Coping with Incomplete Data: Recent Advances

Marco Console, Paolo Guagliardo, Leonid Libkin, Etienne Toussaint

Coping with Incomplete Data: Recent Advances

Marco Console
University of Edinburgh

Paolo Guagliardo
University of Edinburgh

Leonid Libkin
University of Edinburgh
ENS-Paris, PSL / INRIA / CNRS

Etienne Toussaint
University of Edinburgh

ABSTRACT

Handling incomplete data in a correct manner is a notoriously hard problem in databases. Theoretical approaches rely on the computationally hard notion of certain answers, while practical solutions rely on ad hoc query evaluation techniques based on three-valued logic. Can we find a middle ground, and produce correct answers efficiently?

The paper surveys results of the last few years motivated by this question. We re-examine the notion of certainty itself, and show that it is much more varied than previously thought. We identify cases when certain answers can be computed efficiently and, short of that, provide deterministic and probabilistic approximation schemes for them. We look at the role of three-valued logic as used in SQL query evaluation, and discuss the correctness of the choice, as well as the necessity of such a logic for producing query answers.

CCS CONCEPTS

• **Theory of computation → Incomplete, inconsistent, and uncertain databases**; **Database theory**; *Logic and databases*; • **Information systems → Incomplete data**; *Data management systems*; **Structured Query Language**;

KEYWORDS

relational databases, incomplete information, certain answers, naive evaluation, approximate query answering, many-valued logics

ACM Reference Format:
Marco Console, Paolo Guagliardo, Leonid Libkin, and Etienne Toussaint. 2020. Coping with Incomplete Data: Recent Advances. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3375395.3387970

1 INTRODUCTION

Handling incomplete data in relational databases is a notoriously hard problem, where a large gap remains between theoretical and practical approaches. Commercial DBMSs, specifically SQL, have been heavily criticized for producing counter-intuitive and/or incorrect answers when handling incomplete data. One often finds statements like *"those SQL features are … fundamentally at odds with the way the world behaves"* [25] or even *"you can never trust the answers you get from a database with nulls"* [24]. Such behavior is often blamed on SQL's three-valued logic (3VL); indeed, programmers tend to think in terms of the familiar two-valued logic, while 3VL underlies the implementation of SQL's null-related features.

On the theory side, correctness is usually associated with the notion of *certain answers*, which are answers we can be sure about no matter how we interpret the incomplete information present in the database. That is, such answers are true in each *possible world* that an incomplete database represents. This approach, first proposed in the late 1970s [35, 53], is now dominant in the literature and it is standard in all applications where incomplete information appears (data integration, data exchange, ontology-based data access, data cleaning, etc.).

The gap between practice and theory is huge. SQL's designers had first and foremost efficient evaluation in mind, but correctness and efficiency do not always get along. Computing certain answers is coNP-hard even for relational calculus/algebra queries [1], i.e., likely to require super-polynomial time in terms of the size of the database. On the other hand, SQL evaluation is very efficient; it is in TC^0 (a small parallel complexity class).

If SQL cannot produce what is generally viewed as *the* correct answers, then what kinds of errors can it generate? To understand this, consider the simple database in Figure 1, adapted from [38]. It shows orders for books, information about customers paying for them, and basic information about customers themselves.

Orders

oid	*title*	*price*
o1	Big Data	30
o2	SQL	35
o3	Logic	50

Payments

cid	*oid*
c1	o1
c2	o2

Customers

cid	*name*
c1	John
c2	Mary

Figure 1: A database of orders, payments, and customers.

Decision support queries against such a database may include finding *unpaid orders*:

```
SELECT oid FROM Orders WHERE oid NOT IN
  ( SELECT oid FROM Payments )
```

or finding customers *who have not placed a paid order*:

```
SELECT C.cid FROM Customers C WHERE NOT EXISTS
  ( SELECT * FROM Orders O, Payments P
      WHERE  C.cid = P.cid AND P.oid = O.oid )
```


PODS'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-7108-7/20/06…$15.00
https://doi.org/10.1145/3375395.3387970

As expected, the first query produces a single answer o3, while the second returns the empty table. But now assume that just a single entry in these tables is replaced by **NULL**: specifically, the value of `oid` in the second tuple of `Payments` changes from o2 to **NULL**. Then the answers to queries change drastically, and in *different ways*: now the unpaid orders query returns the empty table, and the customers without a paid order query returns c2. Because of a single null, we can both miss answers and make up new answers!

Specifically, if certain answers are the correct behavior of query answering over incomplete databases, then SQL evaluation can produce *false negatives*, i.e., miss some of the tuples that belong to certain answers, and can also produce *false positives*, i.e., return tuples that do not belong to certain answers. For example, c2 returned by the second query is a false positive. The unpaid orders query does not generate any false negatives: certain answers are actually empty since we cannot know which order was unpaid. But a simple query

```
SELECT cid FROM Payments
  WHERE oid = 'o2' OR oid <> 'o2'
```

returns only c1 in the database with **NULL** described above, while the certain answer is {c1, c2}.

Can this gap between theory and practice be bridged? In this paper, we survey some recent results whose goal is to do exactly that. Our survey will be structured along three main themes.

Certainty of answers. What does it even mean for an answer to be certain? The first definition was presented informally [35] but then two alternative definitions quickly appeared [43, 53, 54]. Everyone settled on one of them, from [43, 53], which works for the model of relational databases as sets. According to this definition, certain answers are the intersection of query answers in all possible worlds. This definition has many shortcomings, and a more principled approach has been developed over the past few years, starting with [47, 49]. The key idea of the approach is to identify what we *know* about all query answers and then capture this knowledge with an object. The exact shape of certainty depends on how we formulate the knowledge, and which objects can be used to represent it. In particular, this shows that the largely forgotten approach of [54] has many advantages. We survey this line of work in Section 3.

Exact and approximate query answering. We saw that SQL can give all kinds of wrong answers: false positives and false negatives. One question is for what classes of queries the standard query evaluation actually computes certain answers. This was known to work for unions of conjunctive queries but, as shown in [32], the class is actually much larger. We explain this in Section 4.1.

If certain answers cannot be computed precisely, the next best thing is to approximate them. There could be different approaches. One is to produce subsets of certain answers, i.e., eliminate false positives. This idea is not new: it was first explored more than 30 years ago [60, 67]. Those papers assumed the model of databases as logical theories and could not lead to implementations that would handle familiar relational databases with nulls.

The first approximation of certain answers to queries on databases with nulls was proposed in [51], but did not lead to an efficient implementation. A modification of it, on the other hand, was shown to behave well on benchmark queries [37]. It was also more flexible and provided both no-false-positives and no-false-negatives approximations. This line of work is surveyed in Section 4.2.

Another type of approximation is probabilistic. The idea is to compute the probability that a tuple is an answer to the query on a randomly chosen possible world. The notion of choosing a possible world randomly is very close to the notion used in the study of asymptotic properties of logical sentences and 0–1 laws. The approach, proposed in [52], showed that "almost certainly true answers" (those that are correct with probability 1) are much easier to obtain computationally than certain answers. This is described in Section 4.3.

The role of many-valued logics. SQL uses a three-valued logic to evaluate queries in the presence of nulls. Motivated by this, we extend our techniques to many-valued logics and ask two different questions. The first is: under what conditions a many-valued query evaluation is guaranteed to produce an approximation of certain answers? We provide a simple sufficient condition for that. We also show that SQL evaluation fails that condition, and explain the exact culprit in SQL's query evaluation. This line of work, originated in [51], and further developed in [19], is presented in Section 5.1.

The second question we ask goes even deeper: does SQL even need a three-valued logic? If it does, did SQL use the right one? Or could one have used the familiar two-valued Boolean logic all along? The answer to these questions is twofold. We explain that Kleene's three-valued logic, i.e., SQL's way to propagate truth values through connectives, is the right choice to handle incomplete information, if one wants to use the optimization algorithms implemented in DBMSs. However, going to predicate logics, i.e., the actual formalism underlying SQL, three-valued logic is not really needed, as it provides no additional expressive power compared to standard Boolean first-order logic.

2 BASIC CONCEPTS

Incomplete databases

We consider incomplete databases with nulls interpreted as missing information. Below we recall definitions that are standard in the literature [43, 48, 66]. Databases are populated by two types of elements: *constants* and *nulls*, coming from countably infinite sets denoted by Const and Null, respectively. Nulls are denoted by $\perp$, sometimes with subscripts. If nulls can repeat in a database, they are referred to as *marked*, or labeled, nulls; otherwise one speaks of *Codd nulls*, which are the usual way of modeling SQL's nulls. Marked nulls are standard in applications such as data integration, data exchange and OBDA [5, 11, 45], and they are more general than Codd nulls; hence we use them here.

A relational schema is a set of relation names with associated arities. In an incomplete relational instance D, each k-ary relation symbol R from the vocabulary is interpreted as a k-ary relation R^D over $\mathsf{Const} \cup \mathsf{Null}$. In other words, such a relation R^D is a finite subset of $(\mathsf{Const} \cup \mathsf{Null})^k$. Slightly abusing notation (when it does not lead to confusion) we will call it R as well.

The sets of constants and nulls that occur in a database D are denoted by $\mathsf{Const}(D)$ and $\mathsf{Null}(D)$, respectively. The *active domain* of D is $\mathsf{dom}(D) = \mathsf{Const}(D) \cup \mathsf{Null}(D)$. If D has no nulls, we say that it is *complete*.

Valuations and query answering

A *valuation* v on a database D is a map $v : \mathsf{Null}(D) \to \mathsf{Const}$ that assigns constant values to nulls occurring in the database. By $v(D)$ we denote the result of replacing each null $\perp$ with $v(\perp)$ in D. The *semantics* $[\![D]\!]$ of an incomplete database D is the set of all complete databases it can represent, i.e.,

$$[\![D]\!] = \{v(D) \mid v \text{ is a valuation}\}.$$

This is known as the closed-world semantics of incompleteness, or semantics under CWA, or *closed-world assumption* [58]. Semantics of incompleteness can also be defined under OWA, or *open-world assumption* as

$$[\![D]\!]_{\text{OWA}} = \{\text{complete } D' \mid v(D) \subseteq D' \text{ for a valuation } v\}.$$

A k-ary *query* is a map that, with a database D, associates a subset of k-tuples over its elements, i.e., a subset of $\mathsf{dom}(D)^k$. Queries in standard languages such as relational algebra, calculus, Datalog, etc., cannot invent new values, i.e., return constants that are not in the active domain. Such a query is called *generic* if it commutes with permutations of the domain of constants. That is, a query is *generic* if $Q\big(\pi(D)\big) = \pi\big(Q(D)\big)$ whenever $\pi : \mathsf{Const} \to \mathsf{Const}$ is a bijection. The class of generic queries will be denoted by GEN. All classes of queries considered in this survey, which do not mention elements of Const explicitly, are generic. Queries that mention constants are generic in a slightly weaker sense: $Q\big(\pi(D)\big) = \pi\big(Q(D)\big)$ holds for every bijection π such that $\pi(c) = c$ for each constant mentioned in the query. A *Boolean* query is a query of arity zero. There is only one tuple of arity zero, namely the empty tuple (). As usual, we associate *false* with the empty set $\emptyset$, and *true* with the singleton $\{()\}$ consisting only of the empty tuple. For Boolean queries, we can write alternatively $Q(D) = \mathsf{true}$ or $D \models Q$.

Query languages

As our basic query languages we consider relational calculus and its fragments. Relational calculus has exactly the power of *first-order logic*, or FO. The atomic formulae of FO are relational atoms $R(\bar{x})$, equality atoms $x = y$, constant test $\mathsf{const}(x)$ and null test $\mathsf{null}(x)$, where $\mathsf{const}(x)$ is true iff $x \in \mathsf{Const}$ and $\mathsf{null}(x)$ is true iff $x \in \mathsf{Null}$. FO formulae consist of all atoms above and are closed under conjunction $\wedge$, disjunction $\vee$, negation $\neg$, existential quantifiers $\exists$, and universal quantifiers $\forall$. If $\bar{x}$ is the list of free variables of a formula φ, we write $\varphi(\bar{x})$ to indicate this explicitly. We write $|\bar{x}|$ for the length of $\bar{x}$.

Conjunctive queries (CQs, a.k.a. select-project-join queries) are defined as queries expressed in the $\exists, \wedge$-fragment of FO. The class UCQ of *unions of conjunctive queries* is the set of formulae of the form $\varphi_1 \vee \cdots \vee \varphi_m$, where each φ_i is a conjunctive query. In terms of its expressive power, this is the existential-positive fragment of FO, i.e., the $\exists, \vee, \wedge$-fragment.

We shall use *relational algebra*, the procedural language equivalent to FO, that has the operations of selection σ, projection π, Cartesian product $\times$, union $\cup$, and difference $-$. Selection conditions θ are built according to the following grammar:

$$\theta := \mathsf{const}(A) \mid \mathsf{null}(A) \mid A = B \mid A = c \mid A \neq B \mid A \neq c \mid \theta \vee \theta \mid \theta \wedge \theta$$

where A and B are attributes, c is a constant, and $\mathsf{const}(A)$ and $\mathsf{null}(A)$ test whether the value of A is a constant or a null, respectively. As there is no explicit negation $\neg$, negating selection conditions means propagating negations through them, and interchanging $=$ and $\neq$, and const and null. So, for example, $\neg\big(A = B \wedge \mathsf{null}(A)\big)$ denotes the condition $A \neq B \vee \mathsf{const}(A)$. The fragment of relational algebra without the difference operator and inequalities ($\neq$) in the selection conditions is referred to as *positive relational algebra*. Over complete databases, positive relational algebra has the same expressiveness of existential positive formulae (and thus UCQs).

