Approximations and Refinements of Certain Answers via Many-Valued Logics

Marco Console  
Sapienza Università di Roma

Paolo Guagliardo and Leonid Libkin  
University of Edinburgh

Abstract
Computing certain answers is the preferred way of answering queries in scenarios involving incomplete data. This, however, is computationally expensive, so practical systems use efficient techniques based on a particular three-valued logic, even though this often leads to incorrect results. Our goal is to provide a general many-valued framework for correctly approximating certain answers. We do so by defining the semantics of many-valued answers and queries, following the principle that additional knowledge about the input must translate into additional knowledge about the output. This framework lets us compare query outputs and evaluation procedures in terms of their informativeness. For each many-valued logic with a knowledge ordering on its truth values, one can build a syntactic evaluation procedure for all first-order queries, that correctly approximates certain answers; additional truth values are used to refine information about certain answers. For concrete examples, we show that a recently proposed approach fixing some of the inconsistencies of SQL query evaluation is an immediate consequence of our framework, and we further refine it by adding a fourth truth value. We show that no evaluation procedure based on Boolean logic delivers correctness guarantees. Finally, we study the relative power of evaluation procedures based on the informativeness of the answers they produce.

1 Introduction
In a variety of data processing tasks, especially those where knowledge representation techniques play an important role, query answering is often reduced to finding those answers that hold with certainty over an incompletely described data set. Such examples include data integration (Lenzerini 2002; Halevy, Rajaraman, and Ordille 2006), data exchange (Arenas et al. 2014), inconsistent databases (Bertossi 2011), and ontology-based query answering (Calvanese et al. 2007; Kontchakov et al. 2011). In many cases, one builds a database, typically incomplete, and then answers queries over it. The latter of course is often done using traditional database query evaluation methods.

When it comes to evaluation of queries over incomplete databases, we have a well-established theory, as well as practice of relational query answering; both date back more than 30 years, and somehow they are totally out of sync with each other, as extensively discussed by Libkin (2014). The standard theoretical approach to answering queries on incomplete databases is to compute certain answers, i.e., those answers one can be sure about for every substitution of missing values. The main drawback of this approach is its high computational complexity outside restricted classes of queries (Abiteboul, Kanellakis, and Grahne 1991).

In practice, of course, high complexity cannot be tolerated, so systems such as relational DBMSs devise efficient algorithms that are based on a three-valued logic (3VL) for handling incomplete data (Date and Darwen 1996). In addition to the usual true (t) and false (f), 3VL operates with the value unknown (u) to account for operations with incomplete information.

While one might expect that the approaches accepted in practice have something to do with theory, it is not so: standard relational query evaluation based on 3VL and certain answers can be completely unrelated. This was observed recently by Libkin (2016b), who also showed that a modified query evaluation procedure can provide certainty guarantees at least for all first-order (FO) queries, i.e., the backbone of relational query languages. The key idea was to refine the notion of certain answers, and classify possible answers not into “certainly true” and the rest, but rather into “certainly true”, “certainly false”, and the rest. Computing these answers precisely is an intractable task, but one can obtain an efficient evaluation procedure that gives an approximation.

To give a very simple example, consider the following incomplete database and query:

\[ R = \{1, 2\} \quad S = \{\bot, 2\} \quad \varphi(x) = R(x) \land \neg S(x). \] (1)

Here \( \bot \) indicates a missing (currently unknown) value, and \( \varphi \) computes the difference of \( R \) and \( S \). The evaluation procedure of Libkin (2016b) will annotate 1 with \( u \), since it may or may not be in the query answer depending on the interpretation of \( \bot \), and 2 will be annotated with \( f \). The classical definition of certain answers (Abiteboul, Hull, and Vianu 1995; Imieliński and Lipski 1984) which accounts only for “certainly true” information will only tell us that there are no answers (missing the useful bit of information that 2 is never in the answer), but the evaluation by a relational DBMS (more precisely, of the SQL query

\[ \text{SELECT * FROM R EXCEPT} \]
SELECT * FROM S) will return value 1, which is an incorrect – false positive – result.

The goal of this paper is to provide a many-valued framework for obtaining efficient procedures that compute correct approximations of certain answers. They do so by refining the rather coarse information carried by certain answers themselves, in a way that is guided by the underlying many-valued logic. The procedure of Libkin (2016b) mentioned earlier will be just one application of such a framework, when the underlying logic is 3VL, used in commercial DBMSs. To achieve this, we need to answer the following questions:

- How do we define many-valued query answers? What is their semantics, and how can we compare them in terms of the information they contain?
- When can an evaluation scheme be viewed as an approximation for a given semantics of query answers?
- How can we obtain such evaluation schemes if we are given a many-valued logic and a semantics of query answers for it?

To understand the importance of the first question, note that queries Q on incomplete databases must satisfy the information preservation principle (Libkin 2014; 2016a): if a database D′ has at least as much information as a database D, then the answer Q(D′) has at least as much information as the answer Q(D). Indeed, if we are given a less informative D′, then at the very least we can run Q on the less informative D; thus Q(D′) adds information to Q(D).

Since the notion of being more informative is given by the semantics of databases, and query answers are (possibly incomplete) databases themselves, we need to understand their semantics, which no one has properly defined in the context of many-valued query evaluation, as practiced by commercial DBMSs. Understanding and defining such semantics, and the notion of informativeness that comes with it, is the first contribution of this paper.

To define the semantics of many-valued answers we appeal to a well known observation (Belnap 1977) that truth values can be ordered by both their degree of truth, and their degree of knowledge, or informativeness. The key property of the latter is that propositional connectives, such as ∧, ∨ and ¬, preserve it, i.e., they return more informative results when more informative inputs are given (Arieli and Avron 1996; 1998; Ginsberg 1988). Based on this, we define many-valued semantics and informativeness ordering for query answers, with which we can proceed to answer the remaining questions.

Evaluation procedures assign truth values to tuples in such a way that the resulting answer is no more informative than the actual query answer (which may be too costly to compute). Among different evaluation procedures, we identify a particularly nice class that comes with rather strong correctness guarantees. Our next results give a technique for generating procedures in this class. This requires only very mild conditions on the many-valued logic and the query semantics. The evaluation procedures generated by this technique have low computational complexity, either the same as for FO query evaluation (AC^0), or slightly higher in terms of parallel complexity (NC^1).

