Handling Inconsistencies in Tables with Nulls and Functional Dependencies

Dominique Laurent, Nicolas Spyratos

Noname manuscript No.
(will be inserted by the editor)

Handling Inconsistencies in Tables with Nulls and Functional Dependencies

Dominique Laurent · Nicolas Spyratos

Received: date / Accepted: date

Abstract In this paper we address the problem of handling inconsistencies in tables with missing values (also called nulls) and functional dependencies. Although the traditional view is that table instances must respect all functional dependencies imposed on them, it is nevertheless relevant to develop theories about how to handle instances that violate some dependencies. Regarding missing values, we make no assumptions on their existence: a missing value exists only if it is inferred from the functional dependencies of the table.

We propose a formal framework in which each tuple of a table is associated with a truth value among the following: true, false, inconsistent or unknown; and we show that our framework can be used to study important problems such as consistent query answering or data quality measures - to mention just two. In this paper, however, we focus mainly on consistent query answering, a problem that has received considerable attention during the last decades.

The main contributions of the paper are the following: (a) we introduce a new approach to handle inconsistencies in a table with nulls and functional dependencies, (b) we give algorithms for computing all true, inconsistent and false tuples, and (c) we give a novel solution to the consistent query answering problem and compare our solution to that of table repairs.

Keywords Inconsistent database . Functional dependency . Null value . Consistent query answering

Dominique Laurent
ETIS Laboratory - ENSEA, CY Cergy Paris University, CNRS
F-95000 Cergy-Pontoise, France
dominique.laurent@u-cergy.fr

Nicolas Spyratos
LISN Laboratory - University Paris-Saclay, CNRS
F-91405 Orsay, France
nicolas.spyratos@lri.fr

Acknowledgment: Work conducted while the second author was visiting at FORTH Institute of Computer Science, Crete, Greece (https://www.ics.forth.gr/)

1 Introduction

In several applications today we encounter tables with missing values and functional dependencies. Such a table is often the result of merging two or more other tables coming from different sources. Typical examples include recording the results of collaborative work, merging of tables during data staging in data warehouses or checking the consistency of a relational database.

As an example of collaborative work consider two groups of researchers each studying three objects found in an archaeological site. The researchers of each group record in a table data regarding the following attributes of each object:

- Identifier (here of the form i_n where n is a natural number, distinct objects being associated with distinct identifiers)
- Kind (such as statue, weapon, ...)
- Material from which the object is made (such as iron, bronze, marble, ...)
- Century in which the object is believed to have been made.

At the end of their work each group submits their findings to the site coordinator in the form of a table as shown in Figure 1 (tables D_1 and D_2). Each row of a table contains data recorded for a single object. For example, the row $(i_1, \mathit{statue}, \mathit{marble}, 1.BC)$ means that object i_1 is a statue made of marble and believed to have been made in the first century before Christ. Similarly the row $(i_2, \mathit{statue}, \ , 2.BC)$ means that object i_2 is a statue of unknown material, believed to have been made during the second century before Christ. Note that, in this tuple, there is a missing value, meaning that the material from which object i_2 is made could not be determined.

D_1	Id	K	M	C
	i_1	k	m	c
	i_1		m'	
	i_2	k'	m'	c
	i_2	k'	m''	
	i_3		m	

D_2	Id	K	M	C
	i_1	k		c
	i_2	k'		c'
	i_2	k'	m''	
	i_3	k'		

D	Id	K	M	C
	i_1	k	m	c
	i_1		m'	
	i_1	k		c
	i_2	k'	m'	c
	i_2	k'	m''	
	i_2	k'		c'
	i_3		m	
	i_3	k'		

Fig. 1 The tables prepared by the two groups and the merged table

Now, the data contained in the two tables can be merged into a single table D containing all tuples from the two tables, without duplicates, as shown in Figure 1 (table D). In doing this merging, we may have discrepancies between tuples of D. For example, object i_1 appears in D as being made from two different materials; and object i_2 appears as made from two different materials and in two different centuries. This kind of discrepancies may lead to 'inconsistencies' that should be identified by the site coordinator and resolved in cooperation with the researchers of the two groups.

It should be obvious from this example that the merging of two or more tables into a single table more often than not results in inconsistencies even if the individual tables are each consistent. For example, although each of the tables D_1 and

D_2 shown in Figure 1 satisfies the functional dependencies $Id \rightarrow K$ and $Id \rightarrow C$, the merged table D does not satisfy $Id \rightarrow C$.

A similar situation arises in data warehouses where one tries to merge views of the underlying sources into a single materialized view to be stored in the data warehouse.

As a last example, in a relational database, although each table may satisfy its functional dependencies, the database as a whole may violate some dependencies. To determine whether the database is consistent with its dependencies, one proceeds as follows: all tables are merged by placing their tuples into a single universal table D possibly with missing values (under certain assumptions discussed in [21]); then all functional dependencies are applied on D through the well known chase algorithm [11,20]. If the algorithm terminates successfully (*i.e.,* no inconsistency is detected) then the database is consistent; otherwise the algorithm stops when a first inconsistency is detected and the database is declared inconsistent.

So in general the question is: what should we do when a table is inconsistent? There are roughly three approaches: (a) reject the table, (b) try to correct or 'repair' it so that to make it consistent (and therefore be able to work with the repaired table) and (c) keep the table as is but make sure you know which part is consistent and which is not.

The first approach is followed by database theorists when checking database consistency, as explained above. This approach is clearly not acceptable in practice as the universal table might contain a consistent set of tuples that can be useful to users (*e.g.,* users can still query the consistent part of the table).

The second approach tries to alleviate the impact of inconsistent data on the answers to a query by introducing the notion of repair: a repair is a minimally different consistent instance of the table and an answer is consistent if it is present in every repair. This approach, referred to as 'consistent query answering', has motivated important research efforts during the past two decades and is still the subject of current research. The reader is referred to Section 5 for a brief overview of the related literature. However, this approach is always difficult to implement due to important issues related to computational complexity and/or to semantics (there is still no consensus regarding the definition of 'consistent answer').

In our work we follow the third approach that is, we keep inconsistencies in the table but we determine which part of the table is consistent and which is not. More specifically, we use set-theoretic semantics for tuples and functional dependencies that allow us to associate each tuple with one truth value among the following: true, false, inconsistent or unknown. By doing so we can study a number of important problems including in particular the problem of consistent query answering, and the definition of data quality measures.

Regarding consistent query answering, our model offers a fundamentally different and direct solution to the problem: the consistent answer is obtained by simply retrieving true tuples that fit the query requirements.

Moreover our approach offers the possibility of defining meaningful data quality measures. For example if a table contains a hundred tuples of which only five are true while the remaining ones are inconsistent, then the quality of data contained in the table is five percent. Since we have polynomial algorithms for computing all true or inconsistent tuples, we can define several quality measures of the data contained in a table, inspired by the work in [15]. We can then use such measures to accompany query answers so that users are informed of the quality of the answer

they receive (e.g., getting an answer from a table with ninety five per cent of true tuples is more reliable than if the table contained only five per cent of true tuples). However, defining and studying such measures lies outside the goals of the present paper. In this paper we focus on one important application of our approach, namely consistent query answering. A complete account of data quality measures will be reported in a future paper.

The main contributions of the present paper can be summarized as follows:

1. We introduce a new approach to handle inconsistencies in a table with nulls and functional dependencies; we do so by adapting the set-theoretic semantics of [18] to our context and by extending the chase algorithm so that all inconsistencies are accounted for in the table.
2. We give polynomial algorithms in the size of the table for computing all true and all inconsistent tuples in the table.
3. We propose a novel approach for consistent query answering and we investigate how our approach relates to existing approaches.

The paper is organized as follows: In Section 2 we recall basic definitions and notations regarding tables and we introduce the set-theoretic semantics that we use in our work. In Section 3 we give definitions and properties regarding the truth values that we associate with tuples. In Section 4 we study computational issues and give algorithms for computing the truth values of tuples. In Section 5 we present a novel solution to the problem of consistent query answering and compare it to existing approaches. Section 6 contains concluding remarks and suggestions for further research.

2 The Model

In this section we present the basic definitions regarding tuples and tables as well as the set-theoretic semantics that we use for tuples and functional dependencies. Our approach builds upon earlier work on the partition model [18].

2.1 The Partition Model Revisited

Following [18], we consider a universe $U = \{A_1, \ldots, A_n\}$ in which every attribute A_i is associated with a set of atomic values called the domain of A_i and denoted by $dom(A_i)$. An element of $\bigcup_{A \in U} dom(A)$ is called a *domain constant* or a *constant*.

We call *relation schema* (or simply *schema*) any nonempty subset of U and we denote it by the concatenation of its elements; for example $\{A_1, A_2\}$ is simply denoted by A_1A_2. Similarly, the union of schemas S_1 and S_2 is denoted as S_1S_2 instead of $S_1 \cup S_2$.

We define a *tuple* t to be a partial function from U to $\bigcup_{A \in U} dom(A)$ such that, for every A in U, if t is defined over A then $t(A)$ belongs to $dom(A)$. The domain of definition of t is called the *schema* of t, denoted by $sch(t)$. We note that tuples in our approach satisfy the *First Normal Form* [20] in the sense that each tuple component is an atomic value from an attribute domain.

Regarding notation, we follow the usual convention that, whenever possible, lower-case characters denote domain constants and upper-case characters denote

the corresponding attributes. Following this convention the schema of a tuple $t = ab$ is AB and more generally, we denote the schema of t as T.

Assuming that the schema of a tuple t is understood, t is denoted by the concatenation of its values, that is: $t = a_{i_1} \dots a_{i_k}$ means that for every $j = 1, \dots, k$, $t(A_{i_j}) = a_{i_j}$, a_{i_j} is in $dom(A_{i_j})$, and $sch(t) = A_{i_1} \dots A_{i_k}$.