3 WHAT ARE CERTAIN ANSWERS?

An incomplete database can be seen as a compact representation of many possible worlds. When answering queries on an incomplete database, one then wants to find query answers that are "certain", in the sense that they are true in all possible worlds represented by the incomplete database.

We first formalize certain answers at an abstract level, and then look at them more specifically in the context of relational databases.

3.1 Query answering on incomplete databases

We follow the general framework of [49] by considering databases as abstract "objects"; these could be relational databases, or graphs, or XML documents, or other. We will look at the relational setting in Section 3.2; here, to make clear the distinction with concrete relational databases, we denote database objects with lowercase letters x, y, and so on.

Definition 3.1. A *database domain* $\mathbf{D}$ is a triple $(\mathbb{I}, \mathbb{C}, [\![\cdot]\!])$, where $\mathbb{I}$ is a set of database objects, $\mathbb{C} \subseteq \mathbb{I}$ is the set of complete objects (over which queries are defined), and $[\![\cdot]\!]$ is a semantic function from $\mathbb{I}$ to the powerset of $\mathbb{C}$ such that $x \in [\![x]\!]$ for every $x \in \mathbb{C}$.

For convenience of notation, we also use $\mathbb{I}_{\mathbf{D}}$, $\mathbb{C}_{\mathbf{D}}$, $[\![\cdot]\!]_{\mathbf{D}}$ to refer to the set of incomplete objects, the set of complete objects, and the semantic function, respectively, of a database domain $\mathbf{D}$.

The elements of $[\![x]\!]$ are referred to as the *possible worlds* of x. Intuitively, the more possible worlds a database object represents, the more ambiguous it is. To make this intuition formal, with each database domain $\mathbf{D} = (\mathbb{I}, \mathbb{C}, [\![\cdot]\!])$, we associate an information preorder $\preceq_{\mathbf{D}}$ (that is, a reflexive and transitive relation) on $\mathbb{I}$, defined as follows: $x \preceq y$ iff $[\![y]\!] \subseteq [\![x]\!]$. That is, $x \in \mathbb{I}$ is *less informative* than $y \in \mathbb{I}$ if every possible world of y is also a possible world of x.

Given two database domains $\mathbf{S}$ (for source) and $\mathbf{T}$ (for target), a *query* from $\mathbf{S}$ to $\mathbf{T}$ is a mapping Q from $\mathbb{C}_{\mathbf{S}}$ to $\mathbb{C}_{\mathbf{T}}$, i.e., it maps complete databases of a source database domain $\mathbf{S}$ to complete databases of a target database domain $\mathbf{T}$. A natural way to define meaningful query answers on an incomplete database object x is to consider answers that are "true" in all possible worlds of x. To this end, for a set C of complete database objects, we let

$$Q(C) = \big\{\, Q(x) \mid x \in C \,\big\}.$$

Then, to formalize the notion of truth, we assume, for each database domain, the existence of a set $\mathbb{F}$ of formulae expressing *knowledge*

about incomplete objects, and a satisfaction relation $\models$ indicating when a formula φ is true in a database object x, written $x \models \varphi$.

For a set X of database objects we write $X \models \varphi$ if $x \models \varphi$ for each $x \in X$. Similarly, for a set Φ of formulas, we write $x \models \Phi$ if $x \models \varphi$ for each $\varphi \in \Phi$. The *theory* of X is the set $\mathsf{Th}(X) = \{\varphi \mid X \models \varphi\}$, and the *models* of Φ are given by $\mathsf{Mod}(\Phi) = \{x \mid x \models \Phi\}$. Intuitively, $\mathsf{Th}(X)$ contains the knowledge we have about objects of X, and $\mathsf{Mod}(\Phi)$ consists of all objects satisfying the knowledge expressed by Φ. When the database domain is not clear from the context, we will explicitly indicate it as superscript in Th by writing, e.g., $\mathsf{Th}^{\mathbf{D}}$.

For a database domain $\mathbf{D}$, we want the knowledge $\mathbb{F}$ to satisfy some minimal requirements, such as being compatible with the information pre-order $\preceq_{\mathbf{D}}$, and being expressive enough to capture at least the semantics of incompleteness $[\![\cdot]\!]_{\mathbf{D}}$. To this end, we require that

(a) for every $\varphi \in \mathbb{F}$ and every $x, y \in \mathbb{I}_{\mathbf{D}}$, if $x \preceq_{\mathbf{D}} y$ and $x \models \varphi$ then $y \models \varphi$;
(b) for every $x \in \mathbb{I}_{\mathbf{D}}$ there exists a formula $\delta_x \in \mathbb{F}$ equivalent to $\mathsf{Th}(x)$ (i.e., $\mathsf{Mod}(\mathsf{Th}(x)) = \mathsf{Mod}(\delta_x)$), furthermore if $x \not\preceq_{\mathbf{D}} y$, then $y \not\models \delta_x$.

We are finally ready to define what is certainly true when answering queries on incomplete database objects.

Definition 3.2. Let Q be a query from $\mathbf{S}$ to $\mathbf{T}$, and let $x \in \mathbb{I}_{\mathbf{S}}$. The *certain knowledge* of Q on x is the set $\mathsf{Th}^{\mathbf{T}}\big(Q([\![x]\!]_{\mathbf{S}})\big)$.

Thus, the certain knowledge of Q on x is the set of all formulae that are true in every possible world of x. However, in practice, we expect the answer to a query on a database object to be an object itself, ideally one that captures the entire certain knowledge of the query, that is, $o \in \mathbb{I}_{\mathbf{T}}$ such that $\mathsf{Th}^{\mathbf{T}}(o) = \mathsf{Th}^{\mathbf{T}}(Q([\![x]\!]_{\mathbf{S}}))$. Unfortunately, such an object need not exist in general. To overcome this issue, we define the certain answer as the most informative object (w.r.t. $\preceq_{\mathbf{T}}$) whose theory is a subset of the certain knowledge: $\mathsf{Th}^{\mathbf{T}}(o) \subseteq \mathsf{Th}^{\mathbf{T}}(Q([\![x]\!]_{\mathbf{S}}))$. Therefore, the definition of certain answers as an object is an under-approximation of the certain knowledge; the most informative one allowed by the database domain, but an approximation nonetheless.

Definition 3.3. Let Q be a query from $\mathbf{S}$ to $\mathbf{T}$, and let $x \in \mathbb{I}_{\mathbf{S}}$. The *information-based certain answer* to Q on x is

$$\mathsf{cert}_O(Q, x) = \bigwedge Q([\![x]\!]_{\mathbf{S}}), \tag{1}$$

where the greatest lower bound $\bigwedge$ is with respect to $\preceq_{\mathbf{T}}$.

That is, the most informative object that is less informative than every possible answer. This notion satisfies the expected property of answers on incomplete databases: more informative query inputs yield more informative query answers.

Proposition 3.4 (see [49]). *Let Q be a query from $\mathbf{S}$ to $\mathbf{T}$, and let $x, y \in \mathbb{C}_{\mathbf{S}}$ be such that $x \preceq_{\mathbf{S}} y$. If $\mathsf{cert}_O(Q, x)$ and $\mathsf{cert}_O(Q, y)$ exist, then $\mathsf{cert}_O(Q, x) \preceq_{\mathbf{T}} \mathsf{cert}_O(Q, y)$.*

Even though the certain answers as object are an approximation of the certain knowledge, this does not guarantee their existence. Next, we discuss this problem in the context of relational databases.

3.2 Certain answers on relational databases

A *relational* database domain $\mathbf{D}$ is such that $\mathbb{I}_{\mathbf{D}}$ consists of relational databases populated with elements of $\mathsf{Const} \cup \mathsf{Null}$, $\mathbb{C}_{\mathbf{D}}$ is the subset of $\mathbb{I}_{\mathbf{D}}$ of databases without nulls, and $[\![\cdot]\!]_{\mathbf{D}}$ is either the CWA or OWA semantics of incompleteness.

Under CWA, the facts stored in a database are assumed to be the whole truth. When used on the target domain of queries, this would provide a precise semantics for answers; however, in such a case, information-based certain answers may not exist even for very simple queries. The following result is implicit in [49].

Proposition 3.5. *Let $\mathbf{S}$ and $\mathbf{T}$ be relational database domains. If $[\![\cdot]\!]_{\mathbf{T}}$ is the CWA semantics of incompleteness, then there exist a database $D \in \mathbb{I}_{\mathbf{S}}$ and a Boolean conjunctive query Q from $\mathbf{S}$ to $\mathbf{T}$ for which $\mathsf{cert}_O(Q, D)$ does not exist.*

To see this, consider the database $D = \{R(\bot)\}$ and the Boolean query Q returning true iff $R(2)$ belongs to the database. Then, the certain knowledge of Q on D is the set of formulae satisfied by both $\{()\}$ (when $\bot \mapsto 2$) and $\varnothing$ (otherwise); therefore, this knowledge is empty and, under CWA, there exists no database whose theory is the empty set. The reason for this is that all facts in a database are true, but at the same time, under CWA, all other facts not in it are assumed to be false. So, the theory of the empty database is not empty under CWA.

To overcome this difficulty, we consider the OWA semantics of incompleteness (see Section 2), under which the facts stated in a database are true but they are not assumed to be the whole truth. Thus, the more facts an answer contains, the more knowledge it provides. This fits in well with the idea of approximating query answers: then finding additional tuples in the answer makes approximations more informative. Also, with OWA, unlike with CWA, the theory of the empty database is the empty set: $\mathsf{Th}(\varnothing) = \varnothing$. Therefore, as we shall see, the information-based certain answers exist for larger classes of queries.

In what follows, we implicitly consider queries from $\mathbf{S}$ to $\mathbf{T}$, where $\mathbf{S}$ and $\mathbf{T}$ are relational database domains such that $[\![\cdot]\!]_{\mathbf{T}}$ is the OWA semantics of incompleteness. For the source domain $\mathbf{S}$, either CWA or OWA can be used: the former assumes we completely know the real world, while the latter allows for the possibility that we may be missing some facts. When this choice is relevant for the results, we indicate it by saying that a query, or class of queries, is "under CWA" (resp., under "OWA").

Proposition 3.6 (see [4]). *The information-based certain answer (i.e., cert_O) always exists*

(a) for generic queries under CWA;
(b) for unions of conjunctive queries under OWA.

There are first-order queries for which the information-based certain answer does not exist under OWA.

Thus, OWA on the source makes a huge difference in the class of queries for which the information-based certain answers exist. The reason for this is that we can find a first-order query Q and a database D (under OWA) for which the certain knowledge contains infinitely many non-equivalent formulae. Then, for every finite answer database A whose theory is contained in this infinite set,

we can always find a finite A' such that:

$$\mathsf{Th}(A) \subset \mathsf{Th}(A') \subseteq \mathsf{Th}\big(Q([\![D]\!]_{\text{OWA}})\big)$$

and therefore the greatest lower bound does not exist.

Other notions of certain answers exist in the literature. The most common ones are intersection-based certain answers [1, 43, 53, 60], and certain answers with nulls [51, 54]. While these are specific to the relational domain, and therefore less general than information-based certain answers, they have the advantage of existing for larger classes of queries.

Definition 3.7. The *intersection-based certain answer* to a query Q on a database D is the set:

$$\mathsf{cert}_\cap(Q, D) = \bigcap_{D' \in [\![D]\!]_{\mathbf{S}}} Q(D'). \tag{2}$$

Observe that $\mathsf{cert}_\cap(Q, D)$ consists solely of constants. When the target domain of queries allows only for databases without nulls, the intersection-based certain answers are precisely the information-based ones:

Proposition 3.8 (see [4]). *Let* $\mathbf{S}$ *and* $\mathbf{T}$ *be relational database domains such that*

- $\mathbf{S}$ *is under either* OWA *or* CWA,
- $\mathbf{T}$ *is under* OWA, *and* $\mathbb{I}_{\mathbf{T}}$ *consists of databases without nulls.*

Then, for every generic query Q *from* $\mathbf{S}$ *to* $\mathbf{T}$ *and for every database* $D \in \mathbb{I}_{\mathbf{S}}$, *we have that* $\mathsf{cert}_O(Q, D)$ *and* $\mathsf{cert}_\cap(Q, D)$ *exist and coincide.*

One of the problems with intersection-based certain answers is that they only consist of constants and, for this reason, may miss tuples that are in fact certain. To see this, consider the database $D = R(\bot)$ and the query Q that simply returns R. Then $\mathsf{cert}_\cap(Q, D) = \emptyset$, even though we are certain that $\bot$ is in R no matter which missing value it represents. To overcome this shortcoming, nulls in certain answers can be allowed as follows:

Definition 3.9. The *certain answers with nulls* to a query Q on a database D is the set:

$$\mathsf{cert}_\bot(Q, D) = \big\{\, \bar{t} \mid v(\bar{t}) \in Q(D') \text{ for every valuation } v \text{ and for every } D' \in [\![v(D)]\!]_{\mathbf{S}} \big\} \tag{3}$$

When the source relational database semantics is that of CWA, the definition becomes

$$\mathsf{cert}_\bot(Q, D) = \big\{\, \bar{t} \mid v(\bar{t}) \in Q\big(v(D)\big) \text{ for every valuation } v \big\}.$$

In the above example of D containing $R(\bot)$, we have $\mathsf{cert}_\bot(R, D) = \{\bot\}$, i.e., we keep the certain information about $\bot$ being in R. This is the way the definition was originally given in [54]. The formulation of Definition 3.9 applies to other semantics, such as OWA.