We then look at concrete applications of such a technique. We show that the 3VL evaluation procedure of Libkin (2016b) that corrected SQL’s shortcomings is a natural example of the general technique for 3VL. We also look at a 4-valued logic 4VL that further refines 3VL. While 4-valued logics have been considered in relational databases to handle different types of incompleteness (e.g., Gessert 1990), our idea is to provide finer information on possible answers. Consider again the example given in (1). While for possible answer 1, SQL evaluation returned the incorrect value t, the 3VL evaluation with correctness guarantees produced value u, or unknown. However, we do know something about 1, namely that depending on the interpretation of the null ⊥, it may, or may not be in the answer. Hence, it cannot be true with certainty, nor can it be false with certainty. This is extra information compared to u, and it can be captured by an additional truth value in 4VL. We provide an efficient evaluation procedure for this logic, and show that it properly refines the 3VL evaluation procedure.

Finally, we look at the usual two-valued Boolean logic, and show that no evaluation procedure that respects the rules for ∧, ∨, ¬ can guarantee correctness. This justifies the decision by relational DBMS designers to use 3VL, rather than Boolean logic, for handling nulls (although they did not get it quite right). We also use the information ordering to show that among all 3VL procedures that are built inductively on the structure of the query, the one our approach generates is the most informative.

**Organization** Basic notions and definitions are given in Section 2. Section 3 presents the main ideas of the many-valued framework, and Section 4 shows how to achieve correctness guarantees in this framework. Section 5 presents a case study for first-order logic queries based on 3VL and 4VL semantics, and also shows that the usual Boolean two-valued logic cannot provide correctness guarantees. Section 6 shows how to compare evaluation procedures and discusses their optimality. Conclusions are in Section 7.

### 2 Preliminaries

As was explained in the introduction, we are motivated by applications that rely on the standard database technology, and thus we follow a typical database approach to incompleteness, which we outline now. Of course there are many other representations of incompleteness that occur in AI, for instance in logic programming (Gelfond and Lifschitz 1991) or rough sets settings (Doherty et al. 2006).

**Incomplete Databases** We consider databases with missing values represented by marked, also called naïve or labeled, nulls (Abiteboul, Hull, and Vianu 1995; Imielinski and Lipski 1984). These are typical in applications such as data integration and exchange (Arenas et al. 2014; Lenzerini 2002), common in database applications (Grahne 1991; van der Meyden 1998), and furthermore they subsume null values that occur in relational DBMSs (Date and Darwen 1996; 1998; Ginsberg 1988). Based on this, we define many-valued semantics, which no one has properly defined in the context of many-valued query evaluation, as practiced by commercial DBMSs. Understanding and defining such semantics, and the notion of informativeness that comes with it, is the first contribution of this paper.
1996); hence we use this model. Incomplete databases are populated by two types of elements: \textit{constants} and \textit{nulls}, that come from countably infinite sets denoted by \text{Const} and Null, respectively. We use the symbol $\bot$ to denote nulls.

A relational schema (vocabulary) is a set of relation names with associated arities. To each $k$-ary relation symbol $S$ from the vocabulary, an incomplete relational instance (database) $D$ assigns a $k$-ary relation $S^D$ over $\text{Const} \cup \text{Null}$, that is, a finite subset of $(\text{Const} \cup \text{Null})^k$. When the instance is clear from the context we shall write $S$, rather than $S^D$, for the relation itself as well. The sets of constants and nulls that occur in $D$ are denoted by $\text{Const}(D)$ and $\text{Null}(D)$, respectively. If $\text{Null}(D)$ is empty, we refer to $D$ as complete; that is, complete databases are those without nulls. The active domain of $D$ is $\text{dom}(D) = \text{Const}(D) \cup \text{Null}(D)$.

A \textit{valuation} $v$: $\text{Null}(D) \to \text{Const}$ is a map that assigns a constant value to each null in a database. The notion naturally extends to databases, by replacing each $\bot$ with $v(\bot)$ in every place where $\bot$ occurs. We denote the resulting database by $v(D)$, and use valuations to define the semantics of incomplete databases:

$$\llbracket D \rrbracket = \{ v(D) \mid v \text{ is a valuation} \}.$$ 

Intuitively, $\llbracket D \rrbracket$ is the set of possible complete databases that $D$ can represent.

Note that (2) defines a semantics where databases are not open to adding new facts. By analogy with the \textit{closed-world} assumption (Reiter 1977), it is called the CWA semantics in database literature (Imielinski and Lipski 1984). It differs from the open-world (OWA) semantics; here we just restore the missing information already present in the database (whence “closedness”). In the study of incompleteness in databases, the closed-world semantics is more common (Abiteboul, Hull, and Vianu 1995; Abiteboul, Kanellakis, and Grahe 1991; Imielinski and Lipski 1984); it is better behaved, as query answering under the open-world semantics is undecidable for relational calculus queries even in data complexity. Various applications mix OWA and CWA semantics; for instance, both occur in integration and exchange applications, and while ontology-based data access tends to be efficient approximation procedures.

First-Order Logic (FO) As our basic query language we consider FO (often referred to as relational calculus in the database context), whose formulae include relational atoms $R(\bar{x})$, equality atoms $x = y$ and are closed under conjunction $\land$, disjunction $\lor$, negation $\neg$, existential quantifiers $\exists$ and universal quantifiers $\forall$. We write $\varphi(\bar{x})$ to indicate that $\bar{x}$ is the list of free variables of formula $\varphi$, and we write $|\bar{x}|$ for the length of $\bar{x}$. For complete databases, the notion of $D \models \varphi(\bar{a})$, where $\bar{a}$ is a tuple of elements of $\text{dom}(D)$ interpreting $\bar{x}$, is defined in the standard way. The result of the query is then the set of all tuples $\bar{a}$ over $\text{dom}(D)$ such that $D \models \varphi(\bar{a})$. If $|\bar{x}| = k$, we speak of a $k$-ary query. Over incomplete databases, there are many alternative ways of defining semantics that will be discussed in Section 3.