We assume that for any distinct attributes A and B, we have either $dom(A) = dom(B)$ or $dom(A) \cap dom(B) = \emptyset$. However, this may lead to ambiguity when two attributes have the same domain. Ambiguity can be avoided by prefixing each value of an attribute domain with the attribute name. For example, if $dom(A) = dom(B)$ we can say 'an A-value a' to mean that a belongs to $dom(A)$, and 'a B-value a' to mean that a belongs to $dom(B)$. In order to keep the notation simple we shall omit prefixes whenever no ambiguity is possible.

Denoting by $\mathcal{T}$ the set of all tuples that can be built up given a universe U and the corresponding attribute domains, a *table* D is a *finite* sub-set of $\mathcal{T}$ where duplicates are *not allowed*.

Given a tuple t, for every A in $sch(t)$, $t(A)$ is also denoted by $t.A$ and more generally, for every subset S of $sch(t)$ the restriction of t to S, also called *sub-tuple* of t, is denoted by $t.S$. In other words, if $S \subseteq sch(t)$, $t.S$ is the tuple such that $sch(t.S) = S$ and for every A in S, $(t.S).A = t.A$.

Moreover, $\sqsubseteq$ denotes the 'sub-tuple' relation, defined over $\mathcal{T}$ as follows: for any tuples t_1 and t_2, $t_1 \sqsubseteq t_2$ holds if t_1 is a sub-tuple of t_2. It is thus important to keep in mind that whenever $t_1 \sqsubseteq t_2$ holds, it is understood that $sch(t_1) \subseteq sch(t_2)$ also holds.

The relation $\sqsubseteq$ is clearly a partial order over $\mathcal{T}$. Given a table D, the set of all sub-tuples of the tuples in D is called the *lower closure* of D and it is defined by: $\mathsf{LoCl}(D) = \{q \in \mathcal{T} \mid (\exists t \in D)(q \sqsubseteq t)\}$. We shall call a table *reduced* if it contains only maximal tuples (*i.e.,* if no tuple in the set is sub-tuple of some other tuple in the set).

The notion of $\mathcal{T}$*-mapping*, as defined below, generalizes that of interpretation defined in [18].

Definition 1 Let U be a universe. A $\mathcal{T}$*-mapping* is a mapping μ defined from $\bigcup_{A \in U} dom(A)$ to $2^{\mathbb{N}}$, where $\mathbb{N}$ is the set of natural numbers.

A $\mathcal{T}$*-mapping* μ is extended to the set $\mathcal{T}$ as follows: for every $t = a_{i_1} \dots a_{i_k}$ in $\mathcal{T}$, $\mu(t) = \mu(a_{i_1}) \cap \dots \cap \mu(a_{i_k})$.

A $\mathcal{T}$-mapping μ is an *interpretation* if μ satisfies the *partition constraint* stating that for every A in U, and for all distinct a and a' in $dom(A)$, $\mu(a) \cap \mu(a') = \emptyset$. □

We emphasize that in [18] interpretations provide the basic tool for defining true tuples: a tuple t is said to be true in an interpretation μ if $\mu(t)$ is nonempty.

To see the intuition behind this definition consider a relational table D over U and suppose that each tuple is associated with a unique identifier, say a natural number. Now, for every A in U and every a in $dom(A)$, define $\mu(a)$ to be the set of all identifiers of the tuples in D containing a. Then μ is an interpretation as it satisfies the partition constraint. Indeed, due to the fact that, for every attribute A in U, a tuple t can not have more than one A-value, it is then impossible that $\mu(a) \cap \mu(a')$ be nonempty for any distinct values a, a' in $dom(A)$.

Incidentally, if for every A in U we denote by $dom^*(A)$ the set of all A-values such that $\mu(a) \neq \emptyset$, then the set $\{\mu(a) \mid a \in dom^*(A)\}$ is a *partition* of

$\bigcup_{a \in dom^*(A)} \mu(a)$ (whence the name "partition model"). The following example illustrates this important feature.

Example 1 Considering $U = \{A, B, C\}$ and $D = \{ab, bc, ac, a'b', b'c', abc\}$, the tuples in D can be respectively assigned the identifiers 1, 2, 3, 4, 5 and 6. In that case, we have $\mu(a) = \{1, 3, 6\}$, $\mu(a') = \{4\}$, $\mu(b) = \{1, 2, 6\}$, $\mu(b') = \{4, 5\}$, $\mu(c) = \{2, 3, 6\}$, $\mu(c') = \{5\}$, and $\mu(\alpha) = \emptyset$ for any constant α different than a, a', b, b', c and c'.

It is clear that the $\mathcal{T}$-mapping μ is an interpretation and, since $dom^*(A)$, $dom^*(B)$ and $dom^*(C)$ are respectively equal to $\{a, a'\}$, $\{b, b'\}$ and $\{c, c'\}$, it is easy to see that $\{\mu(\alpha) \mid \alpha \in dom^*(A)\}$ is a partition of $\{1, 3, 4, 6\}$, $\{\mu(\beta) \mid \beta \in dom^*(B)\}$ is a partition of $\{1, 2, 4, 5, 6\}$, and $\{\mu(\gamma) \mid \gamma \in dom^*(C)\}$ is a partition of $\{2, 3, 5, 6\}$.

Moreover, extending μ to non unary tuples yields the following regarding the tuples in D: $\mu(ab) = \{1, 6\}$, $\mu(bc) = \{2, 6\}$, $\mu(ac) = \{3, 6\}$, $\mu(a'b') = \{4\}$, $\mu(b'c') = \{5\}$, and $\mu(abc) = \{6\}$. □

Summarizing our discussion, when dealing with consistent tables in [18], only interpretations are relevant. In the present work, we follow the same idea, but we also extend the work of [18] so that we can deal with inconsistencies. As we shall see, non satisfaction of the partition constraint in Definition 1 is the key criterion to characterize inconsistent tuples.

2.2 Functional Dependencies

The notion of functional dependency in our approach is defined as in [18].

Definition 2 Let U be a universe. A *functional dependency* is an expression of the form $X \rightarrow Y$ where X and Y are nonempty sub-sets of U.

A $\mathcal{T}$-mapping μ *satisfies* $X \rightarrow Y$, denoted by $\mu \models X \rightarrow Y$, if for all tuples x and y, respectively over X and Y, the following holds: if $\mu(x) \cap \mu(y) \neq \emptyset$ then $\mu(x) \subseteq \mu(y)$. □

Based on Definition 2, for all X and Y such that $X \cap Y = \emptyset$, and for every $\mathcal{T}$-mapping μ, the following holds:

$$\mu \models X \rightarrow Y \text{ if and only if } \mu \models X \rightarrow A \text{ for every } A \text{ in } Y.$$

This is so because, for every x and y, $\mu(x) \cap \mu(y) \neq \emptyset$ and $\mu(x) \subseteq \mu(y)$ hold if and only if $\mu(x) \cap \mu(a) \neq \emptyset$ and $\mu(x) \subseteq \mu(a)$ hold for every constant a in y.

Therefore without loss of generality we can assume that all functional dependencies are of the form $X \rightarrow A$ where A is an attribute not in X. Under this assumption, the basic notions of databases and of database satisfaction for a $\mathcal{T}$-mapping are defined as follows.

Definition 3 Given universe U, a *database* is a pair $\Delta = (D, \mathcal{FD})$ where D is a table over U and $\mathcal{FD}$ a set of functional dependencies over U whose right hand-side is a single attribute.

A $\mathcal{T}$-mapping μ is said to *satisfy* Δ, denoted by $\mu \models \Delta$, if (i) for every t in D, $\mu(t) \neq \emptyset$, and (ii) μ satisfies every $X \rightarrow A$ in $\mathcal{FD}$. □

To see how our notion of functional dependency relates to the standard one in relational databases [20], recall first that a relation r over universe U satisfies $X \rightarrow A$ if for all tuples t and t' in r such that $t.X = t'.X$, we have $t.A = t'.A$.

In our approach, let $\Delta = (D, \mathcal{FD})$ and consider two tuples t and t' in D such that XA is a subset of $sch(t)$ and of $sch(t')$ and let $t.X = t'.X = x$. Then for every $\mathcal{T}$-mapping μ such that $\mu \models \Delta$, $\mu(t)$ and $\mu(t')$ are nonempty, implying that $\mu(x) \cap \mu(t.A)$ and $\mu(x) \cap \mu(t'.A)$ are also nonempty. By Definition 2, this implies that $\mu(x)$ is a sub-set of $\mu(t.A)$ and of $\mu(t'.A)$. As a consequence, assuming that $t.A \neq t'.A$ (*i.e.,* that $X \rightarrow A$ is not satisfied in the sense of the relational model), means that $\mu(t.A) \cap \mu(t'.A)$ is nonempty, which implies that μ *can not be an interpretation.*

Therefore if we restrict $\mathcal{T}$-mappings to be interpretations then the notion of functional dependency satisfaction in our approach is *the same* as that of relational databases. As we shall see, this observation supports the notion of consistency for Δ, to be given later (in Definition 5).

Given $\Delta = (D, \mathcal{FD})$ and tuples t, t', t'', the following notations are extensively used in the remainder of the paper.

- $\Delta \vdash t$, denotes that if $\mu \models \Delta$ then $\mu(t) \neq \emptyset$.
- $\Delta \vdash (t \sqcap t')$, denotes that if $\mu \models \Delta$ then $\mu(t) \cap \mu(t') \neq \emptyset$.
- $\Delta \vdash (t \preceq t')$ denotes that if $\mu \models \Delta$ then $\mu(t) \subseteq \mu(t')$.
- $\Delta \vdash (t \preceq t' \sqcap t'')$ denotes that if $\mu \models \Delta$ then $\mu(t) \subseteq \mu(t') \cap \mu(t'')$.