While $\mathsf{cert}_\bot(Q, D)$ may contain nulls and constants, these may only come from $\mathsf{dom}(D)$. Thus, certain answers with nulls exist for every generic query Q and for every database D.

Unlike intersection-based certain answers, certain answers with nulls cannot be captured by the information-based ones. The reason for this is that the general framework of Section 3.1 cannot distinguish between the possible worlds of a database, whereas a valuation provides an "explanation" for each of them. Certain answers with nulls exploit this by ensuring that the *same* explanation that accounts for a possible world also accounts for a tuple being an answer on that possible world [4]. They are also conservative over intersection-based certain answers, as the following proposition shows.

Proposition 3.10 (see [51]). *Let* $\mathbf{S}$ *and* $\mathbf{T}$ *be relational database domains under* CWA *and* OWA, *respectively. Let* Q *be an* m*-ary generic query from* $\mathbf{S}$ *to* $\mathbf{T}$, *and let* $D \in \mathbb{I}_{\mathbf{S}}$. *Then, for every valuation* v, *we have that*

$$\begin{aligned} v\big(\mathsf{cert}_\bot(Q, D)\big) &\subseteq Q\big(v(D)\big) \\ \mathsf{cert}_\cap(Q, D) &= \mathsf{cert}_\bot(Q, D) \cap \mathsf{Const}(D)^m. \end{aligned}$$

In light of this, we could conclude that certain answers with nulls are a better notion than information-based certain answers, as they exist for a larger class of queries and preserve explanations in the form of valuations. On the other hand, information-based certain answers, unlike certain answers with nulls, permit output values that were not in the input database; they could therefore prove very useful for investigating certainty for value-inventing queries, such as those involving aggregation or arithmetic operations. As the study of these non-generic queries is still in its infancy, here we focus on certain answers with nulls.

Complexity and size of certain answers. By definition, the information-based certain answer is the most informative database consistent with all query answers. Since answers are interpreted under OWA, more informativeness means more tuples, which of course comes at a cost in space.

Theorem 3.11 (see [4, 6]). *For generic queries, under the* CWA *semantics of input databases, the size of the information-based certain answer is at most doubly exponential in the size of the database. Moreover, under both* OWA *and* CWA *interpretation of input databases, there exist a query* $Q \in \mathsf{UCQ}^{\neq}$ *and a database* D *for which the size of* $\mathsf{cert}_O(Q, D)$ *is exponential in the size of* D.

At the moment we do not know how to close the gap between the single-exponential lower bound and the double exponential upper bound for queries on CWA databases. This depends on some unresolved problems related to families of cores of graphs [41]; see [4] for more details.

On the other hand, as the certain answer with nulls consists only of tuples over the domain of the database, the size of $\mathsf{cert}_\bot(Q, D)$ is at most polynomial in the size of D. However, computing it is intractable in data complexity under CWA, and undecidable (still in data complexity, i.e., for a fixed query) under OWA.

Theorem 3.12 (see [1, 31]). *Under* OWA, *there exist a (fixed) first-order query* Q *for which, given a database* D *and a tuple* $\bar{t}$, *it is undecidable whether* $\bar{t} \in \mathsf{cert}_\bot(Q, D)$ *(resp.,* $\bar{t} \in \mathsf{cert}_O(Q, D)$*).*

Under CWA, *there exist a (fixed) query* $Q \in \mathsf{UCQ}^{\neq}$ *such that deciding, given a database* D *and a tuple* $\bar{t}$, *whether* $\bar{t} \in \mathsf{cert}_\bot(Q, D)$ *(resp.,* $\bar{t} \in \mathsf{cert}_O(Q, D)$ *is* coNP*-complete.*

4 EXACT AND APPROXIMATE COMPUTATION OF CERTAIN ANSWERS

Even for FO queries, computing certain answers is intractable. To find ways around this problem, we will now look at three different scenarios.

First, we look at cases when the standard efficient evaluation of FO queries produces certain answers. In such an evaluation, called *naïve evaluation*, nulls are simply treated as new constants.

Failing that, we look at *rewriting* queries, so that the rewriting Q' of Q returns a subset of the certain answers to Q. It turns out that surprisingly simple modifications of relational algebra queries can achieve this property.

Finally, we pass from absolute to probabilistic guarantees, and show that for a very large class of queries, including all FO queries, naïve evaluation returns answers which are almost certainly true.

4.1 Naïve evaluation

The idea of naïve evaluation is simple: treat nulls as new values, and evaluate the query by using normal evaluation techniques on databases with nulls. For example, if we have a graph with edges $\{(1, \bot_1), (\bot_1, 2)\}$ and we ask whether there exists a path from 1 to 2, say by a Boolean conjunctive query $Q() :- R(1, x_1), R(x_1, 2)$, then evaluating it naïvely amounts to changing $\bot_1$ to a new constant c, and then evaluating Q on the database $\{(1, c), (c, 2)\}$, which results in a positive answer. More precisely, we say that $v : \mathsf{Null}(D) \to \mathrm{Const}$ is a *bijective valuation* if it is a bijection and $v\big(\mathsf{Null}(D)\big)$ is disjoint from $\mathrm{dom}(D)$ and all the constants mentioned in q. Then

$$Q^{\text{naïve}}(D) \;=\; v^{-1}\Big(Q\big(v(D)\big)\Big).$$

It is easy to see that for queries that are generic, i.e., that are invariant under permutations of the domain (see Section 2), this definition does not depend on the choice of a particular valuation v.

The question we address is the following: when will naïve evaluation produce certain answers, specifically *certain answers with nulls*? The early result goes back 35 years:

Theorem 4.1 (see [43]). *For unions of conjunctive queries, null-free tuples in the output of the naïve evaluation coincides with intersection-based certain answers under both* CWA *and* OWA *semantics.*

In other words, if Q is an m-ary query in UCQ, then $Q^{\text{naïve}}(D) \cap \mathrm{Const}(D)^m = \mathrm{cert}_\cap(Q, D)$. This was the state of the art for a long time, until [47] remarked, using Rossman's preservation theorem [62], that the result is optimal for FO queries under OWA semantics:

Proposition 4.2. *If naïve evaluation computes certain answers for an* FO *Boolean query Q under the* OWA *semantics, then Q is equivalent to a union of conjunctive queries.*

Th connection with Rossman's theorem saying that an FO sentence preserved under homomorphisms is equivalent to an existential positive sentence (and thus to a UCQ) suggested that naïve evaluation could be related to homomorphism preservation. Indeed, the semantics of incomplete databases can be formulated in terms of homomorphisms. We have $D' \in [\![D]\!]_{\text{OWA}}$ iff D' is complete and there is a homomorphism $h : D \to D'$ that is the identity on constants, that is, $h(c) = c$ for all $c \in \mathrm{Const}(D)$. One can similarly restate the closed-world semantics: $D' \in [\![D]\!]$ iff D' is complete and there is a homomorphism $h : D \to D'$ that is the identity on constants and $D' = h(D)$. Such homomorphisms are called *strong onto homomorphisms*.

Let us now look at databases (or more generally first-order relational structures) and connect homomorphism preservation and naïve evaluation more precisely. A *homomorphism* from D to D' is a map $h : \mathrm{dom}(D) \to \mathrm{dom}(D')$ such that, for each tuple $\bar{a} \in R^D$, the tuple $h(\bar{a})$ is in $R^{D'}$. A sentence (i.e., a Boolean query) φ is *preserved under a class $\mathcal{H}$ of homomorphisms* if $D \models \varphi$ implies $D' \models \varphi$ whenever there is a homomorphism $h : D \to D'$ from the class $\mathcal{H}$.

With each class $\mathcal{H}$ of homomorphisms we can also associate a semantics $[\![D]\!]_{\mathcal{H}}$ that consists of all complete D' so that there is a homomorphism $h : D \to D'$ from $\mathcal{H}$ that is the identity on $\mathrm{Const}(D)$. When $\mathcal{H}$ consists of all homomorphisms, this gives us $[\![D]\!]_{\text{OWA}}$, while for strong onto homomorphisms we get $[\![D]\!]$.

Theorem 4.3 (see [32]). *If $\mathcal{H}$ is a class of homomorphisms, then naïve evaluation computes certain answers with nulls to a query Q under the $[\![\cdot]\!]_{\mathcal{H}}$ semantics if and only if Q is preserved under homomorphisms from $\mathcal{H}$.*

We now look at the specific case of FO queries, and three classes of homomorphisms which give rise to natural semantics of incompleteness, and for which we have preservation results. These are arbitrary homomorphisms, strong onto, and also *onto*, or surjective homomorphisms $h : D \to D'$ satisfying $h\big(\mathrm{dom}(D)\big) = \mathrm{dom}(D')$. Note that this is not the same as strong onto. For example, with $D = \{R(\bot_1, \bot_2)\}$ and $D' = \{R(1, 2), R(2, 1)\}$, the homomorphism $h(\bot_1) = 1, h(\bot_2) = 2$ is an onto homomorphism as $h(\{\bot_1, \bot_2\}) = \{1, 2\}$, but not strong onto since there is no tuple in D whose image would be $(2, 1)$.

Note that standard results on preservation under homomorphisms [17] are shown for *arbitrary* structures: finite and infinite. They are usually of the following kind: an FO sentence φ is preserved by a class $\mathcal{H}$ of homomorphisms iff φ is equivalent to a sentence ψ from a syntactic fragment $\mathrm{FO}_{\mathcal{H}}$. In database theory we are of course interested in finite structures. Some of these results work in the finite case, but some do not. However, one direction always works: since sentences in $\mathrm{FO}_{\mathcal{H}}$ are preserved under homomorphisms from $\mathcal{H}$, they are preserved under them on all structures, in particular finite ones. Hence, even if the results are not "if and only if" in the finite, they can still help us identify syntactic classes where naïve evaluation works. We now list what is known about homomorphism preservation for different classes of homomorphisms.

Preservation under arbitrary homomorphisms. It is known that an FO formula is preserved under arbitrary homomorphisms iff it equivalent to an existential positive formula, i.e., a formula in the $\exists, \wedge, \vee$-fragment of FO. In other words, such formulae are precisely unions of conjunctive queries. This is true both for arbitrary structures, and for finite structures [62].

Preservation under onto homomorphisms. An FO formula is preserved under onto homomorphisms on arbitrary – finite and infinite – structures iff it equivalent to a positive formula, i.e., a formula in the $\exists, \forall, \wedge, \vee$-fragment of FO, cf. [17]. This fails however in the finite case [61, 64]: there are FO formulae preserved under onto homomorphisms over all finite structures that are not, over finite structures, equivalent to a positive formula.

Preservation under strong onto homomorphisms. We start with a sufficient condition using a class of *positive formulae with universal guards*, denoted $\mathrm{Pos}^{\forall \mathrm{G}}$, first defined in [18]. Formulae in $\mathrm{Pos}^{\forall \mathrm{G}}$ include all atomic formulae, and they are closed under $\wedge, \vee, \exists, \forall$ and the following formation rule: if $\varphi(\bar{x}, \bar{y})$ is a formula in $\mathrm{Pos}^{\forall \mathrm{G}}$,

and $\alpha(\bar{x})$ is an atomic formula, with all variables in $\bar{x}$ distinct, then

$$\psi(\bar{y}) \;=\; \forall \bar{x}\,\big(\alpha(\bar{x}) \to \varphi(\bar{x},\bar{y})\big)$$

is a $\mathrm{Pos}^{\forall \mathrm{G}}$ formula.

Formulae in $\mathrm{Pos}^{\forall \mathrm{G}}$ are preserved under strong onto homomorphisms [18, 32]. Moreover, over arbitrary structures, they are exactly the FO formulae preserved under strong onto homomorphisms; however, like in the case of onto homomorphisms, the result does not hold for finite structures [16].

The class $\mathrm{Pos}^{\forall \mathrm{G}}$ has a natural description in terms of relational algebra: it consists of queries closed under selection, projection, Cartesian product, union, and *division* by a relation in the schema (or equality). Recall that for a relation S over attributes $B_1, \ldots, B_m$ and a relation R over attributes $A_1, \ldots, A_n, B_1, \ldots, B_m$, the division of R by S is a new relation $R \div S$ over attributes $A_1, \ldots, A_n$ that consists of all tuples $\bar{a}$ such that for each tuple $\bar{b} \in S$, the tuple $(\bar{a}, \bar{b})$ is in R. This is very useful for expressing universal queries such as "find employees who participate in all projects".

Theorem 4.4 (see [32, 49]). *Naïve evaluation outputs certain answers with nulls for:*

- *unions of conjunctive queries under the* OWA *semantics, and*
- $\mathrm{Pos}^{\forall \mathrm{G}}$ *queries under the* CWA *semantics.*

Thus, under the usual closed-world semantics, one can go well beyond unions of conjunctive queries and still produce certain answers with naïve evaluation. There is also an analog of this result for positive formulae and a weaker form of CWA semantics proposed in [59].