Certain Answers Traditionally, certain answers to a query $Q$ on an incomplete database $D$ are defined as $\Box(Q, D) = \bigcap \{ Q(D') \mid D' \in \llbracket D \rrbracket \}$. This definition is restrictive in a variety of ways (Libkin 2016a); in particular it does not allow for tuples with nulls. The concrete semantics of query answering considered here will use a slight modification (Lipski 1984) that overcomes this deficiency. Note that $\Box(Q, D)$ consists of tuples $\bar{u}$ over $\text{Const}$ such that, for every valuation $v$, the tuple $v(\bar{u})$ is in $Q(v(D))$. The notion of \textit{certain answers with nulls} (borrowing the name from Libkin 2016b) simply drops the requirement that tuples be constant. For a $k$-ary query $Q$, they are defined as

$$\Box_\bot(Q, D) = \{ \bar{u} \in \text{dom}(D)^k \mid v(\bar{u}) \in Q(v(D)) \} \text{ for every valuation } v \}.$$ 

In particular, $\Box_\bot(Q, D)$ is the set of constant tuples in $\Box_\bot(Q, D)$.

Many-Valued Logics A many-valued logic $\mathcal{L}$ is given by its set of truth values $T = \{ \tau_0, \ldots, \tau_n \}$, a set $\Omega$ of propositional connectives (e.g., $\land$, $\lor$, $\neg$) with their arities, and an interpretation of each $k$-ary connective $\omega$ as a map $\omega: T^k \to T$. We usually omit $\Omega$ when it is clear from the context. We shall use the convention here that $\tau$ and $\sigma$ range over truth values of many-valued logics.

Logics come equipped with a \textit{knowledge ordering} $\preceq$ on their values, indicating when a truth value carries more information (Belnap 1977; Ginsberg 1988). We shall assume that the value $\tau_0$ is the least among those in $T$, indicating “no information whatsoever”, that is, $\tau_0 \preceq \tau_i$ for every $i \leq n$.

For the 3VL used by relational DBMS (also known as the Kleene logic), its truth tables and knowledge ordering are:

<table>
<thead>
<tr>
<th>$\neg$</th>
<th>$\land$</th>
<th>$\lor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_0$</td>
<td>$\tau_0$</td>
<td>$\tau_0$</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>$\tau_0$</td>
<td>$\tau_1$</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>$\tau_1$</td>
<td>$\tau_1$</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>$\tau_2$</td>
<td>$\tau_1$</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>$\tau_2$</td>
<td>$\tau_2$</td>
</tr>
<tr>
<td>$\tau_5$</td>
<td>$\tau_2$</td>
<td>$\tau_2$</td>
</tr>
<tr>
<td>$\tau_6$</td>
<td>$\tau_2$</td>
<td>$\tau_2$</td>
</tr>
<tr>
<td>$\tau_7$</td>
<td>$\tau_2$</td>
<td>$\tau_2$</td>
</tr>
<tr>
<td>$\tau_8$</td>
<td>$\tau_2$</td>
<td>$\tau_2$</td>
</tr>
<tr>
<td>$\tau_9$</td>
<td>$\tau_2$</td>
<td>$\tau_2$</td>
</tr>
</tbody>
</table>

| $\tau_0$ | $\tau_1$ | $\tau_2$ | $\tau_3$ | $\tau_4$ | $\tau_5$ | $\tau_6$ | $\tau_7$ | $\tau_8$ | $\tau_9$ |

\[ f \]

\[ u \]

\[ v \]

\[ \tau \]

\[ \sigma \]
In terms of knowledge, rather than truth, \( t \) and \( f \) are incomparable, but \( u \) gives us less knowledge than both \( t \) and \( f \). The crucial property of the knowledge ordering \( \preceq \) is that the connectives of the logic are monotone with respect to it; indeed, the more knowledge we have about the input, the more we know about the output. It is easy to check that \( \land, \lor, \neg \) above are monotone with respect to \( \preceq \).

Information Orderings A key concept in defining the notion of certainty, and in providing the correct semantics of queries and their answers, is the notion of information ordering on databases (Libkin 2016a). This, given a semantics of incomplete databases, allows us to compare them in terms of their information content. The idea is that the larger the semantics of an incomplete database, the less informative it is. Indeed, the more an incomplete database can potentially denote, the less we know about it: if we have full knowledge, the database can denote just itself, if we have no knowledge whatsoever, it can denote anything.

For input databases, the information ordering \( \preceq \) is given by

\[
D \preceq D' \iff \llbracket D' \rrbracket \subseteq \llbracket D \rrbracket.
\]

and in general this notion can be used with other semantics of incompleteness as well. For the cwa semantics \( \llbracket \cdot \rrbracket \), it is known (Gheerbrant, Libkin, and Sirangelo 2014) that \( D \preceq D' \) iff there is a homomorphism \( h \) from \( D \) to \( D' \) such that \( h(D) = D' \), which is called a strong onto homomorphism.

3 Many-Valued Query Answering

Let us come back to the example from the introduction: the query \( \varphi(x) = R(x) \land \neg S(x) \) on the database \( D \) with \( R = \{1, 2\} \) and \( S = \{\bot, 2\} \). The answer annotates each element with a truth value. We assume that the correct annotation should assign \( u \) to 1 and \( f \) to 2, although we saw that under the semantics of SQL negation, 1 is incorrectly assigned \( t \).

This tells us that a many-valued answer is more than just a set of tuples: it assigns truth values from \( T \) to tuples. The semantics of a query should tell us how to assign these values correctly, but a particular evaluation mechanism need not necessarily coincide with the semantics (in fact this is rarely the case, as the correct semantics may be too expensive to compute).

This suggests that we have to clearly distinguish three distinct concepts related to queries:

- their syntax \( \varphi \) (in this paper, the standard FO formulæ);
- their semantics, which will be denoted by \( \text{Sem}(\varphi) \); and
- their evaluations, which will be denoted by \( \text{Eval}(\varphi) \).

The main questions we need to address are:

1) How can we say when a semantics \( \text{Sem}(\varphi) \) is correct?
2) When do we know that an evaluation \( \text{Eval}(\varphi) \) is appropriate for \( \text{Sem}(\varphi) \)?
3) How can we compare different evaluations?

We now proceed to define a framework in which we can answer these questions.

Many-Valued Query Answers With a two-valued logic, each tuple is classified into either true or false: we thus think of a query returning the set of tuples on which it evaluates to true, and the other set is its complement. With many-valued semantics, tuples can be annotated with different truth values of a logic \( L \) (even if only the set corresponding to \( t \) is given to the user). This is captured by the following definition.