Given $\Delta = (D, \mathcal{FD})$, we now build a particular $\mathcal{T}$-mapping μ such that $\mu \models \Delta$ as follows: Let $(\mu_i)_{i \geq 0}$ be the sequence defined by the steps below:

1. Associate each tuple t in D with an identifier, $id(t)$, called the *tuple identifier* of t. $id(t)$ is a natural number that identifies t uniquely.
2. Let μ_0 be the mapping defined for every domain constant a by:
   - $\mu_0(a) = \{id(t) \mid t \in D \text{ and } a \sqsubseteq t\}$.
3. Starting with $\mu_i = \mu_0$, while μ_i changes define μ_{i+1} as follows:
   - For every constant a, let $M(a)$ be the union of all $\mu_i(x)$ for which there exists $X \rightarrow A$ in $\mathcal{FD}$ such that $sch(x) = X$, a is in $dom(A)$, $\mu_i(xa) \neq \emptyset$ and $\mu_i(x) \not\subseteq \mu_i(a)$.
   - If $M(a) \neq \emptyset$, $\mu_{i+1}(a) = \mu_i(a) \cup M(a)$; otherwise, $\mu_{i+1}(a) = \mu_i(a)$.

We notice that computing μ_{i+1} based on μ_i is finite even if attribute domains are infinite. This is so because if $M(a)$ is nonempty then so are $\mu_i(a)$ and all $\mu_i(x)$ that have to be tested. Therefore, the computation has to be processed only against tuples t such that $\mu_i(t) \neq \emptyset$, which are in finite number since the set D is finite.

The following lemma shows that the sequence $(\mu_i)_{i \geq 0}$ has a limit, which plays a central role in our approach, as will be seen later.

Lemma 1 *For every $\Delta = (D, \mathcal{FD})$, the sequence $(\mu_i)_{i \geq 0}$ has a unique limit μ^* such that $\mu^* \models \Delta$. Moreover:*

1. *For all a_1 and a_2 in the* same attribute domain *$dom(A)$, if $\mu^*(a_1) \cap \mu^*(a_2) \neq \emptyset$ then there exist $X \rightarrow A$ in $\mathcal{FD}$ and x over X such that $\mu^*(x) \neq \emptyset$ and $\mu^*(x) \subseteq \mu^*(a_1) \cap \mu^*(a_2)$.*
2. *For all α and β, $\Delta \vdash (\alpha \sqcap \beta)$ holds if and only if $\mu^*(\alpha) \cap \mu^*(\beta) \neq \emptyset$ holds.*

Proof See Appendix A. □

Given $\Delta = (D, \mathcal{FD})$, Lemma 1 shows the following:

1. There always exists a $\mathcal{T}$-mapping μ such that $\mu \models \Delta$.

2. When two constants from the same domain have common identifiers with respect to μ^* then this is due to a functional dependency.
3. For every tuple t, $\Delta \vdash t$ if and only if $\mu^*(t) \neq \emptyset$.

It is important to note that the $\mathcal{T}$-mapping μ^* as defined in Lemma 1 is not necessarily an interpretation as the following example shows.

Example 2 Let $U = \{A, B, C\}$ and $\Delta = (D, \mathcal{FD})$ where $D = \{ab, bc, abc'\}$ and $\mathcal{FD} = \{B \rightarrow C\}$. Associating ab, bc and abc' respectively with 1, 2 and 3, μ^* is obtained as follows:
• First, we have $\mu_0(a) = \{1, 3\}$, $\mu_0(b) = \{1, 2, 3\}$, $\mu_0(c) = \{2\}$ and $\mu_0(c') = \{3\}$ and $\mu_0(\alpha) = \emptyset$ for any other domain constant α.
• Then, considering $B \rightarrow C$, we have $M(c) = \mu_0(b)$ (because $\mu_0(bc) = \{2\}$ but $\mu_0(b) \not\subseteq \mu_0(c)$) and $M(c') = \mu_0(b)$ (because $\mu_0(bc') = \{3\}$ but $\mu_0(b) \not\subseteq \mu_0(c')$). Thus μ_1 is defined by $\mu_1(a) = \{1, 3\}$, $\mu_1(b) = \mu_1(c) = \mu_1(c') = \{1, 2, 3\}$ and $\mu_1(\alpha) = \emptyset$ for any other domain constant α.

Hence, $\mu^* = \mu_1$ and we remark that $\mu^*(c) \cap \mu^*(c') \neq \emptyset$, thus that μ^* is not an interpretation. Nevertheless, as stated by Lemma 1, it is easy to see that $\mu^* \models \Delta$. □

Notice that the authors of [19] use a construction similar to that of Lemma 1 to define a minimal model of Δ, called 'query model', assuming that D is consistent with $\mathcal{FD}$.

Now, in order to characterize when $\Delta \vdash (t \preceq a)$ holds, we introduce the notion of *closure of a tuple t in Δ* inspired by the well known relational notion of closure of a relation scheme with respect to a set of functional dependencies [20].

Definition 4 Given a database $\Delta = (D, \mathcal{FD})$ and a tuple t, the *closure of t in Δ* (or *closure of t* for short, when Δ is understood), denoted by t^+, is the set of all domain constants a such that $\Delta \vdash (t \preceq a)$ holds. □

We notice that, based on Definition 4, for every constant a occurring in a tuple t (*i.e.,* if $a \sqsubseteq t$ holds) then a is in t^+, because, in this case, $\mu(t) \subseteq \mu(a)$ holds for every $\mathcal{T}$-mapping μ. However constants not occurring in t may also appear in t^+ due to functional dependencies, as shown in the following example.

Example 3 Continuing Example 2 where $U = \{A, B, C\}$ and $\Delta = (D, \mathcal{FD})$ with $D = \{ab, bc, abc'\}$ and $\mathcal{FD} = \{B \rightarrow C\}$, we show that c belongs to $(ab)^+$.

Indeed, for every μ such that $\mu \models \Delta$, we have $\mu(ab) \subseteq \mu(b)$ (since $b \sqsubseteq ab$) and $\mu(b) \subseteq \mu(c)$ (due to $B \rightarrow C$ and the fact that $\mu(b) \cap \mu(c) \neq \emptyset$ must hold). Hence, by transitivity, $\mu(ab) \subseteq \mu(c)$ holds, implying that $\Delta \vdash (ab \preceq c)$ holds, which by Definition 4, means that c belongs to $(ab)^+$. It should also be noticed that a similar argument shows that c' also belongs to $(ab)^+$. □

Clearly computing the closure directly from its definition is inefficient. Algorithm 1 gives a method for computing the closure, and the following lemma states that the algorithm computes the closure correctly.

However, based on Lemma 1, in order to effectively implement Algorithm 1, the test line 5 requires to compute the $\mathcal{T}$-mapping μ_t^* associated with Δ_t, as explained earlier before stating the lemma.

Lemma 2 *Let $\Delta = (D, \mathcal{FD})$ and t a tuple. Then Algorithm 1 computes correctly the closure t^+ of t.*

Algorithm 1 Closure of t

Input: $\Delta = (D, \mathcal{FD})$ and a tuple t.
Output: The closure t^+ of t
1: $\Delta_t := (D_t, \mathcal{FD})$ where $D_t = D \cup \{t\}$
2: $t^+ := \{a \mid a \sqsubseteq t\}$
3: **while** t^+ changes **do**
4: **for all** $X \to A \in \mathcal{FD}$ **do**
5: **for all** x such that for every b in x, $b \in t^+$ and $\Delta_t \vdash xa$ **do**
6: $t^+ := t^+ \cup \{a\}$
7: **return** t^+

Proof See Appendix B. □

We draw attention on the fact that the database involved in Algorithm 1 is not Δ but the database Δ_t that can be seen as Δ in which the tuple t has been added.

It should however be noticed that in case $\Delta \vdash t$, this distinction is not necessary because in this case, for every tuple q, $\Delta \vdash q$ holds if and only if $\Delta_t \vdash q$ holds. This is a consequence of the fact that, as seen in Appendix B, if $\Delta \vdash t$ then for every $\mathcal{T}$-mapping μ, $\mu \models \Delta$ holds if and only if $\mu \models \Delta_t$.

On the other hand, the following example shows that when $\Delta \nvdash t$, the introduction of Δ_t instead of Δ is necessary for correctly computing t^+.

Example 4 Let $U = \{A, B, C\}$ and $\Delta = (D, \mathcal{FD})$ where $D = \{ac, b\}$ and $\mathcal{FD} = \{A \to B, B \to C\}$.

It is easy to see that when numbering the tuples in D by 1 for ac and 2 for b, the $\mathcal{T}$-mapping μ^* for Δ is defined by: $\mu^*(a) = \mu^*(c) = \{1\}$, $\mu^*(b) = \{2\}$ and $\mu^*(\alpha) = \emptyset$ for any other constant α.

For $t = ab$, we argue that c is in t^+, that is, for every μ such that $\mu \models \Delta$, $\mu(ab) \subseteq \mu(c)$ holds. Indeed, this trivially holds if $\mu(ab) = \emptyset$ (as is the case with μ^*), and otherwise the following proof can be done:
• As $\mu(ab) \neq \emptyset$, $A \to B$ implies that $\mu(a) \subseteq \mu(b)$.
• As $\mu(ac) \neq \emptyset$, $\mu(a) \subseteq \mu(b)$ implies $\mu(bc) \neq \emptyset$. Thus, $\mu(b) \subseteq \mu(c)$, due to $B \to C$.
• Therefore, $\mu(a) \subseteq \mu(c)$, and since $\mu(ab) \subseteq \mu(a)$, we have $\mu(ab) \subseteq \mu(c)$.