For full FO, or relational algebra, this is impossible: naïve evaluation still has very low AC^0 complexity, while finding certain answers is coNP-hard. One can easily give a direct example: $\{1\} - \{\bot\}$ results in $\{1\}$ under naïve evaluation, but the certain answers in this case are empty. Thus, next we shall see how to approximate certain answers for arbitrary relational algebra queries.

4.2 Approximations with absolute guarantees

When working with full relational algebra, due to the high complexity of computing certain answers, we must settle for approximations that can be computed efficiently. Although efficient, standard SQL evaluation may produce answers that are not certain, so we need alternative evaluation schemes that have correctness guarantees and tractable complexity.

Definition 4.5. We say that a query evaluation algorithm has *correctness guarantees* for a query Q if for every database D it returns a subset of $\mathrm{cert}_\bot(Q, D)$.

One such scheme was devised in [51]. The main idea behind it is to translate a query Q into a pair $(Q^{\mathbf{t}}, Q^{\mathbf{f}})$ of queries that have correctness guarantees for Q and its complement $\overline{Q}$, respectively. That is, for every database D, the tuples in $Q^{\mathbf{t}}(D)$ are certainly true, and the tuples in $Q^{\mathbf{f}}(D)$ are certainly false:

$$Q^{\mathbf{t}}(D) \;\subseteq\; \mathrm{cert}_\bot(Q, D) \tag{4a}$$

$$Q^{\mathbf{f}}(D) \;\subseteq\; \mathrm{cert}_\bot(\overline{Q}, D) \tag{4b}$$

The translations of [51] are shown in Figure 2(a), where:

- Dom refers to the query computing the active domain, and $\mathrm{ar}(Q)$ denotes the arity of Q. So, $\mathrm{Dom}^{\mathrm{ar}(Q)}$ refers to the Cartesian product $\mathrm{Dom} \times \cdots \times \mathrm{Dom}$ taken $\mathrm{ar}(Q)$ times.
- The operator $\overline{\ltimes}_{\Uparrow}$ used in the rule $R^{\mathbf{f}}$ is defined as a standard anti-semijoin where the join condition is unifiability of tuples: $\bar{r}$ and $\bar{s}$ are *unifiable* if there exists a valuation v of nulls such that $v(\bar{r}) = v(\bar{s})$.
- The translation θ^* of selection conditions θ is obtained by replacing all comparisons of the form $A \neq x$ with
  - $(A \neq x) \wedge \mathrm{const}(A)$, if x is a constant, and
  - $(A \neq x) \wedge \mathrm{const}(A) \wedge \mathrm{const}(x)$, if x is an attribute name.

Theorem 4.6 (see [51]). *The translations of Figure 2(a) have correctness guarantees:* (4a) *and* (4b) *hold for every relational algebra query Q and for every database D.*

Moreover, both translations $Q^{\mathbf{t}}$ and $Q^{\mathbf{f}}$ have AC^0 *data complexity, and $Q^{\mathbf{t}}(D) = Q(D)$ for complete databases.*

Thus, $Q^{\mathbf{t}}$ has correctness guarantees and does not miss any answers on complete databases; moreover both $Q^{\mathbf{t}}$ and $Q^{\mathbf{f}}$ have good theoretical complexity. However, they suffer from a number of problems that hinder their practical implementation. Crucially, they require the computation of active domains and, even worse, their Cartesian products. While expressible in relational algebra, the $Q^{\mathbf{f}}$ translations for selections, products, projections, and even base relations become prohibitively expensive. Indeed, they are already infeasible for very small databases: simple queries start running out of memory on instances with fewer than 10^3 tuples [37]. Although several optimizations (at the price of missing some certain answers) have been suggested in [51], the cases of projection and base relations do not appear to have any reasonable alternatives.

To overcome the practical difficulties posed by the translations in Figure 2(a), [37] proposed an improved approximation scheme that comes with sufficient correctness guarantees and is implementation-friendly. The main idea is to avoid the translation $Q^{\mathbf{f}}$ that under-approximates certain answers to the negation of the query. This is the principal source of complexity and, in the $Q^{\mathbf{t}}$ translation, it is only used in the rule for difference: a tuple $\bar{a}$ is a certain answer to $Q_1 - Q_2$ if

- $\bar{a}$ is a certain answer to Q_1, and
- $\bar{a}$ is a certain answer to the complement of Q_2.

To avoid working with the complex $Q^{\mathbf{f}}$ translation, the approach of [37] uses a different rule: a tuple $\bar{a}$ is a certain answer to $Q_1 - Q_2$ if

- $\bar{a}$ is a certain answer to Q_1, and
- $\bar{a}$ does not match any tuple that could possibly be an answer to Q_2.

Following this intuition, a query Q is then translated into a pair $(Q^+, Q^?)$ of queries where Q^+ has correctness guarantees for Q, and $Q^?$ approximates possible answers to Q. The advantage of this is that the query $Q^?$ is much simpler than $Q^{\mathbf{f}}$. For instance, for a base relation R, it will be just R itself, as opposed to the complex expression involving Dom.

The translations of [37] are shown in Figure 2(b). Note that the rule $(Q_1 - Q_2)^+$ captures the intuition discussed earlier: $Q_1^+ \,\overline{\ltimes}_{\Uparrow}\, Q_2^?$ gives all the tuples $\bar{r}$ in Q_1^+ for which there does not exist a tuple $\bar{s}$ in $Q_2^?$ such that $\bar{r}$ and $\bar{s}$ unify.

$$
\begin{array}{rcl@{\qquad}rcl}
R^{\mathbf{t}} & = & R & R^{\mathbf{f}} & = & \mathsf{Dom}^{\mathsf{ar}(R)} \,\overline{\ltimes}_{\Uparrow}\, R \\
(Q_1 \cup Q_2)^{\mathbf{t}} & = & Q_1^{\mathbf{t}} \cup Q_2^{\mathbf{t}} & (Q_1 \cup Q_2)^{\mathbf{f}} & = & Q_1^{\mathbf{f}} \cap Q_2^{\mathbf{f}} \\
(Q_1 - Q_2)^{\mathbf{t}} & = & Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{f}} & (Q_1 - Q_2)^{\mathbf{f}} & = & Q_1^{\mathbf{f}} \cup Q_2^{\mathbf{t}} \\
\big(\sigma_\theta(Q)\big)^{\mathbf{t}} & = & \sigma_{\theta^*}(Q^{\mathbf{t}}) & (\sigma_\theta(Q))^{\mathbf{f}} & = & Q^{\mathbf{f}} \cup \sigma_{(\neg\theta)^*}\big(\mathsf{Dom}^{\mathsf{ar}(Q)}\big) \\
(Q_1 \times Q_2)^{\mathbf{t}} & = & Q_1^{\mathbf{t}} \times Q_2^{\mathbf{t}} & (Q_1 \times Q_2)^{\mathbf{f}} & = & Q_1^{\mathbf{f}} \times \mathsf{Dom}^{\mathsf{ar}(Q_2)} \cup \mathsf{Dom}^{\mathsf{ar}(Q_1)} \times Q_2^{\mathbf{f}} \\
\big(\pi_\alpha(Q)\big)^{\mathbf{t}} & = & \pi_\alpha(Q^{\mathbf{t}}) & \big(\pi_\alpha(Q)\big)^{\mathbf{f}} & = & \pi_\alpha(Q^{\mathbf{f}}) - \pi_\alpha\big(\mathsf{Dom}^{\mathsf{ar}(Q)} - Q^{\mathbf{f}}\big)
\end{array}
$$

(a) Translations $Q \mapsto (Q^{\mathbf{t}}, Q^{\mathbf{f}})$ in the approximation scheme of [51].

$$
\begin{array}{rcl@{\qquad}rcl}
R^{+} & = & R & R^{?} & = & R \\
(Q_1 \cup Q_2)^{+} & = & Q_1^{+} \cup Q_2^{+} & (Q_1 \cup Q_2)^{?} & = & Q_1^{?} \cup Q_2^{?} \\
(Q_1 - Q_2)^{+} & = & Q_1^{+} \,\overline{\ltimes}_{\Uparrow}\, Q_2^{?} & (Q_1 - Q_2)^{?} & = & Q_1^{?} - Q_2^{+} \\
\big(\sigma_\theta(Q)\big)^{+} & = & \sigma_{\theta^*}(Q^{+}) & \big(\sigma_\theta(Q)\big)^{?} & = & \sigma_{\neg(\neg\theta)^*}\big(Q^{?}\big) \\
(Q_1 \times Q_2)^{+} & = & Q_1^{+} \times Q_2^{+} & (Q_1 \times Q_2)^{?} & = & Q_1^{?} \times Q_2^{?} \\
\big(\pi_\alpha(Q)\big)^{+} & = & \pi_\alpha(Q^{+}) & \big(\pi_\alpha(Q)\big)^{?} & = & \pi_\alpha\big(Q^{?}\big)
\end{array}
$$

(b) Translations $Q^{+}, Q^{?}$ in the approximation scheme of [37].

Figure 2: Approximation schemes with correctness guarantees for certain answers to relational algebra queries.

Theorem 4.7 (see [37]). *For the translations in Figure 2(b), the queries Q^{+} and $Q^{?}$ are such that $Q^{+}(D) \subseteq \mathsf{cert}_{\perp}(Q, D)$ and*

$$v\big(Q^{+}(D)\big) \;\subseteq\; Q\big(v(D)\big) \;\subseteq\; v\big(Q^{?}(D)\big) \tag{5}$$

for every database D and for every valuation v.

The theoretical complexity bounds for queries Q^{+} and $Q^{\mathbf{t}}$ are the same: both have the low AC^0 data complexity. However, the real world performance of Q^{+} is significantly better, as it completely avoids large Cartesian products.

The good behavior of the translations in Figure 2(b) was confirmed in [37] by proof-of-concept experiments on the TPC Benchmark H [65]. The results of that evaluation showed that the performance overhead of the rewritten queries is limited to a slowdown of 1-4% w.r.t. the original SQL queries. However, there are also cases where performance becomes an issue, even though this is mainly due to the poor way in which the query optimizers of commercial database systems handle disjunctions.

Bag semantics. As prescribed by the SQL Standard, relational database management systems use bag semantics in query evaluation. In a data model based on bags, the same tuple can occur more than once in a relation. The *multiplicity* (i.e., number of occurrences) of a tuple $\bar{a}$ in a relation R is denoted by $\#(\bar{a}, R)$. Relational algebra operations under bag semantics are interpreted in a way that is consistent with SQL evaluation: for example, union adds up multiplicities, while difference subtracts them up to zero (see [22] for further details).

In this context, instead of saying that a tuple is certainly in the answer, we have more detailed information: namely, the range of multiplicities of the tuple in query answers. This is captured by the following definitions, that extend the notion of certain answers with nulls:

$$\Box_Q(D, \bar{a}) \;=\; \min\big\{\#\big(v(\bar{a}), Q(v(D))\big) \mid v \text{ is a valuation}\big\} \tag{6a}$$

$$\Diamond_Q(D, \bar{a}) \;=\; \max\big\{\#\big(v(\bar{a}), Q(v(D))\big) \mid v \text{ is a valuation}\big\} \tag{6b}$$

When a query is evaluated under set semantics, $\Box_Q(D, \bar{a}) = 1$ means that $\bar{a} \in \mathsf{cert}_{\perp}(Q, D)$.

For full relational algebra under bag semantics, the complexity of the bounds (6a) and (6b) mimics analogous results for set semantics. Thus, also in this case, we need to resort to tractable approximation schemes with correctness guarantees. In this respect, (5) suggests a natural extension of the correctness criterion for the translation scheme $(Q^{+}, Q^{?})$ of [37], as follows:

Theorem 4.8 (see [20]). *When queries are interpreted under bag semantics, the translation $Q \mapsto (Q^{+}, Q^{?})$ in Figure 2(b) satisfies*

$$\#\big(\bar{a}, Q^{+}(D)\big) \;\le\; \Box_Q(\bar{a}, D) \;\le\; \#\big(\bar{a}, Q^{?}(D)\big) \tag{7}$$

for every database D and every tuple $\bar{a}$.

On the other hand, the translation of Figure 2(a) loses its good theoretical complexity bounds and becomes intractable under bag semantics. A simple analysis of the definition of queries $Q^{\mathbf{t}}, Q^{\mathbf{f}}$ in Figure 2(a) shows that, for every tuple $\bar{a}$,

$$\#\big(\bar{a}, Q^{\mathbf{t}}(D)\big) \;\le\; \Box_Q(D, \bar{a}) \tag{8a}$$

$$\#\big(\bar{a}, Q^{\mathbf{f}}(D)\big) \;\le\; \big(1 + \Diamond_Q(D, \bar{a})\big) \mod 2 \tag{8b}$$

This suggests a natural extension of the translation scheme $(Q^{\mathbf{t}}, Q^{\mathbf{f}})$ to bags: we omit modulo 2 in (8b), since it is only needed to force multiplicities to be either 0 or 1. But this is problematic, as $\Diamond_Q(D, \bar{a})$ is intractable already for base relations [20]. Thus, when we use bag semantics, implementing this approximation scheme in a real-life RDBMS (which is bag-based) is infeasible not only practically but also theoretically.