**Definition 1.** For a many-valued logic \( L \) with truth values \( T \), a \( k \)-ary query answer is a map \( \kappa \) that assigns each truth value \( \tau \in T \) to a set \( \kappa^\tau \) of \( k \)-ary tuples over \( \text{Const} \) and Null. Such a query answer is complete if no nulls occur in tuples. We also write \( A^\tau \) for \( \bigcup \{ A^\sigma \mid \sigma \supseteq \tau \} \), i.e., the set of tuples corresponding to truth values at least as informative as \( \tau \).

Note that we do not insist on the components of the answer being disjoint. Although in many concrete semantics they will be, there are cases when they are not, for instance in many-valued logics having truth values that are combinations of others, such as inconsistency, which means both \( t \) and \( f \) at the same time (Ginsberg 1988). Also, from the technical point of view all the results are true without such a restriction, and hence we do not impose it.

Semantics and Ordering Our next goal is to define the semantics \( \langle A \rangle \) of query answers, and the information ordering \( \preceq \) associated with it. Note that query answers themselves are incomplete databases, and thus we need to understand how to compare them in terms of their information content.

For this, we again appeal to example (1) from the introduction. If we deal with 3VL, then tuples are annotated with \( t, f \) or \( u \). One possibility is \( \{ t \mapsto \varnothing, f \mapsto \varnothing, u \mapsto \{1\} \} \). It provides rather little information, just that the value 1 may be considered as an answer, but we do not know whether it is in fact the answer. A slightly better option is \( \{ t \mapsto \varnothing, f \mapsto \varnothing, u \mapsto \{1, \bot\} \} \), which in addition says that some currently unknown value is a potential answer. A still better one is \( \{ t \mapsto \varnothing, f \mapsto \varnothing, u \mapsto \{1, 2\} \} \), which says that a concrete value 2 is a potential answer. And an even more informative one is \( \{ t \mapsto \varnothing, f \mapsto \{2\}, u \mapsto \{1\} \} \); value 2 is no longer annotated with \( u \), but rather with a more informative value \( f \), saying that 2 is never in the difference of \( R \) and \( S \).

This suggests that the semantics \( \langle A \rangle \) of a \( k \)-ary query answer \( A \) is a set of complete answers \( C \) that can improve our knowledge of \( A \) in three possible ways:

1) some nulls in \( \bar{u} \in A^\tau \) can be replaced with constants by means of a valuation \( v \), and \( v(\bar{u}) \) can be placed in \( C^\tau \);
2) we can gain knowledge about a tuple in \( C^\tau \) and move it to \( C^\sigma \) if \( \tau \preceq \sigma \); and
3) we can add a new tuple to any of the \( C^\tau \)’s, which reflects the fact that we approximate certain answers, hence by adding tuples we gain additional information.

All of the above ways of improving our knowledge of \( A \) are captured by the following semantics of \( k \)-ary query answers.

\[
\langle A \rangle = \{ \text{complete answer } C \mid \exists \text{ valuation } v \text{ such that } \forall \sigma \preceq \tau \{ v(A^\tau) \subseteq C^\tau \text{ for each } \tau \in T \} \}.
\]

Every semantics of incompleteness gives us an information ordering (see Section 2). For input databases, the order-
ing $D \sqsubseteq D'$ was characterized by the existence of a homomorphism $h$ from $D$ to $D'$ such that $h(D) = D'$. For query outputs, the ordering is denoted by $\sqsubseteq$, and it is given by

$$A \sqsubseteq A' \iff \langle A \rangle \subseteq \langle A' \rangle.$$  

The ordering $\sqsubseteq$ can be characterized as follows.

**Proposition 1.** For $k$-ary query answers, we have $A \sqsubseteq B$ if there exists a mapping $h$: Null $\rightarrow$ Const $\cup$ Null such that $h(A^k) \subseteq B^k$ for every truth value $\tau$.

This is an analog of the existence of a homomorphism from $A$ to $B$, except that the image of a tuple in $A^k$ may appear not only in $B^k$ but also in any $B^\sigma$ for $\sigma \supsetneq \tau$. When the statement of Proposition 1 holds, we say that $A \sqsubseteq B$ is witnessed by $h$.

**Query Semantics and Evaluations**

We now use the notions of query answers and information orderings to explain what semantics and evaluation are.

**Definition 2.** A query semantics $\phi$ for a many-valued logic $\mathcal{L}$ with truth values $\mathcal{T}$ is a function $\text{Sem}$ that, with a syntactic query $\varphi(x)$ where $|x| = k$, and a database $D$, associates a $k$-ary query answer $\text{Sem}(\varphi, D)$ such that:

1. the union of all $\text{Sem}^\tau(\varphi, D)$ is $\text{adam}(D)^k$,
2. if $D \sqsubseteq D'$, then $\text{Sem}(\varphi, D) \subseteq \text{Sem}(\varphi, D')$.

The first condition simply says that every tuple is accounted for, i.e., assigned at least one truth value by the semantic function. The second condition is the preservation of informativeness (Libkin 2016a): if we know more about the input, we know more about the output.

**Definition 3.** A $\mathcal{T}$-valued evaluation procedure $\text{Eval}$ takes a syntactic query $\varphi(x)$ with $|x| = k$, a database $D$ and an assignment $\bar{a}$ of elements of $\text{adam}(D)$ to $\bar{x}$, and returns a truth value $\text{Eval}(\varphi, D, \bar{a}) \in \mathcal{T}$. The $k$-ary query answer produced by $\text{Eval}$ on $\varphi$ and $D$ is denoted by $\text{Eval}(\varphi, D)$, where

$$\text{Eval}(\varphi, D) = \{\bar{a} : \bar{a} \in \text{adam}(D)^k \mid \text{Eval}(\varphi, D, \bar{a}) = \tau\}$$

is the set of tuples on which $\text{Eval}$ returns $\tau$.

We remark that, for any two distinct truth values $\tau$ and $\sigma$, $\text{Sem}^\tau(\varphi, D)$ and $\text{Sem}^\sigma(\varphi, D)$ may have tuples in common, whereas $\text{Eval}(\varphi, D)$ and $\text{Eval}(\varphi, D)$ are always disjoint, as an evaluation procedure assigns precisely one truth value to each tuple.

