consistency (which might lead to loss of useful information), the repair strategy adopted by ACQUA performs value updates within facts, thereby preserving more information in the knowledge base. An inconsistent knowledge base can admit multiple repairs; the ACQUA system computes a compact representation of all of them, called universal repair, which is also leveraged for query answering. Since consistent query answering is intractable in the considered setting, ACQUA implements a polynomial time algorithm to compute a sound (but not necessarily complete) set of consistent query answers. The classification of query answers into consistent and non-consistent ones does not provide much information about the non-consistent query answers (e.g., a query answer entailed by 99 out of 100 repairs might be considered "almost consistent"). To overcome this limitation, we proposed a probabilistic approach to querying inconsistent knowledge bases, which provides more informative query answers by associating a degree of consistency with each query answer by associating a probability to each repair, depending on the changes needed to obtain it. Further developments should consider more general frameworks, e.g., with TGDs and logic rules.

Bibliography

[1] Serge Abiteboul and Gösta Grahne. "Update Semantics for Incomplete Databases". In: *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*. Ed. by Alain Pirotte and Yannis Vassiliou. Morgan Kaufmann, 1985, pp. 1–12.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Vol. 8. Addison-Wesley Reading, 1995.

[3] Serge Abiteboul, Paris Kanellakis, and Gösta Grahne. "On the representation and querying of sets of possible worlds". In: *Theoretical computer science* 78.1 (1991), pp. 159–187.

[4] Periklis Andritsos, Ariel Fuxman, and Renée J. Miller. "Clean Answers over Dirty Databases: A Probabilistic Approach". In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. Ed. by Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang. IEEE Computer Society, 2006, p. 30.

[5] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. "Consistent Query Answers in Inconsistent Databases". In: *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*. Ed. by Victor Vianu and Christos H. Papadimitriou. ACM Press, 1999, pp. 68–79.

[6] Marcelo Arenas, Pablo Barceló, Leonid Libkin, and Filip Murlak. *Foundations of data exchange*. Cambridge University Press, 2014.

[7] Paolo Atzeni and Nicola M. Morfuni. "Functional Dependencies and Constraints on Null Values in Database Relations". In: 70.1 (1986), pp. 1–31.

[8] L Bertossi, Andrea Calì, and Mostafa Milani. "Query answering on expressive Datalog+/-ontologies". In: *CEUR Workshop Proceedings*. 1644. CEUR. 2016.

[9] Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[10] Leopoldo E. Bertossi, Loreto Bravo, Enrico Franconi, and Andrei Lopatenko. "The complexity and approximation of fixing numerical attributes in databases under integrity constraints". In: *Inf. Syst.* 33.4-5 (2008), pp. 407–434.

[11] Meghyn Bienvenu, Camille Bourgaux, and François Goasdoué. "Querying Inconsistent Description Logic Knowledge Bases under Preferred Repair Semantics". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. Ed. by Carla E. Brodley and Peter Stone. AAAI Press, 2014, pp. 996–1002.

[12] Meghyn Bienvenu and Magdalena Ortiz. "Ontology-Mediated Query Answering with Data-Tractable Description Logics". In: *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures*. Ed. by Wolfgang Faber and Adrian Paschke. Vol. 9203. Lecture Notes in Computer Science. Springer, 2015, pp. 218–307.

[13] Meghyn Bienvenu and Riccardo Rosati. "Tractable Approximations of Consistent Query Answering for Robust Ontology-based Data Access". In: *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. Ed. by Francesca Rossi. IJCAI/AAAI, 2013, pp. 775–781.

[14] Meghyn Bienvenu, Balder Ten Cate, Carsten Lutz, and Frank Wolter. "Ontology-based data access: A study through disjunctive datalog, CSP, and MMSNP". In: *ACM Transactions on Database Systems (TODS)* 39.4 (2014), pp. 1–44.

[15] Joachim Biskup. "A Formal Approach to Null Values in Database Relations". In: *Advances in Data Base Theory*. 1979, pp. 299–341.

[16] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. "A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. Ed. by Fatma Özcan. ACM, 2005, pp. 143–154.

[17] Marco Calautti, Leonid Libkin, and Andreas Pieris. "An Operational Approach to Consistent Query Answering". In: *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*. Ed. by Jan Van den Bussche and Marcelo Arenas. ACM, 2018, pp. 239–251.