The approximation scheme in Figure 2(b) was tested in a commercial DBMS under bag semantics and compared against approaches that do not have correctness guarantees, that is, may return non-certain answers. That study [27] found that, w.r.t. the ground truth, the Q^{+} translation had obviously perfect precision (100%), but recall degraded quickly with the increase in the amount of incompleteness present in the database.

Approximation schemes based on conditional tables. Greco et al. [36] proposed a number of approximation algorithms with correctness guarantees that make use of *conditional tables*, or *c-tables* for short; cf. [43]. In such tables, each tuple $\bar{t}$ is associated with a condition φ that indicates when $\bar{t}$ holds; the pair $\langle \bar{t}, \varphi \rangle$ is referred to as a c-tuple. In this context, relational algebra operations are evaluated by taking into account the conditions associated with c-tuples. For example, for every two input c-tuples $\langle \bar{r}, \varphi_1 \rangle$ and $\langle \bar{s}, \varphi_2 \rangle$, Cartesian product produces the output c-tuple $\langle \bar{r}\bar{s},\ \varphi_1 \wedge \varphi_2 \rangle$, where juxtaposition of tuples denotes concatenation. We refer the reader to [43] for further details on the conditional evaluation of queries on c-tables.

The main idea of [36] consists in converting a database D into a conditional database D', where all conditions are true, to be used as the starting point for the conditional evaluation of queries. At each step of the query evaluation, the conditions associated with c-tuples can be manipulated and grounded, that is, reduced to either true (**t**), false (**f**) or unknown (**u**). Then, different evaluation strategies are obtained depending on how and when c-tuple conditions are handled. Given one such algorithm Eval, we let

$$\mathsf{Eval}_{\mathsf{t}}(Q, D) = \left\{ \bar{t} \mid \langle \bar{t}, \mathbf{t} \rangle \in \mathsf{Eval}(Q, D) \right\} \tag{9a}$$

$$\mathsf{Eval}_{\mathsf{p}}(Q, D) = \left\{ \bar{t} \mid \langle \bar{t}, \tau \rangle \in \mathsf{Eval}(Q, D),\ \tau \in \{\mathbf{t}, \mathbf{u}\} \right\} \tag{9b}$$

Then, Eval has correctness guarantees for a query Q if $\mathsf{Eval}_{\mathsf{t}}(Q, D) \subseteq \mathsf{cert}_{\perp}(Q, D)$ for every database D.

The four approximation algorithms proposed in [36] are informally described as follows:

(1) **Eager** $\mathsf{Eval}^{\mathsf{e}}$: conditions are grounded immediately after each relational algebra operator is applied.
(2) **Semi-eager** $\mathsf{Eval}^{\mathsf{s}}$: similar to the eager strategy, but in addition it also propagates equalities; e.g., the c-tuple $\langle \perp_2, \perp_1 = c \wedge \perp_1 = \perp_2 \rangle$ would give $\langle c, \mathbf{u} \rangle$ rather than the less informative $\langle \perp_2, \mathbf{u} \rangle$.
(3) **Lazy** Eval^{ℓ}: equality propagation and grounding are only performed on the result of each difference operator, but postponed for all other operators.
(4) **Aware** $\mathsf{Eval}^{\mathsf{a}}$: equality propagation and grounding are postponed until the very end of query evaluation, and performed on a minimal rewriting of the conditions.

Theorem 4.9 (see [36]). *For each $\star \in \{e, s, \ell, a\}$, the approximation algorithm $\mathsf{Eval}^{\star}$ has correctness guarantees for all relational algebra queries, and $\mathsf{Eval}^{\star}(Q, D)$ can be computed in polynomial time in the size of D, for every query Q and every database D. Moreover,*

$$Q^{+}(D) = \mathsf{Eval}^{\mathsf{e}}_{\mathsf{t}}(Q, D) \qquad \text{and} \qquad Q^{?}(D) = \mathsf{Eval}^{\mathsf{e}}_{\mathsf{p}}(Q, D)\ .$$

All of the above approximation algorithms were implemented in a proof of concept system [28]. However, the implementation is not integrated into a real DBMS, which makes it hard to compare the overhead of the algorithms w.r.t. the baseline performance of the original SQL queries, as done in [37]. Another obstacle is the use of conditional tables, which seem to slowdown performance [27].

4.3 Approximations with probabilistic guarantees

Let us look again at the simple example of computing the difference $R - S$ where $R = \{1\}$ and $S = \{\perp\}$. While naïve evaluation gives us $\{1\}$, certain answers are empty. However, they are empty only because they have to account for the situation when $\perp$ is interpreted as 1. In all other cases, the answer produced by the naïve evaluation is actually correct; if $\perp$ can be interpreted as an arbitrary element of Const, then the chance of it being 1 is small. So it seems that naïve evaluation produces answers, in this case, that are very likely to be true.

This is no accident; in fact, if we pick an interpretation of nulls uniformly at random and look at answers that are true with probability 1, they are precisely those that are returned by naïve evaluation. To make this intuition precise though, we must say what it means to pick a valuation at random. For a database D, we denote by $\mathsf{V}(D)$ the set of all valuations on D. This set is infinite, and no uniform distribution can be defined on it. To pick v uniformly at random from $\mathsf{V}(D)$, we use the approach from the study of 0–1 laws and asymptotic behavior of logical properties, where one tries to define how likely a randomly chosen structure is to satisfy a given property [26, 46, 63].

Given a query Q, a database D, and a tuple $\bar{a}$ over $\mathsf{dom}(D)$, define the *support* of $\bar{a}$ being an answer to Q on D as the set of all valuations that witness it:

$$\mathsf{Supp}(Q, D, \bar{a}) = \left\{ v \in \mathsf{V}(D) \mid v(\bar{a}) \in Q\big(v(D)\big) \right\}.$$

Supports thus measure how closely a tuple is to certainty. A tuple $\bar{a}$ is in $\mathsf{cert}(Q, D)$ iff $\mathsf{Supp}(Q, D, \bar{a}) = \mathsf{V}(D)$, i.e., the support includes all valuations. We then want to see how likely a randomly chosen valuation is to be in $\mathsf{Supp}(Q, D, \bar{a})$.

For this, assume that the set of Const constants is enumerated as $\{c_1, c_2, \ldots\}$. Define $\mathsf{V}^k(D)$ as the set of valuations whose range is contained among the first k elements of this enumeration, i.e., $\mathsf{V}^k(D) = \left\{ v \in \mathsf{V}(D) \mid \mathsf{range}(v) \subseteq \{c_1, \ldots, c_k\} \right\}$. Let $\mathsf{Supp}^k(Q, D, \bar{a})$ be the restriction of $\mathsf{Supp}(Q, D, \bar{a})$ to $\mathsf{V}^k(D)$, i.e., $\mathsf{Supp}(Q, D, \bar{a}) \cap \mathsf{V}^k(D)$. Then we define

$$\mu^k(Q, D, \bar{a}) = \frac{|\mathsf{Supp}^k(Q, D, \bar{a})|}{|\mathsf{V}^k(D)|}$$

as the proportion of valuations from $\mathsf{V}^k(D)$ that belong to $\mathsf{Supp}(Q, D, \bar{a})$. That is, $\mu^k(Q, D, \bar{a})$ is the probability that a valuation picked uniformly at random from $\mathsf{V}^k(D)$ witnesses that $\bar{a}$ is an answer to Q.

Finally, to eliminate the dependence on k, we look at the asymptotic behavior of this sequence:

$$\mu(Q, D, \bar{a}) = \lim_{k \to \infty} \mu^k(Q, D, \bar{a})\ .$$

This definition assumes some particular enumeration of the set Const. However, it is easy to see that the limit value $\mu(Q, D, \bar{a})$ is independent of a particular enumeration for generic queries: once the set $\{c_1, \ldots, c_k\}$ contains all constants in Q and $\mathsf{Const}(D)$, the value $\mu^k(Q, D, \bar{a})$ does not depend on the remaining elements of this set, just their number. We then call a tuple $\bar{a}$ an *almost certainly true* answer to Q on D if $\mu(Q, D, \bar{a}) = 1$.

Theorem 4.10 (see [52]). *A tuple $\bar{a}$ is an almost certainly true answer to a generic query Q on D if and only if $\bar{a} \in Q^{\text{naïve}}(D)$.*

Furthermore, if $\bar{a} \notin Q^{\text{naïve}}(D)$, then it follows that $\mu(Q, D, \bar{a}) = 0$. In other words, we have a version of a 0–1 law: every tuple is either an almost certainly true or an almost certainly false answer. Note that finding almost certainly true answers is much simpler than finding certain answers: for example, for FO, the complexity is AC^0 instead of coNP. An analog of Theorem 4.10 can be shown when instead of counting valuations v witnessing $v(\bar{a}) \in Q\big(v(D)\big)$ we count isomorphism types on $\mathsf{V}(D)$ witnessing the condition, i.e., the set of databases $\{v(D) \mid v \in \mathsf{V}(D)\}$ such that $v(\bar{a}) \in Q\big(v(D)\big)$. As distinct valuations may define the same isomorphism type, the ratio of databases in $\{v(D) \mid v \in \mathsf{V}^k(D)\}$ such that $v(\bar{a}) \in Q\big(v(D)\big)$ needs to be the same as the ratio of valuations $\mathsf{V}^k(D)$ in $\mathsf{Supp}^k(Q, D, \bar{a})$, for any given k. The asymptotic behavior of these two sequences, however, is the same.

Let us now change the example a bit and assume that we have a relation $T = \{1, 2\}$ and we look for answers to $T - S$ under the inclusion constraint $S \subseteq T$. In this case, $\perp$ can only take values 1 or 2, and thus the answer to Q is $\{1\}$ with probability 1/2 and empty with probability 1/2. Can we use the framework to capture this?

It turns out that the framework is easily adaptable. In real life databases must satisfy integrity constraints, most commonly keys and foreign keys, which are special cases of functional dependencies and inclusion constraints (which in turn are special cases of equality- and tuple-generating dependencies). A set of constraints Σ can be viewed as a Boolean query, returning *true* if the constraints are satisfied and *false* if they are not. Most such constraints, at least those listed above, are generic Boolean queries.

We want to find the conditional probability $\mu(Q|\Sigma, D, \bar{a})$ of $\bar{a}$ being an answer to Q, given that Σ holds. In other words, choose a random assignment of constants to nulls; what is the probability that Q is true under the assumption that Σ is true? As above, we first define it for valuations with the range $\{c_1, \ldots, c_k\}$ and then analyze its asymptotic behavior. That is, define the probability that a randomly chosen valuation v with $\mathrm{range}(v) \subseteq \{c_1, \ldots, c_k\}$ such that $v(D) \models \Sigma$ also satisfies $v(\bar{a}) \in Q\big(v(D)\big)$, i.e.,

$$\mu^k(Q|\Sigma, D, \bar{a}) = \frac{|\mathrm{Supp}^k(\Sigma \wedge Q, D, \bar{a})|}{|\mathrm{Supp}^k(\Sigma, D)|}$$

and then

$$\mu(Q|\Sigma, D, \bar{a}) = \lim_{k\to\infty} \mu^k(Q|\Sigma, D, \bar{a})$$

if such a limit exists. If the denominator $|\mathrm{Supp}^k(\Sigma, D)|$ is zero, i.e., $\mathrm{Supp}^k(\Sigma, D) = \varnothing$, adopt the convention that $\mu^k(Q|\Sigma, D, \bar{a}) = 0$.

Again, it is easy to show that for generic Q and Σ the value $\mu(Q|\Sigma, D, \bar{a})$ does not depend on a particular enumeration of Const. Our earlier example showed that $\mu(Q|\Sigma, D, \bar{a})$ could be a number between 0 and 1, namely 1/2. In general, if the 0–1 law fails, the next best thing we can hope for is convergence, i.e., the existence of the limit.

Theorem 4.11 (see [52]). *If both Σ and Q are generic, then for every database D and for every tuple $\bar{a}$ over the domain, $\mu(Q|\Sigma, D, \bar{a})$ exists and is a rational number in $[0, 1]$.*

Moreover, every number in $\mathbb{Q} \cap [0, 1]$ appears as $\mu(Q|\Sigma, D, \bar{a})$ for a conjunctive query Q and an inclusion constraint Σ.

In some cases, $\mu(Q|\Sigma, D, \bar{a})$ can only take values 0 or 1. For example, this is so if Σ contains only functional dependencies. In fact in this case $\mu(Q|\Sigma, D, \bar{a}) = \mu(Q, D_\Sigma, \bar{a})$, where D_Σ is the result of chasing D with Σ.

To describe the complexity of computing $\mu(Q|\Sigma, D, \bar{a})$ note that, as a rational number, it is represented by a pair (p, r) with $p, r \in \mathbb{N}$. Hence computing it is a problem in a *function* class (rather than a complexity class capturing decision problems), and amounts to computing two numbers. The exact complexity happens to be $\mathrm{FP}^{\#\mathrm{P}}$, which is the class of functions computable in polynomial time with an access to a #P oracle. Furthermore, there are cases when $\mu(Q|\Sigma, D) = p/r$ such that r can be computed in polynomial time while computing p is #P-hard.

$\wedge$	**t**	**f**	**u**
t	**t**	**f**	**u**
f	**f**	**f**	**f**
u	**u**	**f**	**u**

$\vee$	**t**	**f**	**u**
t	**t**	**t**	**t**
f	**t**	**f**	**u**
u	**t**	**u**	**u**

	$\neg$
t	**f**
f	**t**
u	**u**

Figure 3: Truth tables of Kleene's three-valued logic.