Most commonly, the evaluation output given to the user is simply $\text{Eval}(\varphi, D)$, serving as an approximation of certain answers. As we saw, this is often refined with information about other truth values. Moreover, other sets often need to be computed even if the goal is to just return $\text{Eval}(\varphi, D)$.

Information orderings on query answers suggest what can be considered as potential evaluations for a given semantics. Indeed, the least condition we should impose on evaluations is that they cannot produce more information than the semantics gives us; that is, $\text{Eval}(\varphi, D) \subseteq \text{Sem}(\varphi, D)$ for every query $\varphi$ and every database $D$.

This is very permissive however; for instance, evaluations satisfying this condition may unnecessarily identify nulls with constants or with each other. As a simple example, consider the ordering $f \preceq t$, and a query $\varphi(x)$ so that on a database with $\text{adam}(D) = \{\bot\}$ its semantics assigns $f$ to $\bot$ and $t$ to $1$. Now suppose an evaluation does the opposite, that is, it assigns $f$ to 1 and $t$ to $\bot$. Intuitively this is wrong, as the evaluation somehow knows more about the null $\bot$ than the semantics does. However, the ordering allows this, since $\text{Eval}(\varphi, D) \subseteq \text{Sem}(\varphi, D)$ is witnessed by a map $h$ that sends $\bot$ to 1.

This suggests getting rid of arbitrary maps witnessing $\text{Eval}(\varphi, D) \subseteq \text{Sem}(\varphi, D)$ and only permitting identity; in this case, evaluation simply distributes tuples among sets corresponding to different truth values.

**Definition 4.** We say that $\text{Eval}$ is an evaluation for semantics $\text{Sem}$ if $\text{Eval}(\varphi, D) \subseteq \text{Sem}(\varphi, D)$ is witnessed by the identity mapping, that is, $\text{Eval}^\tau(\varphi, D) \subseteq \text{Sem}^\tau(\varphi, D)$ for every query $\varphi$, database $D$, and truth value $\tau$.

Thus, if an evaluation $\text{Eval}$ assigns truth value $\tau$ to a tuple, then the semantics must assign a value $\sigma$ that is at least as informative as $\tau$. The following is an immediate consequence of the definition.

**Proposition 2.** Let $\text{Eval}$ be an evaluation for $\text{Sem}$. If $\tau$ is a maximal truth value with respect to $\preceq$, then $\text{Eval}^\tau(\varphi, D) \subseteq \text{Sem}^\tau(\varphi, D)$.

In most of the cases naturally arising in handling incompleteness, the semantics of truth value $\mathbf{t}$ is the set of certain answers (with nulls), and $\mathbf{t}$ is a maximal truth value with respect to $\preceq$. Thus, in this case, the proposition implies that in an evaluation the set of tuples assigned $\mathbf{t}$ gives us an approximation of certain answers.

It would be desirable to extend Proposition 2 to other truth values: if $\text{Eval}$ assigns $\tau$ to a tuple $\bar{a}$, then the semantics does the same (which is so far guaranteed only for the maximal truth values). However, such a requirement is too strong: if $\text{Sem}$ assigns each tuple $\bar{a}$ to only one truth value, this simply means that $\text{Eval}$ and $\text{Sem}$ coincide, which usually precludes us from having a computationally efficient $\text{Eval}$. Ideally, we want $\text{Eval}$ to deviate from $\text{Sem}$ as little as possible. Thus, we only allow them to be different when $\text{Eval}$ cannot infer any information about a tuple and so assigns it the least truth value $\tau_0$ (in the concrete semantics that we use, it will be the unknown $\mathbf{u}$).

**Definition 5.** For a query $\varphi$, an evaluation procedure $\text{Eval}$

- has correctness guarantees for semantics $\text{Sem}$ if, for every database $D$ and for every truth value $\tau \neq \tau_0$, we have $\text{Eval}(\varphi, D) \subseteq \text{Sem}(\varphi, D)$;
- preserves informativeness if $\text{Eval}(\varphi, D) \subseteq \text{Eval}(\varphi, D')$ whenever $D \sqsubseteq D'$.

In the next section we give a recipe for obtaining evaluation procedures with correctness guarantees, and then study it for some concrete logics.

### 4 Correct and Efficient Evaluation

The previous section gave us a general framework for reasoning about many-valued query answers, as well as query evaluations and semantics. We now demonstrate that, in this
framework, it is easy to generate evaluation procedures with correct¬
ness guarantees. In fact, all we need to do is define them for atomic formulae and then lift to all of FO

We always assume that the set Ω or propositional con¬
nectives of a many-valued logic L includes the connectives ∧
and ∨; the only condition on their interpretation is that they
are associative so that we can unambiguously define ∧
and ∨ T when T is a tuple of truth values.

By FO_L we mean the closure of atomic formulae R(\bar{x})
and x = y under propositional connectives from Ω and
quantifiers ∀, ∃. If Ω = {∧, ∨, ¬} this is the usual first-order
logic, but in general we can be more permissive about propo¬
sitional connectives.

We look at procedures that lift evaluations of atoms fol¬
lowing the syntax of formulae;¹ in fact standard query eval¬
uation procedures are such (even if they optimize a query
first, then they follow the syntax). This is captured by the
following definition.

**Definition 6.** An evaluation procedure Eval for FO_L is
called syntactic if it follows the rules below. For proposi¬
tional connectives, we have

\[
\text{Eval}(\omega(\varphi_1(x_1), \ldots, \varphi_m(x_m)), D, \bar{a}) = \\
\omega_L \left( \text{Eval}(\varphi_1(x_1), D, \bar{a}_1), \ldots, \text{Eval}(\varphi_m(x_m), D, \bar{a}_m) \right)
\]

where \( \bar{a}_i \) is the subtuple of \( \bar{a} \) corresponding to the variables
in \( \bar{x}_i \). For quantifiers, we have

\[
\text{Eval}(\exists y \varphi(x, y), D, \bar{a}) = \bigvee_{a \in \text{dom}(D)} \text{Eval}(\varphi(x, y), D, (\bar{a}, a)),
\]

\[
\text{Eval}(\forall y \varphi(x, y), D, \bar{a}) = \bigwedge_{a \in \text{dom}(D)} \text{Eval}(\varphi(x, y), D, (\bar{a}, a)).
\]

In other words, for a syntactic procedure, we need to de¬
fine rules for atoms \( R(\bar{x}) \) and \( x = y \), and then lift to all of
FO_L using the above rules. Our goal is then to show that
correctness of such procedures can also be lifted from atoms
to all of FO_L. For this, we need some mild conditions on the
semantics.