[18] Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "Checking Termination of Logic Programs with Function Symbols through Linear Constraints". In: *Rules on the Web. From Theory to Applications - 8th International Symposium, RuleML 2014, Co-located with the 21st European Conference on Artificial Intelligence, ECAI 2014, Prague, Czech Republic, August 18-20, 2014. Proceedings*. Ed. by Antonis Bikakis, Paul Fodor, and Dumitru Roman. Vol. 8620. Lecture Notes in Computer Science. Springer, 2014, pp. 97–111.

[19] Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "Logic Program Termination Analysis Using Atom Sizes". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Qiang Yang and Michael J. Wooldridge. AAAI Press, 2015, pp. 2833–2839.

[20] Marco Calautti, Nicola Fiorentino, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "Probabilistic Answers over Inconsistent Knowledge Bases". In: *Proceedings of the 28th Italian Symposium on Advanced Database Systems, Villasimius, Sud Sardegna, Italy (virtual due to Covid-19 pandemic), June 21-24, 2020*. Ed. by Maristella Agosti, Maurizio Atzori, Paolo Ciaccia, and Letizia Tanca. Vol. 2646. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 48–55.

[21] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. "A general Datalog-based framework for tractable query answering over ontologies". In: *J. Web Semant.* 14 (2012), pp. 57–83.

[22] Andrea Calì, Georg Gottlob, and Andreas Pieris. "Advanced processing for ontological queries". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 554–565.

[23] Ashok K. Chandra and Philip M. Merlin. "Optimal Implementation of Conjunctive Queries in Relational Data Bases". In: 1977, pp. 77–90.

[24] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (1970), pp. 377–387.

[25] E. F. Codd. “Extending the Database Relational Model to Capture More Meaning”. In: 4.4 (1979), pp. 397–434.

[26] E. F. Codd. “Relational Completeness of Data Base Sublanguages”. In: *Research Report / RJ / IBM / San Jose, California* RJ987 (1972).

[27] Edgar F Codd. “Further normalization of the data base relational model”. In: *Data base systems* 6 (1972), pp. 33–64.

[28] Marco Console, Paolo Guagliardo, and Leonid Libkin. “Approximations and Refinements of Certain Answers via Many-Valued Logics”. In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*. Ed. by Chitta Baral, James P. Delgrande, and Frank Wolter. AAAI Press, 2016, pp. 349–358.

[29] Transaction processing performance council. “TPC benchmark H standard specification, November 2014. Revision 2.17.1.” In: *(TPC-H)*.

[30] Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. “On reconciling data exchange, data integration, and peer data management”. In: *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2007, pp. 133–142.

[31] Cristhian Ariel D Deagustini, Maria Vanina Martinez, Marcelo A Falappa, and Guillermo R Simari. “Datalog+-ontology consolidation”. In: *Journal of Artificial Intelligence Research* 56 (2016), pp. 613–656.

[32] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. “Data exchange: semantics and query answering”. In: *Theoretical Computer Science* 336.1 (2005), pp. 89–124.

[33] Ronald Fagin, Phokion G Kolaitis, Lucian Popa, and Wang-Chiew Tan. “Reverse data exchange: coping with nulls”. In: *ACM Transactions on Database Systems (TODS)* 36.2 (2011), pp. 1–42.

[34] Nicola Fiorentino, Cristian Molinaro, and Irina Trubitsyna. “Optimizing the Computation of Approximate Certain Query Answers over Incomplete Databases”. In: *International Conference on Flexible Query Answering Systems*. Springer. 2019, pp. 48–60.

[35] Nicola Fiorentino, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "ACID: A System for Computing Approximate Certain Query Answers over Incomplete Databases". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1685–1688.

[36] Nicola Fiorentino, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "ACQUA: Approximate Consistent Query Answering Over Inconsistent Knowledge Bases". In: *2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. IEEE. 2019, pp. 107–110.

[37] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. "Querying and repairing inconsistent numerical databases". In: *ACM Trans. Database Syst.* 35.2 (2010), 14:1–14:50.

[38] Enrico Franconi and Sergio Tessaris. "On the Logic of SQL Nulls". In: *Proc. 6th Alberto Mendelzon Int. Workshop on Foundations of Data Management*. 2012, pp. 114–128.

[39] Enrico Franconi and Sergio Tessaris. "The Algebra and the Logic for SQL Nulls". In: *Proc. 20th Italian Symposium on Advanced Database Systems*. 2012, pp. 163–175.