5 MANY-VALUED LOGICS FOR INCOMPLETE INFORMATION

The high complexity of certain answers is unsuitable for real-world query evaluation. The approach most broadly taken, in particular by SQL-based relational DBMSs, is to introduce small and easily implementable modifications to how queries are evaluated, so as to handle incomplete information expressed via null values. SQL's approach, in particular, introduces an additional truth value *Unknown* (**u**) to provide extra information about our knowledge of the results produced by operations involving nulls. The truth value **u** is assigned to comparisons involving nulls, such as $1 =$ `NULL`, and it is then propagated through the Boolean connectives $\wedge$, $\vee$, $\neg$ using the truth tables in Figure 3.

This particular propositional logic, known as *Kleene's logic* [13], underlies SQL implementations [25]. More generally, a *propositional many-valued logic* is a pair $(\mathbf{T}, \Omega)$, where $\mathbf{T}$ is a set of *truth values*, and Ω is a set of functions $\omega : \mathbf{T}^n \to \mathbf{T}$ called *connectives*. The most common of those are $\wedge$, $\vee$, $\neg$, but to model SQL properly we need to add one more, as we shall see later. In the familiar two-valued Boolean logic, denoted by $\mathbb{L}_{2v}$, the set $\mathbf{T}$ contains just **t** (true) and **f** (false), and the truth tables are the standard ones (in fact, the restriction of those in Figure 3 to $\{\mathbf{t}, \mathbf{f}\}$). In Kleene's logic, denoted by $\mathbb{L}_{3v}$, the set $\mathbf{T}$ is $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$.

Database query languages are based on predicate logic, which means that we need to define many-valued first-order logics. Normally, in FO, we have the notion of $D \models \varphi(\bar{a})$, i.e., a formula $\varphi(\bar{x})$ is true in D if its free variables are interpreted as $\bar{a}$. That means $\varphi(\bar{a})$ is assigned the truth value **t**; alternatively, when $\varphi(\bar{a})$ does not hold, it is assigned truth value **f**. In a many-valued logic, $\varphi(\bar{a})$ can be assigned any truth value from $\mathbf{T}$.

For a propositional logic $\mathbb{L}$, a *first-order many-valued logic* is a pair $(\mathrm{FO}(\mathbb{L}), (\!|\cdot|\!))$ where $\mathrm{FO}(\mathbb{L})$ is the language of first-order formulae built over the connectives of $\mathbb{L}$, and $(\!|\cdot|\!)$ is a semantics for $\mathrm{FO}(\mathbb{L})$ formulas. That is, for each formula $\varphi(\bar{x})$, database D, and assignment $\bar{a}$ of elements of $\mathrm{dom}(D)$ to the free variables $\bar{x}$ of φ, the semantics $(\!|\varphi|\!)_{D,\bar{a}}$ is a value in $\mathbf{T}$. We assume that such semantics follows the syntax of the connectives, that is,

$$(\!|\omega(\varphi_1, \ldots, \varphi_n)|\!)_{D,\bar{a}} = \omega\big((\!|\varphi_1|\!)_{D,\bar{a}}, \ldots, (\!|\varphi_n|\!)_{D,\bar{a}}\big) \tag{10}$$

for every n-ary connective ω of $\mathbb{L}$, and

$$(\!|\exists x\, \varphi(\bar{x})|\!)_{D,\bar{a}} = \bigvee_{a \in \mathrm{dom}(D)} (\!|\varphi|\!)_{D,(\bar{a},a)} \tag{11}$$

and likewise for $\forall x\, \varphi$ with $\bigwedge$ used in the place of $\bigvee$.

We now discuss recent developments on the use of many-valued logics to deal with databases with incomplete information. In Section 5.1, we examine the work of [51] and [19] about refining query answers using many-valued logics. In Section 5.2, we consider

the many-valued logic underlying SQL, and present results from [21] showing that it does not add expressiveness beyond the usual Boolean logic, despite the generally held belief that it does.

5.1 Correctness of many-valued evaluation procedures

When does the semantics of first-order many-valued logic produce correct answers? To answer this question, we assume that there is a notion of correct answers, given by a function $\mathsf{Answ}(\varphi)_{D,\bar{a}}$ that produces a truth value $\tau \in \mathbf{T}$. For example, one may want to impose $\mathsf{Answ}(\varphi)_{D,\bar{a}} = \mathbf{t}$ if $\bar{a} \in \mathsf{cert}_\perp(\varphi, D)$ stating that correct true answers are certain answers with nulls.

To state our assumptions on $\mathsf{Answ}(\cdot)$ and explain when the semantics $(\!|\cdot|\!)$ gives rise to an evaluation procedure for such answers, we need a basic concept related to many-valued logics, namely a *knowledge order* $\preceq_{\mathbb{L}}$ on truth values of the propositional logic $\mathbb{L}$; see [10, 33]. While truth values can be ordered by the degree of truth they provide (in this case, **t** is more true than **f**), in the many-valued context they can also be ordered by the degree of knowledge they provide. In $\mathbb{L}_{3\mathrm{v}}$, the natural such ordering is $\mathbf{u} \preceq_{\mathbb{L}} \mathbf{t}$ and $\mathbf{u} \preceq_{\mathbb{L}} \mathbf{f}$. Note that **f** and **t** are incomparable: each of them provides complete knowledge, while **u** has less knowledge than either. Such knowledge orderings exist for many other many-valued logics, and they have been an object of extensive study, especially in connection with paraconsistent logics (see, e.g., [7, 8]). One common assumption is the existence of a least element for $\preceq_{\mathbb{L}}$, denoted by τ_0; for $\mathbb{L}_{3\mathrm{v}}$, this is **u**. Intuitively, τ_0 represents the no-information value, i.e., the fall-back position when we cannot assign any value carrying real information.

The semantics $(\!|\cdot|\!)$ has *correctness guarantees* for query answers $\mathsf{Answ}(\cdot)$ if whenever $(\!|\varphi|\!)_{D,\bar{a}} = \tau$ and $\tau \neq \tau_0$, then $\mathsf{Answ}(\varphi)_{D,\bar{a}} = \tau$. Intuitively, for every meaningful truth value τ, the semantics of formulae has to agree with the correct answers. In cases when it cannot capture the correct answers precisely, it has to resort to the no-information truth value τ_0.

To ensure correctness guarantees we need two conditions, for each n-ary propositional connective ω of $\mathbb{L}$.

(1) The notion of correct answers must respect the propositional logic $\mathbb{L}$. That is, if $\mathsf{Answ}(\varphi_i)_{D,\bar{a}} = \tau_i$ for $1 \le i \le n$, then $\mathsf{Answ}\big(\omega(\varphi_1, \ldots, \varphi_n)\big)_{D,\bar{a}} = \omega(\tau_1, \ldots, \tau_n)$ as long as all the truth values are not the bottom truth value τ_0.
(2) The logic $\mathbb{L}$ must respect the knowledge order. That is, if we have truth values such that $\tau_1 \preceq_{\mathbb{L}} \tau_1', \ldots, \tau_n \preceq_{\mathbb{L}} \tau_n'$, then $\omega(\tau_1, \ldots, \tau_n) \preceq_{\mathbb{L}} \omega(\tau_1', \ldots, \tau_n')$.

Theorem 5.1 (see [19]). *Assume that $\mathbb{L}$ respects the knowledge order, and that the notion of correct answers respects $\mathbb{L}$, as defined above. If $(\!|\cdot|\!)$ has correctness guarantees for atomic formulae, then it has correctness guarantees for all* FO *formulae.*

The results in [19] are actually more general and establish the lifting criterion for a wider range of semantics. They also show how to define query answers for arbitrary queries, not only FO, in a way that more informative answers are obtained on more informative inputs.

Data complexity too is lifted from atomic to arbitrary formulae: if $(\!|\cdot|\!)$ has NC^1 data complexity for atomic formulae, then it has NC^1 data complexity for all $\mathsf{FO}(\mathbb{L})$. Moreover, NC^1 can be replaced by AC^0 data complexity if both $\wedge$ and $\vee$ are idempotent in $\mathbb{L}$.

Correctness for Kleene's logic. Now, for SQL's many-valued logic $\mathbb{L}_{3\mathrm{v}}$, we give a concrete example of a semantics for $\mathsf{FO}(\mathbb{L}_{3\mathrm{v}})$ with correctness guarantees. As the notion of correctness we take certain answers with nulls:

$$\mathsf{Answ}(\varphi)_{D,\bar{a}} = \begin{cases} \mathbf{t} & \bar{a} \in \mathsf{cert}(\varphi, D) \\ \mathbf{f} & \bar{a} \in \mathsf{cert}(\neg\varphi, D) \\ \mathbf{u} & \text{otherwise} \end{cases}$$

First look at the standard two-valued semantics of atomic formulae, that is:

$$(\!|R(\bar{x})|\!)^{\mathsf{bool}}_{D,\bar{a}} = \begin{cases} \mathbf{t} & \bar{a} \in R^D \\ \mathbf{f} & \bar{a} \notin R^D \end{cases} \quad ; \quad (\!|x = y|\!)^{\mathsf{bool}}_{D,(a,b)} = \begin{cases} \mathbf{t} & a = b \\ \mathbf{f} & a \neq b \end{cases} \tag{12}$$

This does not have correctness guarantees. Consider D with the fact $R(1, \perp)$. Then for $\bar{a} = (1, 1)$ we have $(\!|R(\bar{x})|\!)^{\mathsf{bool}}_{D,\bar{a}} = \mathbf{f}$ which, if we had correctness guarantees, would have implied $\bar{a} \in \mathsf{cert}_\perp(\neg\varphi, D)$. However, the valuation $v(\perp) = 1$ shows that $\bar{a} \notin \mathsf{cert}_\perp(\neg\varphi, D)$

Thus, we need a more refined procedure to ensure correctness. By Theorem 5.1, we only need to provide the semantics of atomic formulae. For this, recall the notion of *unification* of two tuples $\bar{r}$ and $\bar{s}$: they are unifiable, written $\bar{r} \Uparrow \bar{s}$, if there is a valuation v such that $v(\bar{r}) = v(\bar{s})$. This is known to be checkable in linear time [57]. We then define the three-valued semantics as follows:

$$(\!|R(\bar{x})|\!)^{\mathsf{unif}}_{D,\bar{a}} = \begin{cases} \mathbf{t} & \bar{a} \in R^D \\ \mathbf{f} & \nexists \bar{b} \in R^D \text{ such that } \bar{a} \Uparrow \bar{b} \\ \mathbf{u} & \text{otherwise} \end{cases} \tag{13a}$$

$$(\!|x = y|\!)^{\mathsf{unif}}_{D,(a,b)} = \begin{cases} \mathbf{t} & a = b \\ \mathbf{f} & a \neq b \text{ and } a, b \in \mathsf{Const} \\ \mathbf{u} & \text{otherwise} \end{cases} \tag{13b}$$

The first rule says that $\bar{a} \notin R^D$ is not yet enough to declare the truth value of $R(\bar{a})$ to be **f**. There may be a tuple $\bar{b} \in R^D$ unifiable with $\bar{a}$, and thus in some possible world given by a valuation v we may have $v(\bar{a}) \in v(R^D)$. Thus, only when there is no tuple in R^D that unifies with $\bar{a}$ can we say with certainty that $R(\bar{a})$ is false. The same reasoning for equalities tells us that we can only be sure that $a \neq b$ when both a and b are different *constants*. Otherwise, if a and b are different, we need to assign truth value **u** to their comparison.

Corollary 5.2 (see [51]). *The $(\!|\cdot|\!)^{unif}$ semantics has correctness guarantees with respect to certain answers with nulls:*

$$\text{if } (\!|\varphi|\!)^{unif}_{D,\bar{a}} = \mathbf{t}\,, \text{ then } \bar{a} \in \mathsf{cert}_\perp(\varphi, D)\,.$$

Moreover, $(\!|\varphi|\!)^{unif}_{D,\bar{a}}$ can be computed with AC^0 *data complexity.*

If one can have three-valued evaluation procedures with correctness guarantees, then why SQL does not provide them? We shall explain this in the next section; for now we offer a simple example. Given three unary relations R, S, T, all with a single attribute A, let Q compute $R - (S - T)$:

```
SELECT R.A FROM R WHERE R.A NOT IN
  ( SELECT S.A FROM S WHERE S.A NOT IN
    (  SELECT * FROM T ) )
```

If $R^D = S^D = \{1\}$ and $T^D = \{\bot\}$, we have $Q(D) = \{1\}$. However, 1 is not a certain answer, and moreover $\mu(Q, D, 1) = 0$. Even with available correct three-valued procedures, SQL can give answers that not only fail to be certain, but do so with probability 1.

5.2 Does SQL need many-valued logics?

Much of the criticism towards SQL revolves around its behavior with incomplete information, that is, with null values. As we explained earlier, SQL operates with Kleene's logic $\mathbb{L}_{3v}$, having truth values **t**, **f**, and **u** (i.e., unknown). This was a decision of the committee that designed the language, and we now address the following questions:

- Was Kleene's logic the right choice for handling incomplete information?
- Does it add any extra expressiveness compared to the usual Boolean logic for FO queries?

In what follows, we give a brief account of the results from [21] that provide answers to these questions – yes and no, respectively.