First, we assume that each formula \( \varphi(\bar{x}) \) can be viewed
also as a formula \( \varphi(\bar{y}) \), as long as all variables in \( \bar{x} \) occur in
\( \bar{y} \), in some order. For instance, a formula \( \varphi(x, y) \) with output
\( X \) (which is a set of pairs) can also be viewed as a formula
\( \varphi(y, x, z) \) with the output \{ (b, a, c) \mid (a, b) \in X, c \in \text{dom}(D) \}. In general, \( \text{Sem}^\tau(\varphi(\bar{y}), D) \) is defined as the set
of all tuples \( \bar{a} \in \text{dom}(D) \) such that the subtuple of \( \bar{a} \) corresponding to the positions of \( \bar{x} \) belongs to \( \text{Sem}^\tau(\varphi(\bar{x}), D) \). This is essentially a free condition; all natural logic-based
semantics will be such, and thus we assume it always holds.

The main condition is the compatibility of Sem with L. To
explain it informally, suppose that a tuple \( \bar{u} \) belongs to
\( \text{Sem}^\tau(\varphi, D) \) and \( \text{Sem}^\tau(\psi, D) \), where \( \tau, \sigma \), and \( \tau \land \sigma \) are truth
values carrying some information, i.e., different from
\( \tau_0 \). Then we would expect \( \bar{u} \) to be in the set corresponding
to \( \tau \land \sigma \) in \( \text{Sem}(\varphi \land \psi, D) \), if the semantics reflects the meaning of the connectives. Note that \( \tau_0 \) is essentially the

¹In some contexts, such procedures are called compositional.
weak idempotency, saying that there is \( k > 0 \) so that iterating the operation \( k + 1 \) times is the same as iterating \( k \) times (for \( k = 1 \), this is the usual idempotency \( \tau * \tau = \tau \), when \( * \) is either \( \land \) or \( \lor \)).

### 5 Applications: 3VL, 4VL, and 2VL

We now demonstrate the applicability of the framework developed in the previous section. To begin with, we show that the main result of Libkin (2016b) – an efficient and correct 3VL evaluation procedure that mends some of the most glaring issues with SQL’s evaluation of nulls – is an immediate application of the framework for 3VL. As a bonus, we show that the evaluation procedure also preserves informativeness. We then provide a refinement, already hinted at in the introduction, by showing how to classify some of the unknown tuples in a way that generates additional useful information (for instance, ruling them out as certain answers). We finally show that certainty guarantees are impossible with the usual Boolean 2-valued logic, thus justifying the decision by the designers of practical query languages (SQL) to use 3VL.

#### 3VL

We start with the usual truth values \( t, f, u \). For a tuple \( \bar{a} \), an incomplete database \( D \), and a \( k \)-ary query \( \varphi(x) \), \( t \) means that \( \bar{a} \) is in the answer for all \( D' \in [D] \) and \( f \) means that \( \bar{a} \) is not in the answer for all \( D' \in [D] \). Thus, we define the three-valued semantics \( \text{Sem}_{3v}^D(\varphi, D) \) as follows:

- \( \text{Sem}_{3v}^D(\varphi, D) = \square_\bot(\varphi, D) \) ,
- \( \text{Sem}_{3v}^D(\varphi, D) = \Box_\bot(\neg \varphi, D) \) ,
- \( \text{Sem}_{3v}^D(\varphi, D) = \text{adom}(D) \neg \left( \square_\bot(\varphi, D) \cup \Box_\bot(\neg \varphi, D) \right) \) .

That is, the set corresponding to \( t \) is the set of tuples which are certainly true, the set of \( f \) contains tuples which are certainly false, and other tuples are classified as unknown.

Note that one still needs to make sure that this is a semantics in the sense of Definition 2, that is, it classifies all the tuples in \( \text{adom}(D) \) and it returns more informative results on more informative tuples. We can show this, as well as the fact that this semantics is compatible with 3VL in the sense of Definition 7.

**Proposition 4.** \( \text{Sem}_{3v} \) is a query semantics that is compatible with 3VL.

Therefore, by Theorem 1, we only need to provide a correct evaluation procedure \( \text{Eval}_{3v} \), for atomic formulæ \( R(\bar{x}) \) and \( x = y \), and then lift to full FO using the rules for syntactic evaluation procedures. Recall the notion of correctness: for each truth value \( \tau \in \{ t, f \} \), if \( \text{Eval}(\varphi(\bar{x}), D, \bar{a}) = \tau \), then \( \bar{a} \in \text{Sem}_{3v}^D(\varphi, D) \). Combining this with the definition of \( \square_\bot \), we derive evaluations for atomic formulæ.

- If \( \text{Eval}_{3v}(R(\bar{x}), D, \bar{a}) = t \), then \( v(\bar{a}) \in v(R^D) \) for every valuation \( v \), so we must require \( \bar{a} \in R^D \).
- If \( \text{Eval}_{3v}(R(\bar{x}), D, \bar{a}) = f \), then \( v(\bar{a}) \notin v(R^D) \) for every valuation \( v \), so we make it the condition for \( \text{Eval}_{3v} \) returning false.

#### 4VL

The second of the above conditions can be expressed with the notion of unifiable tuples: these are tuples \( \bar{a} \) and \( \bar{b} \) of the same length such that \( v(\bar{a}) = v(\bar{b}) \) for some valuation \( v \). We then write \( \bar{a} \sqsupseteq \bar{b} \). This condition can be tested in linear time (Paterson and Wegman 1978).

All of the above tells us that the evaluation for atoms is:

\[
\text{Eval}_{3v}(R(\bar{x}), D, \bar{a}) = \begin{cases} 
    t & \text{if } \exists b \in R^D \text{ such that } \bar{a} \sqsupseteq \bar{b} \\
    f & \text{if } a \neq b \text{ and } a,b \in \text{Const} \\
    u & \text{otherwise} 
\end{cases}
\]

Extended to all of FO by the rules of Definition 6, this is exactly the procedure proposed by Libkin (2016b). Since in 3VL binary connectives are idempotent, the main result of that paper is an immediate corollary of our framework:

**Corollary 1.** \( \text{Eval}_{3v} \) has correctness guarantees with respect to the three-valued semantics, and \( AC^0 \) data complexity.