[40] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. "Census Data Repair: a Challenging Application of Disjunctive Logic Programming". In: *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*. Ed. by Robert Nieuwenhuis and Andrei Voronkov. Vol. 2250. Lecture Notes in Computer Science. Springer, 2001, pp. 561–578.

[41] Filippo Furfaro, Sergio Greco, and Cristian Molinaro. "A three-valued semantics for querying and repairing inconsistent databases". In: *Ann. Math. Artif. Intell.* 51.2-4 (2007), pp. 167–193.

[42] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. "The LLUNATIC data-cleaning framework". In: *Proceedings of the VLDB Endowment* 6.9 (2013), pp. 625–636.

[43] Gösta Grahne. *The problem of incomplete information in relational databases*. Vol. 554. Springer Science & Business Media, 1991.

[44] John Grant. "Null Values in a Relational Data Base". In: 6.5 (1977), pp. 156–157.

[45] Sergio Greco and Cristian Molinaro. "Approximate Probabilistic Query Answering over Inconsistent Databases". In: *Conceptual Modeling - ER 2008, 27th International Conference on Conceptual Modeling, Barcelona, Spain, October 20-24, 2008. Proceedings*. Ed. by Qing Li, Stefano Spaccapietra, Eric S. K. Yu, and Antoni Olivé. Vol. 5231. Lecture Notes in Computer Science. Springer, 2008, pp. 311–325.

[46] Sergio Greco and Cristian Molinaro. "Probabilistic query answering over inconsistent databases". In: *Ann. Math. Artif. Intell.* 64.2-3 (2012), pp. 185–207.

[47] Sergio Greco, Cristian Molinaro, and Francesca Spezzano. *Incomplete Data and Data Dependencies in Relational Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[48] Sergio Greco, Cristian Molinaro, and Francesca Spezzano. "Incomplete data and data dependencies in relational databases". In: *Synthesis Lectures on Data Management* 4.5 (2012), pp. 1–123.

[49] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "Approximation algorithms for querying incomplete databases". In: *Information Systems* 86 (2019), pp. 28–45.

[50] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "Computing Approximate Certain Answers over Incomplete Databases". In: *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017*. Ed. by Juan L. Reutter and Divesh Srivastava. Vol. 1912. CEUR Workshop Proceedings. CEUR-WS.org, 2017.

[51] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "Computing Approximate Query Answers over Inconsistent Knowledge Bases". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jérôme Lang. ijcai.org, 2018, pp. 1838–1846.

[52] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. "Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments". In: *Theory Pract. Log. Program.* 13.4-5 (2013), pp. 737–752.

[53] Sergio Greco, Fabian Pijcke, and Jef Wijsen. "Certain query answering in partially consistent databases". In: *Proceedings of the VLDB Endowment* 7.5 (2014), pp. 353–364.

[54] Paolo Guagliardo and Leonid Libkin. "Making SQL Queries Correct on Incomplete Databases: A Feasibility Study". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Tova Milo and Wang-Chiew Tan. ACM, 2016, pp. 211–223.

[55] Sven Hartmann, Markus Kirchberg, and Sebastian Link. "Design by example for SQL table definitions with functional dependencies". In: 21.1 (2012), pp. 121–144.

[56] Sven Hartmann and Sebastian Link. "The Implication Problem of Data Dependencies over SQL Table Definitions: Axiomatic, Algorithmic and Logical Characterizations". In: 37.2 (2012).

[57] Sven Hartmann and Sebastian Link. "When data dependencies over SQL tables meet the logics of paradox and S-3". In: 2010, pp. 317–326.

[58] Jian He, Enzo Veltri, Donatello Santoro, Guoliang Li, Giansalvatore Mecca, Paolo Papotti, and Nan Tang. "Interactive and deterministic data cleaning". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 893–907.

[59] Tomasz Imielinski and Witold Lipski Jr. "Incomplete Information in Relational Databases". In: *J. ACM* 31.4 (1984), pp. 761–791.

[60] Tomasz Imielinski and Witold Lipski. "On Representing Incomplete Information in a Relational Data Base". In: 1981, pp. 388–397.

[61] Tomasz Imieliński and Witold Lipski Jr. "Incomplete information in relational databases". In: *Readings in Artificial Intelligence and Databases*. Elsevier, 1989, pp. 342–360.

[62] Paraschos Koutris and Jef Wijsen. "Consistent query answering for primary keys". In: *ACM SIGMOD Record* 45.1 (2016), pp. 15–22.