Propositional logic for incompleteness. To understand what a predicate logic for incompleteness should look like, we need to give a reasonable meaning to its truth values. To this end, we follow the approach of [33] and use sets of possible worlds to model incompleteness.

For a language of propositional formulae $\mathcal{L}$, a *propositional interpretation* for $\mathcal{L}$ is a triple (W, t, f) where W is a set of possible worlds, and t and f are functions mapping formulae of $\mathcal{L}$ into subsets of W. Intuitively, $t(\alpha)$ is the set of worlds that satisfy α, and $f(\alpha)$ is the set of worlds that do not. In our framework, we allow $t(\alpha) \cup f(\alpha) \neq W$, i.e., the knowledge we have on α may be partial. We do require that $t(\alpha) \cap f(\alpha) = \emptyset$, as we do not consider inconsistent interpretations. Taking propositional interpretations directly as truth values was advocated in [33]. This however gives rise to too many truth values. Instead, we want an informative and compact representation of what we know about formulae. To define such a representation, we resort to *epistemic* logic, and define the epistemic modalities of a propositional formula $\alpha \in \mathcal{L}$ as the formulae $\mathbf{K}(\alpha)$, $\mathbf{P}(\alpha)$, and their negations. A propositional interpretation (W, t, f) satisfies $\mathbf{K}(\alpha)$ if $t(\alpha) = W$, that is, we **k**now α. It satisfies $\mathbf{P}(\alpha)$ if $t(\alpha) \neq \emptyset$, that is, α is **p**ossible.

As the truth values for our logic, we take maximally consistent theories of epistemic modalities of a formula and its negation. For every given formula $\alpha \in \mathcal{L}$, we can show that there are at most only six such theories, corresponding to the following scenarios:

- α is true in all worlds (truth value **t**);
- α is false in all worlds (truth value **f**);
- α is true in some worlds, false in others (truth value **s**, meaning "sometimes");
- There is a world where α is true, but we do not know whether α is always true (truth value **st**, meaning "sometimes true");
- There is a world where α is false, but we do not know whether α is always false (truth value **sf**, meaning "sometimes false");
- We have no information whatsoever on α (truth value **u**, "unknown").

To derive the truth tables of the propositional connectives $\wedge$, $\vee$ and $\neg$, we denote by χ^τ_α, for a propositional formula α, the epistemic formula that expresses the maximally consistent theory stating that α has truth value τ. For example, $\chi^{\mathbf{t}}_\alpha = \mathbf{K}(\alpha) \wedge \mathbf{P}(\alpha) \wedge \neg\mathbf{K}(\neg\alpha) \wedge \neg\mathbf{P}(\neg\alpha)$. Then, we impose the following two requirements:

- If $\omega(\tau_1, \tau_2) = \tau$, then τ *is consistent with* τ_1, τ_2, i.e., the epistemic formula $\chi^{\tau_1}_\alpha \wedge \chi^{\tau_2}_\beta \wedge \chi^{\tau}_{\omega(\alpha,\beta)}$ is satisfiable; and
- When more than one truth value is consistent with τ_1, τ_2, we choose the most general one.

The logic derived in this way, denoted by $\mathbb{L}_{6v}$, is perfectly equipped to handle incomplete information. To be used in database systems, however, a logic needs to be compatible with standard query optimizations, which always require distributivity and idempotency (see [34, 44]). The logic $\mathbb{L}_{6v}$ is neither distributive nor idempotent. Therefore, we want to find a maximal sublogic of it with these two properties that make it suitable for the database context.

Theorem 5.3 (see [21]). *The maximal sublogic of* $\mathbb{L}_{6v}$ *that is both distributive and idempotent is Kleene's three-valued logic* $\mathbb{L}_{3v}$.

Thus, purely at the propositional level, SQL designers did choose the right logic for handling incompleteness.

Predicate logics and SQL. Using Kleene's three-valued logic in SQL appears to be well justified. However, our justification applies purely at the propositional level, and SQL is, after all, based on a predicate logic. Thus, we would like to understand whether Kleene's logic is still necessary.

We consider many-valued FO based on a propositional many-valued logic $\mathbb{L}$ as defined above, i.e., $(\mathsf{FO}(\mathbb{L}), (\!|\cdot|\!))$. Recall that we lift the semantics from atomic formulae to arbitrary ones by following the semantics of the propositional connectives, as in (10) and (11). Now we look at the common semantics of atoms. The most natural one is, of course, the Boolean semantics given by (12). We have also seen the *unification* semantics given by (13a) and (13b). As our final example, we consider a *null-free semantics* corresponding to the way SQL treats comparisons:

$$(\!|R(\bar{x})|\!)^{\mathsf{nullfree}}_{D,\bar{a}} = \begin{cases} \mathbf{t} & \bar{a} \in R^D \text{ and } \mathsf{Const}(\bar{a}) \\ \mathbf{f} & \bar{a} \notin R^D \text{ and } \mathsf{Const}(\bar{a}) \\ \mathbf{u} & \neg\mathsf{Const}(\bar{a}) \end{cases} \tag{14}$$

where $\mathsf{Const}(\bar{a})$ states that $\bar{a}$ consists of constants only. This could apply to the equality predicate as well, by simply viewing it as an extra relation $\mathsf{Eq}^D = \{(a, a) \mid a \in \mathsf{dom}(D)\}$.

These semantics give rise to a number of different logics. The familiar FO we dealt with previously is simply $\mathsf{FO}(\mathbb{L}_{2v}, [\![]\!]_{\mathsf{bool}})$. To define $\mathsf{FO}_{\mathsf{SQL}}$, a logic underlining SQL, we need a semantics that combines some of the previously defined ones. A *mixed semantics* is any semantics $(\!|\cdot|\!)$ of base relations, including Eq, that uses one of $(\!|\cdot|\!)^{\mathsf{bool}}$, $(\!|\cdot|\!)^{\mathsf{unif}}$ and $(\!|\cdot|\!)^{\mathsf{nullfree}}$, for different base relations.

The following mixed semantics captures the behavior of SQL:

$$(\!|R(\bar{x})|\!)^{\mathsf{sql}}_{D,\bar{a}} = \begin{cases} (\!|R(\bar{x})|\!)^{\mathsf{bool}}_{D,\bar{a}} & R \text{ relation of the schema} \\ (\!|R(\bar{x})|\!)^{\mathsf{nullfree}}_{D,\bar{a}} & R = \mathsf{Eq} \end{cases} \tag{15}$$

and gives rise to the FO core of SQL, namely

$$\mathsf{FO}_{\mathsf{SQL}} = (\mathsf{FO}(\mathbb{L}_{3v}), (\!|\cdot|\!)^{\mathsf{sql}}).$$

What is the expressiveness of all these different logics? It looks as though we have a huge space of possibilities, but as a matter of

fact we do not. We say that Boolean FO *captures* a logic $(\mathsf{FO}(\mathbb{L}), (\!|\cdot|\!))$ if for every formula $\varphi(\bar{x})$ of $(\mathsf{FO}(\mathbb{L}), (\!|\cdot|\!))$ and every truth value $\tau \in \mathbf{T}$ there exists an FO ψ^τ such that $(\!|\varphi(\bar{x})|\!)_{D,\bar{a}} = \tau$ iff $D \models \psi^\tau(\bar{a})$.

Theorem 5.4 (see [21]). *Boolean* FO *captures* $(\mathsf{FO}(\mathbb{L}_{3\mathrm{v}}), (\!|\cdot|\!))$ *for every mixed semantics* $(\!|\cdot|\!)$.

The result of [21] is again more general: Boolean FO captures every many-valued logic $(\mathsf{FO}(\mathbb{L}), (\!|\cdot|\!))$ as long as it captures atomic formulae and $\mathbb{L}$ has connectives $\wedge$ and $\vee$ that are weakly idempotent ($a \vee a \vee a = a \vee a$, and likewise for $\wedge$).

This seems to suggest that all SQL queries evaluated under the three-valued semantics can be expressed in the usual Boolean version of FO. While this is the case, at least for the fragment of SQL that corresponds to relational algebra queries, we need to be careful. To see why, recall that SQL's approach is to keep tuples on which the evaluation of logical conditions produces the truth value **t**, i.e.:

$$Q_\varphi(D) = \left\{ \bar{a} \mid (\!|\varphi|\!)^{\mathsf{sql}}_{D,\bar{a}} = \mathbf{t} \right\}.$$

Then [52] showed that for every formula φ of $\mathsf{FO}(\mathbb{L}_{3\mathrm{v}})$ the condition $\bar{a} \in Q_\varphi(D)$ implies $\mu(Q_\varphi, D, \bar{a}) = 1$, i.e., $\bar{a}$ is an almost certainly true answer. But at the end of the previous section we saw that SQL queries can return tuples that are almost certainly false.

What is the source of the mismatch? The textbook approach comes close to describing the logic of SQL, but it misses one important feature of such logic. We can think of core SQL queries as expressions of the form:

SELECT	$\bar{x}$
FROM	$Q_1, \ldots, Q_n$
WHERE	$\theta(\bar{x}_1, \ldots, \bar{x}_n)$

where $Q_1, \ldots, Q_n$ are either queries or relations, $\bar{x}_i$ is a tuple of variables returned by Q_i, and θ is a condition composed of equality of variables, or statements $Q'(\bar{y})$ in which Q' is another query, or statements $Q' \neq \varnothing$, or a combination of these via $\wedge$, $\vee$, and $\neg$.

Note that, in SQL's query evaluation, it is the conditions θ that are evaluated in $\mathbb{L}_{3\mathrm{v}}$; once the evaluation of the **WHERE** θ clause is finished, only tuples that evaluated to **t** are kept, i.e., one goes back to two-valued logic. To capture this, we need a propositional operator that collapses **f** and **u** into **f**. Such an operator does exist in propositional many-valued logics [12] and is known as an *assertion* operator: $\uparrow p$ for a proposition p evaluates to **t** if p evaluates to **t**, and to **f** otherwise. Let $\mathbb{L}^{\uparrow}_{3\mathrm{v}}$ be the extension of $\mathbb{L}_{3\mathrm{v}}$ with this operator.

The basic SQL query can then be expressed in $\mathsf{FO}(\mathbb{L}^{\uparrow}_{3\mathrm{v}})$:

$$Q(\bar{x}) \;=\; \exists \bar{y} \bigwedge_{i=1}^{n} Q_i(\bar{x}_i) \wedge \uparrow\theta(\bar{x}_1, \ldots, \bar{x}_n),$$

where $\bar{y}$ lists variables present in $\bar{x}_1, \ldots, \bar{x}_n$ but not $\bar{x}$. Thus, the many-valued predicate logic capturing SQL's behavior is

$$\mathsf{FO}^{\uparrow}_{\mathsf{SQL}} = \left(\mathsf{FO}(\mathbb{L}^{\uparrow}_{3\mathrm{v}}), (\!|\cdot|\!)^{\mathsf{sql}}\right)$$

rather than just $\mathsf{FO}_{\mathsf{SQL}}$. But even this logic is no more expressive than Boolean FO.

Theorem 5.5 (see [21]). *For every formula* $\varphi(\bar{x})$ *of* $\mathsf{FO}^{\uparrow}_{\mathsf{SQL}}$*, the query* Q_φ *is expressible in Boolean* FO.

Therefore, while justified at the propositional level, the use of a three-valued logic is not needed to handle incomplete databases, at least for queries capturing the expressiveness of relational algebra.

We conclude by explaining why queries in $\mathsf{FO}^{\uparrow}_{\mathsf{SQL}}$ (and thus real-life SQL) can produce almost certainly false answers, while queries in $\mathsf{FO}_{\mathsf{SQL}}$ cannot. It turns out that being almost certainly true is guaranteed if the connectives of the many-valued logic preserve the knowledge order $\mathbf{u} \preceq \mathbf{t}$ and $\mathbf{u} \preceq \mathbf{f}$. The usual connectives $\wedge$, $\vee$, $\neg$ are such, but the assertion operator is not: while $\mathbf{u} \preceq \mathbf{t}$, we do not have $\uparrow\mathbf{u} \;\preceq\; \uparrow\mathbf{t}$. Therefore, the real culprit in SQL's behavior is not the three-valued logic per se (even though it can be completely avoided) but rather the *mix* of two- and three-valued logics, i.e., not carrying full information about the three truth values through the entire query evaluation.

6 OPEN PROBLEMS

We conclude by discussing possible avenues for future research on incomplete information, which we mostly foresee at the intersection of theory and practice. On the one hand, theoretical investigations should be focused on problems that have practical relevance; for example, queries with arithmetic and aggregation in typed data models (not necessarily limited to the relational setting) under bag semantics. On the other hand, real systems should be built and extended by taking into account the lessons learned from theory; in particular, we need implementations of models of incompleteness that are more expressive and flexible than what is currently provided by SQL. Below, we discuss some of these open problems in more detail.

Bag semantics. Real-life RDBMSs use a data model that is based on bags, where tuples in a relation are allowed, unlike for sets, to occur more than once. While much theoretical research – like most of the works reviewed in this survey – is focused on set semantics, in recent years there have been several efforts towards handling incomplete data in bag databases [20, 22, 42, 55]. However, it is still early days for the study of certain answers under bag semantics, and several questions remain open. Under bags, as mentioned in Section 4.2, we have more fine grained information about tuples, namely their minimum and maximum number of occurrences across all possible worlds. In [20] and [22], the minimum multiplicity is taken as the notion of certainty, but is this the right one? When working with bags, valuations can be applied to databases in different ways: tuples that unify under a valuation can be collapsed, or their multiplicities added up [42]. What are the differences between those when defining and computing certain answers? Are there other possibilities?