In addition, we can show that the procedure \( \text{Eval}_{3v} \) returns more informative results on more informative inputs:

**Proposition 5.** \( \text{Eval}_{3v} \) preserves informativeness.

#### 2VL

We will now show how to refine the notion of unknown (\( u \)) to provide additional information with query answers. So far the appearance of \( u \) in query evaluation accounts for two distinct possibilities. One is that a tuple is classified as \( t \) or \( f \) by the semantics, but the evaluation procedure – which approximates the semantics – does not see this. The second possibility is that it is actually known that a tuple cannot be classified as either \( t \) or \( f \) by the semantics, as it may, or may not, be in the answer depending on the interpretation of incomplete information.
We now separate the two by adding a new truth value \( s \) (sometimes), which means that sometimes a tuple is in the query result, and sometimes it is not. We define a 4-valued semantics \( \text{Sem}_4 \) with sets corresponding to \( t, f, s, u \). The sets \( \text{Sem}_4^t(\varphi, D) \) and \( \text{Sem}_4^s(\varphi, D) \) are defined in the same way as for 3VL, i.e., they are \( \square_\perp(\varphi, D) \) and \( \square_\perp(\neg \varphi, D) \), respectively; \( \text{Sem}_4^u(\varphi, D) \) is defined as
\[
\{ \bar{a} \in \text{dom}(D)^k \mid \exists \text{valuations } v, v' \text{ such that } v(D) = \varphi(v(\bar{a})), v'(D) \neq \varphi(v'(\bar{a})) \}
\]
and \( \text{Sem}_4^u(\varphi, D) = \emptyset \) (i.e., tuples can be annotated with \( u \) only by evaluation procedures). If a tuple is in \( \text{Sem}_4^u(\varphi, D) \), we can conclude that there is a valuation \( v \) such that \( v(D) = \varphi \lor \psi(v(\bar{a})) \), but we cannot conclude that \( v'(D) \neq \varphi \lor \psi(v'(\bar{a})) \) for some \( v' \). Likewise, we have \( s \land s = u \).

To explain the ordering of \( s, t, \) and \( f \), assume that we get a value \( s \) for some tuple \( \bar{a} \) on a database \( D \); it means that \( [D] \) contains databases on which \( \bar{a} \) is in the answer as well as databases on which it is not. If we now have a more informative database \( D' \), that is, \( D \subseteq D' \), then \( [D'] \subseteq [D] \), and we have three possibilities: either \( \bar{a} \) is in the answer on all databases in \( [D'] \) (value \( t \)), or it is not in the answer on all such databases (value \( f \)), or it is on some and not on others (value \( s \)). Hence, by going to a more informative database, \( s \) either stays, or improves to \( t \) or \( f \). The latter in turn give us complete knowledge: going to a more informative database does not affect the answer. This leads to the ordering shown in Figure 1.

As for 3VL, we have defined a proper semantics (it preserves informativeness) that is compatible with the logic.

**Proposition 6.** \( \text{Sem}_4 \) is a query semantics that is compatible with 4VL, and the operations \( \land, \lor, \) and \( \neg \) are monotone with respect to \( \preceq \) in 4VL.

Thus, by Theorem 1, to get a four-valued evaluation procedure \( \text{Eval}_4 \), we need to define it for atomic formulae and lift by the rules of Definition 6. The evaluation of \( R(x) \) and \( x = y \) in \( \text{Eval}_4 \) is the same as in \( \text{Eval}_3 \), with just one difference: \( u \) is replaced by \( s \).

The correctness of \( \text{Eval}_4 \) can be easily shown. Moreover, while the connectives \( \land \) and \( \lor \) are idempotent (\( s \land s = s \land s = u \), and this is how \( u \) can be produced by \( \text{Eval}_4 \), they are weakly idempotent: \( s \lor s = s \lor s = s \), and likewise for \( \land \).

Therefore, we have:

**Theorem 3.** \( \text{Eval}_4 \) has correctness guarantees w.r.t. \( \text{Sem}_4 \), and \( AC^0 \) data complexity. Furthermore, \( \text{Eval}_4 \) preserves informativeness for all FO queries.

Let us now return to the example (1) from the introduction. The procedure \( \text{Eval}_3 \) will assign \( u \) to 1 and \( \perp \), and \( f \) to 2, in line with the fact that \( \square_\perp(\varphi, D) = \emptyset \), and refining this information by stating that 2 belongs to \( \square_\perp(\neg \varphi, D) \).

The procedure \( \text{Eval}_4 \) does leave open the possibility that 1 could be a certain true or a certainly false answer, as it assigns \( u \) in an attempt to provide an approximation of certain answers in a computationally efficient way. The evaluation procedure \( \text{Eval}_4 \) further refines this by assigning \( s \) to 1, thus confirming that it cannot be a certain answer.

**Revisiting Boolean Logic**

We were able to obtain many-valued evaluation procedures with ease. Can we perhaps adapt the techniques so as to get an evaluation procedure based on Boolean logic for approximating certain answers? To the SQL designers, it appeared that they could not, hence they used 3VL. Now we formally prove that they were right.

We say that \( \text{Eval} \) is a Boolean evaluation if it assigns values \( t, f \) to formulae, and satisfies the following rules:
\[
\text{Eval}(\varphi \land \psi, D, a) = \text{Eval}(\varphi, D, a) \land \text{Eval}(\psi, D, a),
\]
\[
\text{Eval}(\varphi \lor \psi, D, a) = \text{Eval}(\varphi, D, a) \lor \text{Eval}(\psi, D, a),
\]
\[
\text{Eval}(\neg \varphi, D, a) = \neg \text{Eval}(\varphi, D, a),
\]
where \( \land, \lor \) and \( \neg \) are the usual Boolean connectives interpreted on \( t, f \). In other words, we give ourselves complete freedom in defining the evaluation for atomic formulae and quantifiers, but insist on respecting the usual Boolean logic.

A certain answers semantics \( \text{Sem} \) is any semantics such that \( \text{Sem}(\varphi, D) \subseteq \square_\perp(\varphi, D) \). Then, we have the following impossibility result.