On the other hand, computing $(ab)^+$ using a modified version of Algorithm 1 where Δ_t is replaced by Δ would only output a and b in the closure. It should also be noticed that computing $(ab)^+$ using Algorithm 1 is as follows: by the statement on line 2, a and b are inserted into the closure, and then, since for $t = ab$, $\Delta_t \vdash ab$ the above reasoning shows that $\Delta_t \vdash bc$ as well. Therefore, c is inserted into the closure because the test line 5 succeeds. □

The following example shows a case where the tuple t of which the closure is computed, is such that $\Delta \vdash t$.

Example 5 As seen in Example 2, if $U = \{A, B, C\}$ and $\Delta = (D, \mathcal{FD})$ where $D = \{ab, bc, abc'\}$ and $\mathcal{FD} = \{B \to C\}$, μ^* is defined by: $\mu^*(a) = \{1, 3\}$, $\mu^*(b) = \mu_1(c) = \mu^*(c') = \{1, 2, 3\}$ and $\mu^*(\alpha) = \emptyset$ for any other domain constant α.

In this case, the computation of $(ab)^+$ according to Algorithm 1 is as follows:
• As ab is in D, $\Delta_t = \Delta$. We thus run Algorithm 1 with Δ instead of Δ_t.
• $(ab)^+$ is first set to $\{a, b\}$.
• Considering $B \to C$, since b is in $(ab)^+$, and since $\Delta \vdash bc$ and $\Delta \vdash bc'$ (this holds because $\mu^*(c)$ and $\mu^*(c')$ are nonempty), c and c' are inserted in $(ab)^+$.

As no further step is processed, $(ab)^+ = \{a, b, c, c'\}$, as seen in Example 3. Thus $\Delta \vdash (ab \preceq c)$ and $\Delta \vdash (ab \preceq c')$ hold, implying $\Delta \vdash (ab \preceq c \sqcap c')$. □

3 Semantics

In this section we provide basic definitions and properties regarding the truth value associated with a tuple. The following definition is borrowed from [18].

Definition 5 Δ is said to be *consistent* if there exists an *interpretation* μ such that $\mu \models \Delta$. □

Since in our approach, inconsistent tables are *not* discarded, it is crucial to be able to provide semantics to any $\Delta = (D, \mathcal{FD})$, being it consistent or not. To this end, inspired by Belnap's Four-valued logic [5], we consider *four* possible truth values for a given tuple t in $\mathcal{T}$, namely `true`, `false`, `inc` (*i.e.,* inconsistent) and `unkn` (*i.e.,* unknown).

In order to formalize the exact meaning of these truth values in our approach, we introduce the following terminology and notation:

- If $\Delta \vdash t$ holds, t is said to be *potentially true* in Δ. Notice here that by Lemma 1, t is potentially true if and only if $\mu^*(t) \neq \emptyset$.
- If $\Delta \vdash (t \preceq a \sqcap a')$ holds for some distinct a and a' in the *same* attribute domain, then we use the notation $\Delta \nsim t$, and in this case, t is said to be *potentially false* to reflect that $\mu(a) \cap \mu(a')$ must be empty for μ to be an interpretation. By Definition 4, $\Delta \nsim t$ holds if and only if there exist a and a' in the same attribute domain such that a and a' are in t^+.

Consequently, if a tuple t is such that $\Delta \vdash t$ and $\Delta \nsim t$, then for μ to be an interpretation, μ must associate t with a set expected to be empty and nonempty, which is of course a case of inconsistency! This explains why, in our approach, 'potentially true' and 'potentially false', should respectively be understood as 'true or inconsistent' and 'false or inconsistent'. More precisely, each tuple is assigned one of the four truth values according to the following definition.

Definition 6 Given $\Delta = (D, \mathcal{FD})$ and a tuple t, the truth value of t in Δ, denoted by $v_\Delta(t)$, is defined as follows:

- $v_\Delta(t) = \texttt{true}$ if $\Delta \vdash t$ and $\Delta \not\sim t$; t is said to be *true in* Δ.
- $v_\Delta(t) = \texttt{false}$ if $\Delta \not\vdash t$ and $\Delta \nsim t$; t is said to be *false in* Δ.
- $v_\Delta(t) = \texttt{inc}$ if $\Delta \vdash t$ and $\Delta \nsim t$; t is said to be *inconsistent in* Δ.
- $v_\Delta(t) = \texttt{unkn}$ if $\Delta \not\vdash t$ and $\Delta \not\sim t$; t is said to be *unknown in* Δ. □

We point out that the four truth values as defined above correspond exactly to the four truth values defined in the Four-valued logic [5]. Intuitively speaking, Definition 6 implies the following:

- Tuples whose truth value is `inc` are those tuples that are potentially true *and* potentially false, which fits the intuition that inconsistent information are somehow true and false at the same time.
- Tuples whose truth value is `true` are those tuples occurring in D or that are derived based on functional dependencies, but that are not inconsistent.

– Tuples whose truth value is `false` are those that can *not* be true, in the sense that, otherwise they would generate an inconsistency.
– Tuples whose truth value is `unk` are all other tuples, namely all tuples that do not fall in one of the previous three categories.

Another important consequence of Definition 6 is that our approach does *not* follow the Closed World Assumption (CWA), according to which any non true tuple is false [17]. We illustrate Definition 6 through the following example.

Example 6 As in Example 2, let $U = \{A, B, C\}$ and $\Delta = (D, \mathcal{FD})$ where $D = \{ab, bc, abc'\}$ and $\mathcal{FD} = \{B \rightarrow C\}$.

It has been seen in Example 5 that $(ab)^+ = \{a, b, c, c'\}$. Thus $\Delta \mid\!\sim ab$ holds. Moreover, it is easy to see from Example 2 that $\mu^*(ab) \neq \emptyset$, implying that $\Delta \vdash ab$ holds as well. As a consequence, by Definition 6, $v_\Delta(ab) = \texttt{inc}$, meaning that ab is inconsistent in Δ. We notice that similar arguments hold for abc, abc', bc, bc' and b, showing that these tuples are also inconsistent in Δ.

Moreover, based on Definition 5, we also argue that Δ is *not* consistent, because every μ such that $\mu \models \Delta$ cannot be an interpretation. This is so because $\mu^*(c) \cap \mu^*(c') \neq \emptyset$ and Lemma 1 imply that for μ such that $\mu \models \Delta$, $\mu(c) \cap \mu(c') \neq \emptyset$.

Now, consider the tuple bc'' where c'' is a constant in $dom(C)$ distinct from c and c'. To compute $(bc'')^+$ using Algorithm 1, the database $\Delta_t = (D_t, \mathcal{FD})$ where $D_t = \{ab, bc, abc', bc''\}$ is defined and then, the closure is first set to $\{b, c''\}$. Then in subsequent computations, based on $B \rightarrow C$ and the fact that $\Delta_t \vdash bc$ and $\Delta_t \vdash bc'$, c and c' are inserted in the closure of bc''.

It therefore follows that $(bc'')^+ = \{b, c'', c, c'\}$, thus that $\Delta \mid\!\sim bc''$ holds. Since $\Delta \nvdash bc''$ (because $\mu^*(bc'') = \emptyset$ and $\mu^* \models \Delta$), it follows that $v_\Delta(bc'') = \texttt{false}$. Hence bc'' and all its super-tuples are false in Δ.

As an example of unknown tuple in Δ, let a' be in $dom(A)$ such $a' \neq a$, and consider $a'c$. Since $\mu^*(a'c) = \emptyset$, $\Delta \nvdash a'c$. On the other hand, it can be seen that $(a'c)^+ = \{a', c\}$, because when running Algorithm 1, $D \cup \{a'c\}$ does not allow for any specific tuple derivation using $B \rightarrow C$. Hence, $\Delta \not\mid\!\sim a'c$, which shows that $v_\Delta(a'c) = \texttt{unkn}$. □

Computing true tuples and inconsistent tuples in Δ is addressed in the next section, whereas computing false tuples is still an open issue that is not addressed in the present paper. Nevertheless, the following example shows that computing *all* inconsistent tuples is not an easy task.

Example 7 Let $\Delta = (D, \mathcal{FD})$ be defined over $U = \{A, B, C\}$ by $D = \{abc, ac'\}$ and $\mathcal{FD} = \{A \rightarrow B, B \rightarrow C\}$.

Here again, the tuples in D along with the functional dependencies in $\mathcal{FD}$ show no explicit inconsistency. However computing μ^* yields the following:
• To define μ_0, we associate the tuples abc and ac' with the natural numbers 1 and 2, respectively. It follows that $\mu_0(a) = \{1, 2\}$, $\mu_0(b) = \{1\}$, $\mu_0(c) = \{1\}$, $\mu_0(c') = \{2\}$ and $\mu_0(\alpha) = \emptyset$ for any other domain constant α.
• The next steps modify μ_0 so as to satisfy $A \rightarrow B$ and $B \rightarrow C$ as follows:

1. Due to $A \rightarrow B$, μ_1 is defined by: $\mu_1(a) = \{1, 2\}$, $\mu_1(b) = \{1, 2\}$, $\mu_1(c) = \{1\}$ and $\mu_1(c') = \{2\}$;
2. Due to $B \rightarrow C$, μ_2 is defined by: $\mu_2(a) = \{1, 2\}$, $\mu_2(b) = \{1, 2\}$, $\mu_2(c) = \{1, 2\}$ and $\mu_2(c') = \{1, 2\}$.