[63] Paraschos Koutris and Jef Wijsen. "Consistent query answering for self-join-free conjunctive queries under primary key constraints". In: *ACM Transactions on Database Systems (TODS)* 42.2 (2017), pp. 1–45.

[64] Paraschos Koutris and Jef Wijsen. "The Data Complexity of Consistent Query Answering for Self-Join-Free Conjunctive Queries Under Primary Key Constraints". In: *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Tova Milo and Diego Calvanese. ACM, 2015, pp. 17–29.

[65] Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. "Inconsistency-tolerant query answering in ontology-based data access". In: *J. Web Semant.* 33 (2015), pp. 3–29.

[66] Maurizio Lenzerini. "Data Integration: A Theoretical Perspective". In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. Ed. by Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis. ACM, 2002, pp. 233–246.

[67] Mark Levene and George Loizou. "Database Design for Incomplete Relations". In: 24.1 (1999), pp. 80–125.

[68] Leonid Libkin. "Certain answers as objects and knowledge". In: *Artif. Intell.* 232 (2016), pp. 1–19.

[69] Leonid Libkin. "How to Define Certain Answers". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Qiang Yang and Michael J. Wooldridge. AAAI Press, 2015, pp. 4282–4288.

[70] Leonid Libkin. "Incomplete data: what went wrong, and how to fix it". In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2014, pp. 1–13.

[71] Leonid Libkin. "SQL's Three-Valued Logic and Certain Answers". In: *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*. Ed. by Marcelo Arenas and Martín Ugarte. Vol. 31. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 94–109.

[72] Leonid Libkin. "SQLs three-valued logic and certain answers". In: *ACM Transactions on Database Systems (TODS)* 41.1 (2016), pp. 1–28.

[73] Y. Edmund Lien. "On the Equivalence of Database Models". In: 29.2 (1982), pp. 333–362.

[74] Witold Lipski. "On Semantic Issues Connected with Incomplete Information Databases". In: 4.3 (1979), pp. 262–296.

[75] Witold Lipski Jr. "On relational algebra with marked nulls preliminary version". In: *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*. 1984, pp. 201–203.

[76] Thomas Lukasiewicz, Enrico Malizia, and Cristian Molinaro. "Complexity of Approximate Query Answering under Inconsistency in Datalog+/-". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jérôme Lang. ijcai.org, 2018, pp. 1921–1927.

[77] Thomas Lukasiewicz, Maria Vanina Martinez, Andreas Pieris, and Gerardo I. Simari. "From Classical to Consistent Query Answering under Existential Rules". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. Ed. by Blai Bonet and Sven Koenig. AAAI Press, 2015, pp. 1546–1552.

[78] Cristian Molinaro, Jan Chomicki, and Jerzy Marcinkowski. "Disjunctive databases for representing repairs". In: *Annals of Mathematics and Artificial Intelligence* 57.2 (2009), pp. 103–124.

[79] Marie-Laure Mugnier and Michaël Thomazo. "An introduction to ontology-based query answering with existential rules". In: *Reasoning Web International Summer School*. Springer. 2014, pp. 245–278.

[80] Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. "Formal Semantics of SQL Queries". In: *ACM Trans. Database Syst.* 16.3 (1991), pp. 513–534.

[81] Robert A. Di Paola. "The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas". In: *J. ACM* 16.2 (1969), pp. 324–327.

[82] Riccardo Rosati. "On the Complexity of Dealing with Inconsistency in Description Logic Ontologies". In: *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. Ed. by Toby Walsh. IJCAI/AAAI, 2011, pp. 1057–1062.

[83] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[84] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Vol. 14. Principles of computer science series. Computer Science Press, 1988. ISBN: 0-7167-8069-0.

[85] M.Y. Vardi. "On the Integrity of Databases with Incomplete Information". In: *Symposium on Principles of Database Systems (PODS)*. 1986, pp. 252–266.

[86] Jef Wijsen. "A survey of the data complexity of consistent query answering under key constraints". In: *International Symposium on Foundations of Information and Knowledge Systems*. Springer. 2014, pp. 62–78.

[87] Jef Wijsen. "Database repairing using updates". In: *ACM Trans. Database Syst.* 30.3 (2005), pp. 722–768.

[88] Carlo Zaniolo. "Database Relations with Null Values". In: 28.1 (1984), pp. 142–166.