**Theorem 4.** No certain answers semantics admits a Boolean evaluation.

In other words, if \( \text{Eval} \) respects the Boolean semantics of \( \land, \lor, \neg \), then we cannot have \( \text{Eval}(\varphi, D) \subseteq \square_\perp(\varphi, D) \) for all FO formulae \( \varphi \) and all databases \( D \), that is, \( \text{Eval} \) will sometimes return false positives.

To explain why this is the case, suppose that we can find a Boolean evaluation \( \text{Eval} \) with \( \text{Eval}(\varphi, D) \subseteq \square_\perp(\varphi, D) \). Take \( D = \{ R(a), S(\perp) \} \); if \( \text{Eval}(R(x), D, a) = t \), then \( \text{Eval}(\neg R(x), D, a) = t \), but \( a \notin \square_\perp(\neg R(x), D) \) and hence \( \text{Eval}(R(x), D, a) = t \). Next, if \( \text{Eval}(\neg S(x), D, a) = f \), then \( \text{Eval}(S(x), D, a) = t \). But \( a \notin \square_\perp(S(x), D) \), which implies \( \text{Eval}(\neg S(x), D, a) = t \). Now consider \( \varphi(x) = R(x) \land \neg S(x) \); from the above, we have \( \text{Eval}(\varphi, D, a) = t \), but it is easily seen that \( a \notin \square_\perp(\varphi, D) \), since a complete database where \( \perp \) is interpreted as \( a \) falsifies \( \varphi(a) \). This contradiction shows that a Boolean evaluation for a certain answers semantics cannot exist.

### 6 Comparison and Optimality of Evaluations

The notion of information ordering \( \odot \) on many-valued outputs allows us to compare different evaluation procedures: those returning more informative outputs are preferable. Given two evaluation procedures \( \text{Eval} \) and \( \text{Eval}' \), we say that \( \text{Eval} \) is more informative if \( \text{Eval}(\varphi, D) \subseteq \text{Eval}(\varphi, D) \) for every formula \( \varphi \) and database \( D \).

In fact we can use this notion even if \( \text{Eval} \) and \( \text{Eval}' \) use different sets of truth values, as long as the orderings on the common truth values are the same. Indeed, in such a case,
we simply assume that the set of tuples corresponding to a missing truth value is empty. This allows us to compare 3VL and 4VL evaluations.

Our first result shows that the 4VL evaluation properly refines 3VL, that is, it adds information.

**Proposition 7.** For every FO query \( \varphi \) and every database \( D \), we have \( \text{Eval}^3_\varphi(\varphi, D) = \text{Eval}^4_\varphi(\varphi, D) \) if \( t \) or \( \mathbf{f} \). Furthermore, \( \text{Eval}^4_\varphi(\varphi, D) \) is disjoint from both \( \square_\bot(\varphi, D) \) and \( \square_\bot(\neg \varphi, D) \). Consequently, \( \text{Eval}^4_\varphi \) is more informative than \( \text{Eval}^3_\varphi \).

This indicates that if we only care about certain answers (i.e., tuples that evaluate to \( \mathbf{F} \) in the context, be the set of all atomic subformulae \( \bar{\psi} \)).

Moreover, we can show the following:

**Proposition 8.** The evaluation procedure \( \text{Eval}^\inf \) has correctness guarantees with respect to \( \text{Sem}^3_{\varphi} \) and it is more informative than \( \text{Eval}^3_{\varphi} \).

In fact

\[
\text{Eval}^3_{\varphi}(\varphi, D) \subseteq \text{Eval}^\inf(\varphi, D) \subseteq \square_\bot(\varphi, D)
\]

and both inclusions can be proper.

As an example, consider the query \( \varphi(x) = \exists y \left( (R(x,y) \land (y = 1 \lor y \neq 1)) \right) \) and a database \( D \) where \( R^D = \{ (1, \bot) \} \). Then, \( \text{Eval}^\inf(\varphi, D, 1) = \{ \mathbf{t} \} \) and \( \text{Eval}^3_{\varphi}(\varphi, D, 1) = \{ \mathbf{u} \} \). The SQL counterpart of the query \( \varphi \) is \( \text{SELECT R.A FROM R WHERE R.B=1 OR R.B<>1} \), which returns the empty set, whereas the certain answer is \( \{ \mathbf{t} \} \). Here, \( \text{Eval}^3_{\varphi} \) gives the result of the SQL evaluation, while \( \text{Eval}^\inf \) gives us the certain answer.

### 7 Conclusion

The main message of the paper is that certain answers can be efficiently approximated with the help of many-valued logics, and the structure of truth values in the logic lets us obtain more refined information than just certain answers themselves. Our results also bridge the general approach of Libkin (2016a) to defining and computing certain answers with many-valued procedures as practiced by commercial DBMSs. In particular, we have shown how to define the semantics of many-answers, and how to ensure that queries preserve informativeness of databases.

As for future work, we would like to apply these ideas in some of the scenarios where certain answers are the preferred query semantics, and to look at different semantics of incompleteness. For the first direction, natural candidates are applications such as data integration and exchange (Arenas et al. 2014; Lenzerini 2002), where efficient query answering is hard to achieve for full FO, and where constraints make the process even more complicated (Cali, Lembo, and Rosati 2003; Calì et al. 2004). Note that our approach, not being restricted to a fragment of FO, makes it easy to incorporate constraints.

A slightly different direction is to see if our many-valued approach could be used in knowledge-base reasoning tasks where three-valued logic was previously used to generate sound evaluations (Levesque 1998; Liu and Levesque 2003). While technically different, the approaches share the idea of using many-valuedness for efficient approximation, and Liu
and Levesque (2003) further showed how database evaluation techniques could be used for reasoning with knowledge bases.

For other semantics, one obvious candidate is open-world, which is commonly used, but makes computing certain answers harder due to complexity considerations. Still there is hope of going beyond existential positive queries using many-valuedness: for instance, Grahe, Moulemi, and Onet (2015) showed how to use Belnap’s four-valued logic to compute certain answers for conjunctive queries with negated atoms. In the opposite direction, one can look at semantics that are more restrictive; e.g., semantics of uncertainty represented by a finite number of possible worlds, as in the work by Olteanu, Koch, and Antova (2008).

Acknowledgments We thank the anonymous referees and Eugenia Ternovska for their comments. Part of this work was done when the first author visited the University of Edinburgh. Supported by EPSRC grants J015377, M025268.

References