As $\mu_2 \models \mathcal{FD}$, $\mu^* = \mu_2$. Moreover, we have $a^+ = \{a, b, c, c'\}$ and $b^+ = \{b, c, c'\}$ showing that, by Lemma 2, $\Delta \vdash (a \preceq c \sqcap c')$ and $\Delta \vdash (b \preceq c \sqcap c')$, thus that a and b are inconsistent in Δ. It can then be seen that, for example, abc, bc' and ac are also inconsistent in Δ.

Now, let $\Delta_1 = (D_1, \mathcal{FD})$ such that $D_1 = \{ac, ac'\}$. In this case, μ^* is defined by $\mu^*(a) = \{1, 2\}$, $\mu^*(c) = \{1\}$, $\mu^*(c') = \{2\}$ and $\mu^*(\alpha) = \emptyset$ for any other domain constant α. Therefore, $a^+ = \{a\}$, showing that a is *not* inconsistent in Δ_1. As a consequence, ac, ac' along with all their sub-tuples are true in Δ_1 and all other tuples are unknown in Δ_1. □

The following proposition shows that our notion of inconsistent tuple complies with Definition 5.

Proposition 1 *$\Delta = (D, \mathcal{FD})$ is consistent if and only if there exists no tuple t such that $v_\Delta(t) = \texttt{inc}$.*

Proof We first note that if there exists a tuple t such that $v_\Delta(t) = \texttt{inc}$, then $\Delta \vdash t$ and $\Delta \nvdash t$. Hence there exist a and a' in the same attribute domain $dom(A)$ such that $\Delta \vdash (t \preceq a \sqcap a')$. Thus every $\mathcal{T}$-mapping μ such that $\mu \models \Delta$ satisfies that $\mu(t) \neq \emptyset$ and $\mu(t) \subseteq \mu(a) \cap \mu(a')$, implying that $\mu(a) \cap \mu(a') \neq \emptyset$. Hence, μ is not an interpretation, showing that, by Definition 5, Δ is not consistent.

Conversely, assuming that there is no tuple t such that $v_\Delta(t) = \texttt{inc}$, that is such that $\Delta \vdash t$ and $\Delta \nvdash t$, we prove that μ^* is an interpretation of Δ. Indeed, if a_1 and a_2 are two constants in the same attribute domain A such that $\mu^*(a_1) \cap \mu^*(a_2) \neq \emptyset$, then by Lemma 1(1), there exist $X \rightarrow A$ in $\mathcal{FD}$ and x over X such that $\mu^*(x) \neq \emptyset$ and $\mu^*(x) \subseteq \mu^*(a_1) \cap \mu^*(a_2)$. Thus by Lemma 1(2), for every μ such that $\mu \models \Delta$, $\mu(x) \subseteq \mu(a_1) \cap \mu(a_2)$ and $\mu(x) \neq \emptyset$. We therefore obtain that $\Delta \nvdash x$ and $\Delta \vdash x$, thus that $v_\Delta(x) = \texttt{inc}$, which is a contradiction. Therefore μ^* is an interpretation, and the proof is complete. □

To end the section, based on Definition 6, we stress the following important remarks about potentially true and potentially false tuples in a given Δ:

- Let t be a potentially true tuple. Since $\Delta \vdash t$ holds, as a consequence of Lemma 1, we have that $\mu^*(t) \neq \emptyset$. Therefore true or inconsistent tuples are those tuples that are associated with a nonempty set by *every* $\mathcal{T}$-mapping μ such that $\mu \models \Delta$. This implies that potentially true tuples in Δ are built up with constants occurring in D, and thus are in finite number. We provide in this paper effective algorithms for computing the sets of true tuples and inconsistent tuples.
- As potentially false tuples t are such that $\Delta \nvdash t$, they may not satisfy $\Delta \vdash t$. Hence, Lemma 1 cannot be used to characterize them. Moreover, if $\Delta \nvdash t$, then every tuple t' such that $t \sqsubseteq t'$ also satisfies $\Delta \nvdash t'$. This is so because in this case, if $\Delta \vdash (t \preceq a \sqcap a')$, then $\Delta \vdash (t' \preceq a \sqcap a')$ holds as well. Thus, the number of potentially false tuples may be infinite in case some of the attribute domains are infinite.
- Since every false tuple t is potentially false and does not satisfy $\Delta \vdash t$, it follows that the number of false tuples may be infinite in case some of the attribute domains are infinite. Moreover, contrary to true and inconsistent tuples, characterizing false tuples other than through Definition 6 is currently unknown to the authors. This issue will be the subject of future work.

Algorithm 2 Chasing a table

Input: $\Delta = (D, \mathcal{FD})$
Output: $\Delta^* = (D^*, \mathcal{FD})$ and a set $inc(\mathcal{FD})$ containing sets of tuples associated with each $X \to A$ in $\mathcal{FD}$

$D^* := D$
for all $X \to A$ in $\mathcal{FD}$ **do**
    $inc(X \to A) := \emptyset$
while D^* changes **do**
    for all $X \to A \in \mathcal{FD}$ **do**
        for all t_1 in D^* such that $XA \subseteq sch(t_1)$ **do**
            for all t_2 in D^* such that $X \subseteq sch(t_2)$ and $t_1.X = t_2.X$ **do**
                if $A \notin sch(t_2)$ **then**
                    $D^* := D^* \cup \{t_2 a_1\}$ where $a_1 = t_1.A$
                if $A \in sch(t_2)$ and $t_1.A \neq t_2.A$ **then**
                    Let $y_i = t_i.(sch(t_i) \setminus A)$ and $a_i = t_i.A$, for $i = 1, 2$
                    $D^* := D^* \cup \{y_2 a_1\}$
                    // $y_1 a_2$ is inserted into D^* when processing t_2 in place of t_1
                    // and t_1 in place of t_2
                    $inc(X \to A) := inc(X \to A) \cup \{x\}$ where $x = t_1.X = t_2.X$
// Reduction: keep in D^* only maximal tuples
for all t_1 in D^* **do**
    for all t_2 in D^* **do**
        if $t_2 \sqsubseteq t_1$ and $t_1 \neq t_2$ **then**
            $D^* := D^* \setminus \{t_2\}$
$inc(\mathcal{FD}) := \{inc(X \to A) \mid inc(X \to A) \neq \emptyset\}$
return $\Delta^* = (D^*, \mathcal{FD})$ and $inc(\mathcal{FD})$

4 Computing the Semantics

Similarly to standard two valued logic, where computing the semantics of Δ means computing the set of all tuples true in Δ, in our approach, computing the semantics amounts to compute all true, inconsistent or false tuples, knowing that unknown tuples are the remaining ones.

However, as mentioned above, the set of false tuples may be infinite, making it impossible to compute them all. In this work, we do not address this issue, we rather concentrate on potentially true tuples, with the goal of investigating consistent query answering in our approach (see Section 5).

4.1 The Chase Procedure in our Approach

We first propose an effective algorithm for the computation of all potentially true tuples in a given Δ. This algorithm, referred to as Algorithm 2, is in fact inspired by the standard chase algorithm [18,20], with the main difference that when a functional dependency cannot be satisfied, our algorithm does *not* stop.

Instead, our chasing algorithm carries on the computation, returning a database $\Delta^* = (D^*, \mathcal{FD})$, where Δ^* and D^* are respectively referred to as the *chased database* and the *chased table*. The algorithm also returns a set $inc(\mathcal{FD})$, based on which inconsistent and true tuples are shown to be efficiently computed. We illustrate Algorithm 2 in the context of our introductory example.

Example 8 Running Algorithm 2 with the table D shown in Figure 1, and recalled in Figure 2, produces the table D^* shown in the right of Figure 2, along with

D	Id	K	M	C
	i_1	k	m	c
	i_1		m'	
	i_1	k		c
	i_2	k'	m'	c
	i_2	k'	m''	
	i_2	k'		c'
	i_3		m	
	i_3	k'		

D^*	Id	K	M	C
	i_1	k	m	c
	i_1	k	m'	c
	i_2	k'	m'	c
	i_2	k'	m''	c
	i_2	k'	m'	c'
	i_2	k'	m''	c'
	i_3	k'	m	

Fig. 2 The table D and its chased version D^*

the set $inc(\mathcal{FD}) = \{inc(Id \to K), inc(Id \to C)\}$ where $inc(Id \to K) = \emptyset$ and $inc(Id \to C) = \{i_2\}$. The main steps of the algorithm work as follows:

• First, D^* is assigned D, and $inc(Id \to K)$ and $inc(Id \to C)$ are assigned $\emptyset$.
• Due to the statement on line 9, the first two rows in D (thus in D^*) generate the new tuples (i_1, k, m') and (i_1, m', c). Similarly, applying $Id \to K$ to the last two rows in D generates the new tuple (i_3, k', m).
The rows 4 and 5 in D generate (i_2, k', m'', c) and the rows 5 and 6 generate (i_2, k', m'', c'). Moreover, due to the statement on line 12, the rows 4 and 6 generate (i_2, k', m', c') and (i_2, k', c) and i_2 is inserted in $inc(Id \to C)$.
• With these new tuples at hand, the loop on line 4 proceeds further, generating (i_1, k, m', c) by the statement on line 9. No new tuple is generated at this stage.
• The loop on line 4 is processed once again, producing no new tuple. When running the reduction step against the current state of D^*, the following tuples are removed: (i_1, m'), (i_1, k, m'), (i_1, m', c), (i_1, k, c), (i_2, k', m''), (i_2, k', c'), (i_2, k', c), (i_3, m) and (i_3, k').

It is important to notice that, although tuples have been added in D^* during the processing, the final number of tuples in D^* is less than that in D. Although this particular result cannot be proven in general, it will be shown that in the worst case, the size of D^* remains polynomial in the size of D.

We emphasize that some nulls present in D have been replaced by actual values in D^*, thanks to the functional dependencies in $\mathcal{FD}$. For example the second tuple in D with two nulls has been 'completed' into a total tuple in D^*. However, such a completion has not been possible for *every* tuple in D^*. Namely, the C-value in the last tuple of D^* is left as null.