Marked nulls. While theoretical models of incompleteness rely on marked nulls, in real-life DBMSs based on SQL there is only one single placeholder object for representing missing values: **NULL**. To bridge this gap, the common approach is to interpret each occurrence of **NULL** as a distinct marked null. Then, denoting by codd this transformation of SQL nulls into non-repeating marked nulls, known as *Codd nulls*, it must hold that $Q\big(\mathrm{codd}(D)\big)$ and $\mathrm{codd}\big(Q(D)\big)$ are the same, up to renaming of nulls, for every database D and query Q. That is, transforming SQL nulls to Codd nulls before or after evaluating queries makes no difference. However, this fails

in general, and the class of queries having this property cannot be captured by a syntactic fragment. While syntactic restrictions that enforce it exist [39], it seems more interesting to follow a different approach: implement marked nulls directly in SQL. This would increase SQL's expressiveness (e.g., by allowing one to state that two persons have the same unknown age) and avoid some of the semantic ambiguity around **NULL** (e.g., missing versus non-applicable or undefined values). Of course, this raises several interesting questions, for example: is an implementation of marked nulls possible using only standard SQL features? What are the costs in terms of storage requirements and performance of queries?

Quality of approximations. As we have seen in Section 4, several ways to approximate certain answers exist in the literature. But how good are the approximations they provide? For the approximation schemes [36, 37, 51] presented in Section 4.2 we can say the following: first, the answers they provide on databases without nulls coincide with those returned by evaluating the original query; second, there is a strict containment (witnessed by specific queries and databases) between the answers returned by any two of the four algorithms proposed in [36]; third, the answers returned by the approximation schemes of [51] and [37] are incomparable (w.r.t. subset inclusion) in general.

Comparing the quality of different approximation procedures is not always easy. When feasible, one can of course perform experimental analyzes in the spirit of [27] (cf. also [37] and [28]), but we also need principled theoretical approaches. One possibility is to analyze the knowledge provided by an approximation procedure and compare it with the certain knowledge the query and the database give us. Understanding the amount of certain knowledge a procedure is able to compute, and what logic, if any, can express it, would give us a nice characterization of the quality of the approximation provided. Another idea is to use the probabilistic approach of Section 4.3 to estimate how likely a tuple is to be returned by one approximation but not the other.

Value-inventing queries. Data aggregation and arithmetic operations are among the most used features in SQL, with aggregation alone amounting for the vast majority of the query workloads of many popular benchmarks. For example, no less than 80% of the analytical queries included in the well-known TPC Benchmark H use aggregation [65]. What these operations have in common is that they can (and typically do) produce values that are not present in the database. The theoretical tools we currently have for dealing with incompleteness prove inadequate for value-inventing operations. Indeed, as mentioned in Section 3, the widespread certain answers with nulls cannot return values that do not appear in the original database. The way the problem is addressed in practical scenarios is also unsatisfactory: the common approach is to apply imputation techniques that replace nulls with "likely" values, and then process analytical queries in the standard way *as if* the data were complete [29]. We need new techniques for applying the notions of certainty (either exact or approximate) in the context of value-inventing queries.

Types of attributes. The results shown here, and in general many results on handling incomplete information, are presented in the standard theoretical model of relational databases with the domain consisting of constants and nulls, and with equality and disequality being the only available predicates. In real life, of course, database columns have types: numerical, strings, etc. How can we extend results to the typed model, where type-specific operations, such as arithmetic, are used in queries? In some cases there are simple extensions: for example, the approximation schemes of Section 4.2 can be used by treating type-specific comparisons similarly to disequalities. On the other hand, the probabilistic approximations of Section 4.3 are more complicated, though they can provide more refined information about query answers [23]. Bridging the gap between theoretical models and typed real-life schemas is essential for applying these results in practice.

Other data models. We focused on the relational model, but there are other data models where incompleteness naturally occurs. For XML, for example, incompleteness has been extensively studied [2, 9]. For the very popular and fast developing model of graph databases [14] there has been some initial work [3, 30, 40, 56] but we are nowhere near a well-developed theory similar to what was presented here for relational data. A new standard query language for graph databases called GQL is being designed [68]. This provides an opportunity to influence the design of its null-related features that would lead to less criticism than that experienced by SQL.

Certain answers as knowledge. The framework of [49] presented in Section 3.2 has been used to define certain answers as objects, but there is more to it and we believe it should be further developed. One example of the use the use of the framework is [50], which defined negative and possible answers with the help of certain knowledge. It can also be used to handle semantically opaque object identifiers, a problem which is especially relevant in data integration and OBDA, where object identifiers are retrieved from multiple sources and are often encoded using different conventions. An example of this is the notion of referring answers [15], which encode the certain knowledge entailed by answer tuples.

ACKNOWLEDGMENTS

Most results presented here were obtained by the authors while they were supported by EPSRC projects M025268 "VADA: Value Added Data Systems" and N023056 "Managing Incomplete Data – New Foundations". Work on the semantics of SQL was in conjunction with our collaboration with Neo4j Inc, supported by a grant from them. Finally, the third author is grateful to Foundation Sciences Mathématiques de Paris for supporting his stay in Paris in the Fall of 2019, during which some of this work was done.

REFERENCES

[1] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78(1):158–187, 1991.
[2] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. *ACM TODS*, 31(1):208–254, 2006.
[3] S. Ahmetaj, W. Fischl, R. Pichler, M. Simkus, and S. Skritek. Towards reconciling SPARQL and certain answers. In *WWW*, pages 23–33, 2015.
[4] G. Amendola and L. Libkin. Explainable certain answers. In *IJCAI*, pages 1683–1690, 2018.
[5] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.
[6] M. Arenas, E. Botoeva, E. Kostylev, and V. Ryzhikov. A note on computing certain answers to queries over incomplete databases. In *AMW*, 2017.
[7] O. Arieli and A. Avron. Reasoning with logical bilattices. *Journal of Logic, Language and Information*, 5(1):25–63, 1996.

[8] O. Arieli, A. Avron, and A. Zamansky. What is an ideal logic for reasoning with inconsistency? In *IJCAI*, pages 706–711, 2011.

[9] P. Barceló, L. Libkin, A. Poggi, and C. Sirangelo. XML with incomplete information. *Journal of the ACM*, 58(1):4:1–4:63, 2010.

[10] N. D. Belnap. A useful four-valued logic. In J. M. Dunn and G. Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 8–37. D. Reidel, 1977.

[11] M. Bienvenu and M. Ortiz. Ontology-mediated query answering with data-tractable description logics. In *Reasoning Web*, pages 218–307, 2015.

[12] D. A. Bochvar. On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus. *History and Philosophy of Logic*, 2:87–112, 1981. Translated from *Matematicheskij Sbornik*, 4(46) 287–308,1938.

[13] L. Bolc and P. Borowik. *Many-Valued Logics: Theoretical Foundations*. Springer, 1992.

[14] A. Bonifati, G. H. L. Fletcher, H. Voigt, and N. Yakovets. *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[15] A. Borgida, D. Toman, and G. E. Weddell. On referring expressions in query answering over first order knowledge bases. In *KR*, pages 319–328. AAAI Press, 2016.

[16] F. Capelli, A. Durand, A. Gheerbrant, and C. Sirangelo. Private communication.

[17] C. Chang and H. Keisler. *Model Theory*. North Holland, 1990.

[18] K. Compton. Some useful preservation theorems. *Journal of Symbolic Logic*, 48(2):427–440, 1983.

[19] M. Console, P. Guagliardo, and L. Libkin. Approximations and refinements of certain answers via many-valued logics. In *KR*, pages 349–358. AAAI Press, 2016.

[20] M. Console, P. Guagliardo, and L. Libkin. On querying incomplete information in databases under bag semantics. In *IJCAI*, pages 993–999. ijcai.org, 2017.

[21] M. Console, P. Guagliardo, and L. Libkin. Propositional and predicate logics of incomplete information. In *KR*, pages 592–601. AAAI Press, 2018.

[22] M. Console, P. Guagliardo, and L. Libkin. Fragments of bag relational algebra: Expressiveness and certain answers. In *ICDT*, volume 127 of *LIPIcs*, pages 8:1–8:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[23] M. Console, M. Hofer, and L. Libkin. Queries with arithmetic on incomplete databases. In *PODS*, 2020.

[24] C. J. Date. *Database in Depth - Relational Theory for Practitioners*. O'Reilly, 2005.

[25] C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 1996.

[26] R. Fagin. Probabilities on finite models. *Journal of Symbolic Logic*, 41(1):50–58, 1976.

[27] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Uncertainty annotated databases - A lightweight approach for approximating certain answers. In *SIGMOD Conference*, pages 1313–1330. ACM, 2019.

[28] N. Fiorentino, S. Greco, C. Molinaro, and I. Trubitsyna. ACID: A system for computing approximate certain query answers over incomplete databases. In *SIGMOD Conference*, pages 1685–1688. ACM, 2018.

[29] Y. Gao and X. Miao. *Query Processing over Incomplete Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[30] A. Gheerbrant and G. Fontaine. Querying incomplete graphs with data. In *AMW*, 2014.

[31] A. Gheerbrant and L. Libkin. Certain answers over incomplete XML documents: Extending tractability boundary. *Theory Comput. Syst.*, 57(4):892–926, 2015.

[32] A. Gheerbrant, L. Libkin, and C. Sirangelo. Naïve evaluation of queries over incomplete databases. *ACM Trans. Database Systems*, 39(4):31:1–31:42, 2014.

[33] M. L. Ginsberg. Multivalued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[34] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[35] J. Grant. Null values in a relational data base. *Inf. Process. Lett.*, 6(5):156–157, 1977.

[36] S. Greco, C. Molinaro, and I. Trubitsyna. Approximation algorithms for querying incomplete databases. *Inf. Syst.*, 86:28–45, 2019.

[37] P. Guagliardo and L. Libkin. Making SQL queries correct on incomplete databases: A feasibility study. In *PODS*, pages 211–223. ACM, 2016.

[38] P. Guagliardo and L. Libkin. Correctness of SQL queries on databases with nulls. *SIGMOD Record*, 46(3):5–16, 2017.

[39] P. Guagliardo and L. Libkin. On the Codd semantics of SQL nulls. *Inf. Syst.*, 86:46–60, 2019.

[40] C. Gutiérrez, D. Hernández, A. Hogan, and A. Polleres. Certain answers for SPARQL? In *AMW*, 2016.

[41] P. Hell and J. Nešetřil. *Graphs and homomorphisms*. Oxford University Press, 2004.

[42] A. Hernich and P. G. Kolaitis. Foundations of information integration under bag semantics. In *LICS*, pages 1–12. IEEE Computer Society, 2017.

[43] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

[44] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[45] M. Lenzerini. Data integration: a theoretical perspective. In *PODS*, pages 233–246, 2002.

[46] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[47] L. Libkin. Incomplete information and certain answers in general data models. In *PODS*, pages 59–70, 2011.

[48] L. Libkin. Incomplete information: what went wrong and how to fix it. In *PODS*, pages 1–13, 2014.

[49] L. Libkin. Certain answers as objects and knowledge. *Artificial Intelligence*, 232:1–19, 2016.

[50] L. Libkin. Negative knowledge for certain query answers. In *RR*, volume 9898 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2016.

[51] L. Libkin. SQL's three-valued logic and certain answers. *ACM Trans. Database Syst.*, 41(1):1:1–1:28, 2016.

[52] L. Libkin. Certain answers meet zero-one laws. In *PODS*, pages 195–207, 2018.

[53] W. Lipski. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4(3):262–296, 1979.

[54] W. Lipski. On relational algebra with marked nulls. In *PODS*, pages 201–203, 1984.

[55] C. Nikolaou, E. V. Kostylev, G. Konstantinidis, M. Kaminski, B. C. Grau, and I. Horrocks. Foundations of ontology-based data access under bag semantics. *Artif. Intell.*, 274:91–132, 2019.

[56] C. Nikolaou and M. Koubarakis. Incomplete information in RDF. In *RR*, pages 138–152, 2013.

[57] M. Paterson and M. N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.

[58] R. Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977.

[59] R. Reiter. Equality and domain closure in first-order databases. *Journal of the ACM*, 27(2):235–249, 1980.

[60] R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *Journal of the ACM*, 33(2):349–347, 1986.

[61] E. Rosen. Some aspects of model theory and finite structures. *Bulletin of Symbolic Logic*, 8(3):380–403, 2002.

[62] B. Rossman. Homomorphism preservation theorems. *Journal of the ACM*, 55(3), 2008.

[63] J. Spencer. *The Strange Logic of Random Graphs*. Springer, 2001.

[64] A. Stolboushkin. Finitely monotone properties. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 324–330, 1995.

[65] Transaction Processing Performance Council (TPC). *TPC Benchmark™ H Standard Specification*, revision 2.18.0 edition, 2018. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf`.

[66] R. van der Meyden. Logical approaches to incomplete information: A survey. In *Logics for Databases and Information Systems*, pages 307–356, 1998.

[67] M. Y. Vardi. Querying logical databases. *Journal of Computer and System Sciences*, 33(2):142–160, 1986.

[68] Wikipedia contributors. GQL graph query language, 2020.