Keeping in line with our statement that 'a missing value exists only if it is inferred from the functional dependencies', this indicates that the C-value of this tuple could *not* be determined based on the content of D and $\mathcal{FD}$, and no other conclusion can be drawn regarding this null.

To see why the two insertions mentioned in the statement on line 12 are needed, we first recall from [20] that, in the traditional case, the chased table characterizes the semantics of the input table, in case no inconsistency has been detected[1]. In this work our goal is similar, but has to be adapted to our context. Namely, we expect that the chased table D^* can provide a syntactical characterization of all possibly true tuples in Δ, that is of all tuples t such that $\Delta \vdash t$ holds.

[1] In traditional chase, the semantics of a table D containing nulls is the set of all tuples true in every instance of D, *i.e.,* in every relation R over U with no nulls, that satisfies the functional dependencies and such that for every t in D, there exists r in R such that $r.T = t$.

In the context of our example, if we assume that (i_2, k', m', c') is not inserted during the processing then Algorithm 2 would *not* fit our semantics. Indeed, for every $\mathcal{T}$-mapping μ such that $\mu \models \Delta$, $\mu(i_2) \subseteq \mu(c')$ holds because of $Id \rightarrow C$ applied to the seventh row in D. Thus, $\mu(i_2, k', m', c') = \mu(k', m', c')$, and since $\mu(k', m', c')$ is nonempty (due to the fifth row in D), $\Delta \vdash (i_2, k', m', c')$. Hence (i_2, k', m', c') must appear in D^* to fulfill our expectation.

Adding such 'new' tuples when chasing a table is one of the main features of our approach, as compared with traditional chase. This step should be seen as a 'by-product' of carrying on the computation even after encountering a violation of a functional dependency. □

The following lemma shows that Algorithm 2 provides an operational means to characterize the tuples t such that $\mu^*(t) \neq \emptyset$.

Lemma 3 *Algorithm 2 applied to $\Delta = (D, \mathcal{FD})$ always terminates. Moreover, for every tuple t, $\mu^*(t) \neq \emptyset$ holds if and only if t is in $\mathsf{LoCl}(D^*)$.*

Proof See Appendix C. □

Recalling that $\mathsf{LoCl}(D^*)$ denotes the Lower Closure of D^*, that is the set of all sub-tuples of tuples in D^*, Lemma 3 shows that D^* is a 'tabular' version of the set of all tuples t such that $\mu^*(t) \neq \emptyset$, that is, by Lemma 1, a 'tabular' version of the set of all tuples t such that $\Delta \vdash t$. Therefore, D^* provides a syntactical characterization of the set of all tuples t such that $\Delta \vdash t$, as expected in the previous example.

4.2 Computing True Tuples and Inconsistent Tuples

As mentioned just above, Lemma 1 and Lemma 3 show that, given $\Delta = (D, \mathcal{FD})$, a tuple t is in $\mathsf{LoCl}(D^*)$ if and only if $\Delta \vdash t$ holds, that is, if and only if t is potentially true in Δ, that is if and only if t is either true or inconsistent in Δ.

To see how to compute the set of all inconsistent tuples, we first recall the notion of *closure of a relation scheme* as defined in relational database theory [20].

Given a set $\mathcal{FD}$ of functional dependencies and a relation scheme X, the *closure of X with respect to $\mathcal{FD}$*, or more simply the *closure of X*, denoted by X^+, is the set of all attributes A in U such that every table D satisfying $\mathcal{FD}$ in the sense of relational tables, also satisfies $X \rightarrow A$.

It is well-known that X^+ is computed through the following two steps that are quite similar to the steps of Algorithm 1:

$X^+ := X$
while X^+ changes **do**
  for all $Y \rightarrow B$ in $\mathcal{FD}$ such that $Y \subseteq X^+$ **do**
    $X^+ := X^+ \cup \{B\}$
return X^+

The following proposition shows a strong relationship between the closure of a relation scheme as recalled above and the closure of a tuple as stated in Definition 4.

Proposition 2 *Let $\Delta = (D, \mathcal{FD})$ and t be such that $\Delta \vdash t$. For every tuple q and every a in $dom(A)$ such that $q \sqsubseteq t$ and $a \sqsubseteq t$, a belongs to q^+ if and only if A belongs to Q^+.*

Proof See Appendix D. □

Using the notion of relation scheme closure, we introduce Algorithm 3 which computes the set of inconsistent tuples in Δ. The correctness of this algorithm is shown in Lemma 4.

Algorithm 3 Inconsistent tuples in $\Delta = (D, \mathcal{FD})$

Input: The output of Algorithm 2, that is $\Delta^* = (D^*, \mathcal{FD})$ and $inc(\mathcal{FD})$.
Output: The set $\mathsf{Inc}(\Delta)$

```
1: Inc(Δ) := ∅
2: for all t in D* do
3:     for all X → A in FD such that XA ⊆ T do
4:         if x = t.X is in inc(X → A) then
5:             for all q such that q ⊑ t do
6:                 if X ⊆ Q+ then
7:                     Inc(Δ) := Inc(Δ) ∪ {t.Q}
8: return Inc(Δ)
```

Lemma 4 *Given $\Delta = (D, \mathcal{FD})$, a tuple t is inconsistent in Δ if and only if $t \in \mathsf{Inc}(\Delta)$.*

Proof See Appendix E. □

The following proposition characterizes inconsistent and true tuples in Δ based on Algorithm 2 and Algorithm 3.

Proposition 3 *Given $\Delta = (D, \mathcal{FD})$ and a tuple t:*
1. t is inconsistent in Δ if and only if $t \in \mathsf{Inc}(\Delta)$.
2. t is true in Δ if and only if $t \in \mathsf{LoCl}(D^*) \setminus \mathsf{Inc}(\Delta)$.

Proof Immediate consequence of Definition 6, Lemma 3 and Lemma 4. □

The following example illustrates Algorithm 3 and Proposition 3.

Example 9 As in Example 7, let $\Delta = (D, \mathcal{FD})$ over $U = \{A, B, C\}$ where $D = \{abc, ac'\}$ and $\mathcal{FD} = \{A \to B, B \to C\}$. The tabular version of D is shown on the left below, whereas D^* is shown on the right.

D	A	B	C
	a	b	c
	a		c'

D^*	A	B	C
	a	b	c
	a	b	c'

Running Algorithm 2, D^* is first set to D and abc' is inserted in D^* by the statement line 9 due to $A \to B$. Then, b is inserted in $inc(B \to C)$ by the statement line 13, due to the tuples abc and abc'. Thus, the table D^* output by Algorithm 2 is as shown above and $inc(\mathcal{FD}) = \{inc(A \to B), inc(B \to C)\}$ where $inc(A \to B) = \emptyset$ and $inc(B \to C) = \{b\}$.

When running Algorithm 3 for abc in D^*, since b is in $inc(B \to C)$, b, ab, bc and abc are inserted into $\mathsf{Inc}(\Delta)$, due to the statement on line 7. This is so because the schema Q of each of these tuples contains B, and so, satisfies $B \subseteq Q^+$.

Moreover, for $q = a$, due to $A \to B$, we have $A^+ = ABC$ and thus, $B \subseteq A^+$ holds, showing that a is inserted in $\mathsf{Inc}(\Delta)$ on line 7. A similar reasoning holds

for $q = ac$ because $B \in (AC)^+$. Thus, ac is also inserted in $\mathsf{Inc}(\Delta)$ on line 7. The only remaining possibility is $q = c$, and does not modify $\mathsf{Inc}(\Delta)$ because $B \not\subseteq C^+$. A similar computation is performed with abc' in D^*, adding bc', abc' and ac' in $\mathsf{Inc}(\Delta)$. As no other tuple can be inserted in $\mathsf{Inc}(\Delta)$, Algorithm 3 returns

$\mathsf{Inc}(\Delta) = \{abc,\ abc',\ ab,\ ac,\ ac',\ bc,\ bc',\ a,\ b\}$,

which, by Proposition 3(1), is the set of all inconsistent tuples in Δ. As a consequence, by Proposition 3(2), c and c' are the only true tuples in Δ.

Now, as in Example 7, referring to $\Delta_1 = (D_1, \mathcal{FD})$ with $D_1 = \{ac, ac'\}$, it is easy to see that $D_1^* = D_1$. This implies that Δ_1 is consistent, and that ac, ac', a, c and c' are true in Δ_1. □

4.3 Complexity Issues

We argue that the computation of inconsistent and true tuples in $\Delta = (D, \mathcal{FD})$ is polynomial in the size of the table D and in the order of the 'number of conflicts with respect to functional dependencies' (to be defined shortly). To see this, denoting by $|E|$ the cardinality of a set E, we investigate the complexities of Algorithm 2 and of Algorithm 3.

Regarding Algorithm 2, we first notice that, contrary to the standard chase algorithm [20], rows are added in the table during the computation, and some others are then removed by the reduction statement of line 14. To assess the size of the table D^* during the processing, we point out the following:

- If no inconsistency is found during the processing of the while-loop on line 4, at most one tuple is added in D^* as the 'join' of two tuples in D by statement line 9. Therefore, the cardinality of D^* remains in the same order as that of D. Notice in this respect that, upon reduction, *one* 'join' tuple replaces *two* tuples in D, which reduces the size of the table D^* output by the algorithm.
- However, when inconsistent tuples occur, the statement line 9 adds more than one tuple and statement line 12 inserts tuples resulting from a cross-product.

To find an upper bound of the size of D^*, for every $X \to A$ in $\mathcal{FD}$, let $N(x)$ be the number of different A-values a such that $\Delta \vdash xa$ and x belongs to $inc(X \to A)$. We denote by δ the maximal value of all $N(x)$ for all x in $inc(\mathcal{FD})$; in other words $\delta = \max(\{N(x) \mid x \in inc(\mathcal{FD})\})$. δ is precisely what was earlier referred to as the 'number of conflicts with respect to functional dependencies'.

Given a tuple in D and a functional dependency $X \to A$ in $\mathcal{FD}$, each of the statements line 9 and line 12 generates at most δ tuples. Since several functional dependencies may apply to t, at most $\delta^{|\mathcal{FD}|}$ tuples are generated for the given tuple t. Hence, the number of tuples generated by the statements lines 9 and 12 is in $\mathcal{O}\left(|D|.\delta^{|\mathcal{FD}|}\right)$. We therefore obtain that the size of the table D^* when running Algorithm 2 is in $\mathcal{O}\left(|D|.\left(1+\delta^{|\mathcal{FD}|}\right)\right)$, that is in $\mathcal{O}\left(|D|.\delta^{|\mathcal{FD}|}\right)$.

Since the number of runs of the while-loop on line 4 is at most equal to the number of tuples added into D^*, this number is in $\mathcal{O}\left(|D|.\delta^{|\mathcal{FD}|}\right)$. Since moreover one run of the while-loop is quadratic in the size of D^*, the computational complexity of this while-loop is in $\mathcal{O}\left(|D|^3.\delta^{3.|\mathcal{FD}|}\right)$.

The last point to be mentioned here is that the reduction processing on line 14 is performed through a scan D^* whereby for every t in D^* every sub-tuple of t is removed. Such a processing being quadratic in the size of D^*, the overall computational complexity of Algorithm 2 is in $\mathcal{O}\left(|D|^3.\delta^{3.|\mathcal{FD}|}\right)$.

As the computational complexity of Algorithm 3 is clearly linear in the size of D^*, the global complexity of the computation of inconsistent and true tuples in Δ is as stated just above, and therefore *polynomial in the size of* D. We draw attention on the following important points regarding this complexity result:

1. When the database is consistent, δ is equal to 1, thus yielding a complexity in $\mathcal{O}(|D|^3)$. This result can be shown independently from the above computations as follows: In the case of traditional chase the maximum of nulls in D being bounded by $|U|.|D|$, the number or iterations when running the algorithm is also bounded by $|U|.|D|$. Since the run of one iteration is in $|D|^2$, the overall complexity is in $\mathcal{O}(|U|.|D|^3)$, or in $\mathcal{O}(|D|^3)$, as $|U|$ is independent from $|D|$.
2. The above complexity study can be seen as a *data* complexity study as it relies on an estimation of the size of the chased table D^*. Thus, this study should be further investigated in order to provide more accurate results regarding the estimation of the number of actual tests necessary to the computation of D^*. The results in [10] are likely to be useful for such a more thorough study of this complexity.

5 Consistent Query Answering

5.1 Related Work

The problem of query answering in presence of inconsistencies has motivated important research efforts during the past two decades and is still the subject of current research. As mentioned in the introductory section, the most popular approaches in the literature are based on the notion of 'repair', a repair of $\mathcal{D}$ being intuitively *a consistent database* $\mathcal{R}$ *'as close as possible' to* $\mathcal{D}$; and an answer to a query Q is consistent if it is present in *every repair* $\mathcal{R}$ of $\mathcal{D}$.

However, it has been recognized that generating *all* repairs is difficult to implement - if not unfeasible. This is a well known problem in practice which explains, for instance, why data cleansing is a very important but tedious task in the management of databases and data warehouses [16]. This issue has been thoroughly investigated in [13], where it has been shown that computing repairs of a given relational table in the presence of functional dependencies is either polynomial or APX-complete[2], depending on the form of the functional dependencies. The reader is referred to [1] for theoretical results on the complexity of testing whether $\mathcal{R}$ is a repair of $\mathcal{D}$, when considering a more generic context than we do in this work (more than one table and constraints other than functional dependencies). A Prolog based approach for the generation of repairs can be found in [3].

Dealing with repairs without generating them is thus an important issue, also known as *Consistent Query Answering in Inconsistent Databases*. One of the first

[2] Roughly, APX is the set of NP optimization problems that allow polynomial-time approximation algorithms (source: Wikipedia).

works in this area is [8] and the problem has since been addressed in the context of various database models (mainly the relational model or deductive database models) and under various types of constraints (first order constraints, key constraints, key foreign-key constraints). Seminal papers in this area are [2] and [12, 23], while an overview of works in this area can be found in [6].

The problem considered in all these works can be stated as follows: Given a database $\mathcal{D}$ with integrity constraints $\mathcal{IC}$, assume that $\mathcal{D}$ is inconsistent with respect to $\mathcal{IC}$. Under this assumption, given a query Q against $\mathcal{D}$, what is the *consistent answer* to Q? The usual approach to alleviate the impact of inconsistent data on the answers to a query is to consider that an answer to Q is consistent if it is present in *every repair* $\mathcal{R}$ of $\mathcal{D}$.

Complexity results regarding the computation of the consistent answer have been widely studied in [9]. For example one important case is when $\mathcal{IC}$ consists in having one key constraint per database relation and Q is a conjunctive query containing no self-join (*i.e.,* no join of a relation with itself). In this case computing the consistent answer is polynomial whereas if self-joins occur then the problem is co-NP-complete.

Another important problem in considering repairs is that there are many ways of defining the notion of repair. This is so because there are many ways of defining a distance between two database instances, and there is no consensus as to the 'best' definition of distance. Although the distance based on symmetric difference seems to be the most popular, other distances exist as well based for example on sub-sets, on cardinality, on updates or on homomorphism [22]. Notice in this respect that the results in [13] are set for two distances: one based on sub-sets and one based on updates.

5.2 Consistent Query Answering in our Approach

In our work we do *not* use any notion of repair, thus we avoid the above problem of choosing among all possible ways of defining repairs. Instead, we use set-theoretic semantics for tuples and functional dependencies that allow us to associate each tuple with one truth value among true, false, inconsistent or unknown.

In what follows, we outline the process of consistent query answering in our approach, and then compare it to the approaches based on repairs. In doing so we follow the intuition of the repairs-approach where an answer to a query is consistent if it is present in every repair; and we transpose it in our approach by considering that a tuple is in the consistent answer to the query if its truth value is `true` in the sense of our model.

As usual when dealing with a single table with nulls, a query Q is an SQL-like expression of one of the following two forms:

$$Q : \mathsf{SELECT}\ X \qquad \text{or} \qquad Q : \mathsf{SELECT}\ X\ \mathsf{WHERE}\ \Gamma$$

In either of these forms, X is an attribute list seen as a relation schema, and in the second form, the WHERE clause specifies a selection condition Γ. It should thus be clear that, as in SQL, the where clause in a query is optional. The generic form of a query Q is denoted by $Q : \mathsf{SELECT}\ X\ [\mathsf{WHERE}\ \Gamma]$.

A selection condition Γ is a well formed formula involving the usual connectors $\neg$, $\vee$ and $\wedge$ and built up from atomic boolean comparisons of one of the forms

$A\,\theta\,a$ or $A\,\theta\,A'$, where θ is a comparison predicate, A and A' are attributes in U whose domain elements are comparable through θ, and a is in $dom(A)$.

Moreover, a tuple t satisfies $A\,\theta\,a$ if A is in $sch(t)$ and if $t.A\,\theta\,a$ holds, and t satisfies $A\,\theta\,A'$ if A and A' are in $sch(t)$ and if $t.A\,\theta\,t.A'$ holds. Based on this, determining whether t satisfies Γ done as in [4], that is:

- If $sch(t)$ contains all attributes involved in Γ then determining whether t satisfies Γ follows the rules usual in First Order Logic regarding connectors.
- Otherwise t does *not* satisfy Γ.

For instance, referring to our introductory example, the tuple $t = (k, m)$ such that $sch(t) = KM$ satisfies the condition $(K = k)$ but not the condition $(M = m \vee C = c')$ because C is not in $sch(t)$. Notice also that t does not satisfy the condition $(M = K)$, assuming that m and k are comparable but distinct constants.

Given $\Delta = (D, \mathcal{FD})$, the *answer to Q in Δ* is the set of the restrictions to X of all tuples t in D^* such that $X \subseteq sch(t)$ and such that t satisfies Γ, when present in Q. It follows that answers to queries contain only tuples without nulls.

Now, roughly speaking, the *consistent answer* to Q is the set of all *true* tuples defined over X that satisfy the condition in Q. However, as the following example shows, this rough definition should be carefully stated in particular with regard to the functional dependencies to be taken into account for tuple truth value.

Example 10 In the context of our introductory example, let $\Delta = (D, \mathcal{FD})$ where $\mathcal{FD} = \{Id \rightarrow K, Id \rightarrow C\}$ and where D is displayed in Figure 1. As seen in Example 8, Algorithm 2 returns D^* as shown below and $inc(\mathcal{FD}) = \{inc(Id \rightarrow K), inc(Id \rightarrow C)\}$ where $inc(Id \rightarrow K) = \emptyset$ and $inc(Id \rightarrow C) = \{i_2\}$.

D^*	Id	K	M	C
	i_1	k	m	c
	i_1	k	m'	c
	i_2	k'	m'	c
	i_2	k'	m''	c
	i_2	k'	m'	c'
	i_2	k'	m''	c'
	i_3	k'	m	

It therefore follows from Algorithm 3 that $\mathsf{Inc}(\Delta)$ is defined by:

$$\begin{aligned}\mathsf{Inc}(\Delta) = \ & \{q \mid i_2 \sqsubseteq q \sqsubseteq (i_2, k', m', c)\} \cup \{q \mid i_2 \sqsubseteq q \sqsubseteq (i_2, k', m'', c)\} \cup \\ & \{q \mid i_2 \sqsubseteq q \sqsubseteq (i_2, k', m', c')\} \cup \{q \mid i_2 \sqsubseteq q \sqsubseteq (i_2, k', m'', c')\}\end{aligned}$$

Let Q_1 and Q_2 be two queries (without conditions) as defined below:

$$Q_1 : \mathsf{SELECT}\ Id, K, C \ \text{ and } \ Q_2 : \mathsf{SELECT}\ Id, K, M$$

Projecting the tuples in D^* over the attributes Id, K, C for Q_1 and over Id, K, M for Q_2 produces the tables D_1^* and D_2^* shown below.

D_1^*	Id	K	C
	i_1	k	c
	i_2	k'	c
	i_2	k'	c'

D_2^*	Id	K	M
	i_1	k	m
	i_1	k	m'
	i_2	k'	m'
	i_2	k'	m''
	i_3	k'	m

Since in these two tables, the tuples whose Id-value is i_2, are inconsistent in Δ, it seems justified to exclude them from any consistent answer. In other words,

according to this intuition, the expected consistent answers to Q_1 and Q_2 are respectively $\{(i_1, k, c)\}$ and $\{(i_1, k, m), (i_1, k, m'), (i_3, k', m)\}$.

We explain below why it makes sense to exclude the two tuples in the case of Q_1, whereas the removal in the case of Q_2 is debatable.

1. Regarding Q_1, the tuples (i_2, k', c) and (i_2, k', c') in D_1^* clearly violate $Id \to C$ from $\mathcal{FD}$, and thus can not occur in the *consistent* answer to Q_1.
2. Regarding Q_2 however, no functional dependency is violated by the tuples in D_2^*, and thus, there is no reason for removing any of them when producing the *consistent* answer to Q_2.

Another way of explaining this situation is to notice that, in D^*, the only non satisfied functional dependency is $Id \to C$ and that

1. attributes Id and C occur in the SELECT clause of Q_1, making it necessary to check functional dependency satisfaction;
2. attribute C does not occur in the SELECT clause of Q_2, implying that checking functional dependency satisfaction makes no sense.

Another important point to take into account is the impact of selection conditions on tuple truth value in the answer to a query. To illustrate this point, first notice that, when considering the query Q_1 the only functional dependency to be checked is $Id \to C$, with respect to which the table D_1^* shows inconsistencies regarding i_2. However, let now Q_1' be the query defined by:

$$Q_1' : \mathsf{SELECT}\ Id, K, C\ \mathsf{WHERE}\ C = c'$$

Only the fifth and sixth tuples in D^* satisfy the selection condition and thus, the only possible tuple in the consistent answer to Q_1' is (i_2, k', c'), which alone, trivially satisfies the functional dependency $Id \to C$.

However, the consistency of the answer to Q_1' may seem counter-intuitive, since the tuple (i_2, k', c') is seen as *inconsistent* in the answer to Q_1, where the same attributes are involved. To cope with this counter-intuitive situation, we rather consider that the consistent answer of Q_1' is *empty*, *i.e.,* that consistency has to be checked independently from selection conditions, based only on the functional dependencies involving only attributes from the SELECT clause in the query.

In what follows, we provide the formalism and the definitions to account for these remarks. □

Given a table D over U, a subset X of U and a selection condition Γ, we denote by $\sigma_\Gamma(D)$, $\pi_X(D)$ and $\pi_X(\mathcal{FD})$ the following sets:

- $\sigma_\Gamma(D)$ is the set of all tuples t in D such that t satisfies Γ.
- $\pi_X(D)$ is the set of the restrictions to X of all tuples in D whose schema contains X; that is $\pi_X(D) = \{t \mid (\exists q \in D)(X \subseteq sch(q),\ t = q.X)\}$.
- $\pi_X(\mathcal{FD})$ is the set of all functional dependencies that involve attributes in X only; that is $\pi_X(\mathcal{FD}) = \{(Y \to B) \in \mathcal{FD} \mid YB \subseteq X\}$.

These notation are used in the following definition where the notion of *consistent answer to a query* is introduced.

Definition 7 Given $\Delta = (D, \mathcal{FD})$ and Q : SELECT X [WHERE Γ], let Δ_X be defined by $\Delta_X = (\pi_X(D^*), \pi_X(\mathcal{FD}))$.

The *answer to* Q *in* Δ, denoted by $ans_\Delta(Q)$, is the set $\pi_X(\sigma_\Gamma(D^*))$. Moreover, for every tuple x in $ans_\Delta(Q)$, the *truth value of* x *in* $ans_\Delta(Q)$ is defined by $v_{\Delta_X}(x)$.

Algorithm 4 Consistent answer $ans^+_\Delta(Q)$

Input: A query Q : SELECT X [WHERE Γ], $\Delta^* = (D^*, \mathcal{FD})$ and $inc(\mathcal{FD})$
Output: The set $ans^+_\Delta(Q)$

1: $ans^+_\Delta(Q) := \emptyset$
2: **for all** t in D^* **do**
3: **if** $sch(t)$ contains all attributes in X **then**
4: **if** for every $Y \to B$ in $\pi_X(\mathcal{FD})$, $t.Y$ is not in $inc(Y \to B)$ **then**
5: **if** t satisfies Γ **then**
6: // This test always succeeds if Q involves no selection condition
7: $ans^+_\Delta(Q) := ans^+_\Delta(Q) \cup \{t.X\}$
8: **return** $ans^+_\Delta(Q)$

The *consistent answer* to Q in Δ, denoted by $ans^+_\Delta(Q)$, is the set of all tuples x in $ans_\Delta(Q)$ such that $v_{\Delta_X}(x) = \texttt{true}$. □

It should be noticed that selection conditions are evaluated against the table D^* *before* the projection over X, thus avoiding problematic evaluation of conditions involving attributes not in X. On the other hand we recall that selection conditions are evaluated as usual in standard two-valued logic as earlier explained.

It is also important to notice that, according to Definition 7, given Δ and Q, *two distinct* truth values may be given to a tuple t, namely, its truth value in Δ, *i.e.,* $v_\Delta(t)$, and its truth value in Δ_X, *i.e.,* $v_{\Delta_X}(t)$. Since these truth values are not determined using the *same* set of functional dependencies, they might be distinct.

Referring to Example 10, based on the notation introduced in Definition 7, for $X_1 = Id\,K\,C$, $\pi_{X_1}(\mathcal{FD}) = \mathcal{FD}$, and so, $\Delta_{X_1} = (D^*_1, \mathcal{FD})$. In this case, for every tuple x over X_1, $v_{\Delta_{X_1}}(x) = v_\Delta(x)$. On the other hand, for $X_2 = Id\,K\,M$, $\pi_{X_2}(\mathcal{FD}) = \{Id \to K\}$, and so, $\Delta_{X_2} = (D^*_2, \{Id \to K\})$. Since D^*_2 satisfies $Id \to K$, for $x = (i_2, k', m')$, $v_{\Delta_{X_2}}(x) = \texttt{true}$. However, as x is a super-tuple of i_2, we have $v_\Delta(x) = \texttt{inc}$, showing that $v_{\Delta_{X_2}}(x) \neq v_\Delta(x)$.

The following proposition shows that $ans^+_\Delta(Q)$ is computed from D^* and $inc(\mathcal{FD})$, using Algorithm 4.

Proposition 4 *Given $\Delta = (D, \mathcal{FD})$ and Q : SELECT X [WHERE Γ], Algorithm 4 correctly computes $ans^+_\Delta(Q)$.*

Proof In this proof, denoting by ans the output of Algorithm 4, we prove that $ans = ans^+_\Delta(Q)$. To prove that $ans \subseteq ans^+_\Delta(Q)$, we notice that, by Algorithm 4, every tuple x in ans is the projection over X of a tuple t in D^* satisfying Γ. Thus, x belongs to $\pi_X(\sigma_\Gamma(D^*))$, that is to $ans_\Delta(Q)$. Moreover, since for every t in D^* such that $t.X$ is in ans and every $Y \to B$ in $\pi_X(\mathcal{FD})$, $t.Y$ is not in $inc(Y \to B)$, it holds that $v_{\Delta_X}(x) = \texttt{true}$. It thus follows that x is in $ans^+_\Delta(Q)$.

Conversely, assuming that x is in $ans^+_\Delta(Q)$ implies that x is in $ans_\Delta(Q)$. Hence, D^* contains a tuple t that satisfies Γ and $t.X = x$, meaning that $sch(t)$ contains X and that t satisfies the if-condition on line 5 in Algorithm 4. Moreover, since we also have $v_{\Delta_X}(x) = \texttt{true}$, for every $Y \to B$ in $\pi_X(\mathcal{FD})$, $t.Y$ cannot be in $inc(Y \to B)$. This shows that the if-condition on line 4 in Algorithm 4 is satisfied, and thus that x belongs to ans, which completes the proof. □

Regarding complexity, Proposition 4 shows that, assuming that D^* has been computed, the computation of the consistent answer is *linear* in the size of D^*.

If we assume moreover that $\mathsf{Inc}(\Delta)$ has also been computed, labelling each tuple in $ans^+_\Delta(Q)$ by its truth value in Δ is a relevant option, because it has been seen that the truth value of a tuple t in Δ may be different than the truth value of t in $ans_\Delta(Q)$.

Indeed, knowing that a tuple in the consistent answer, thus having truth value `true` in $ans^+_\Delta(Q)$, has truth value `inc` in the database Δ it comes from, is relevant to a user interested in data quality, as is the case when dealing with data lakes [14]. Investigating further issues related to query answering in our approach, including issues related to data quality is the subject of future work.

Example 11 Running Algorithm 4 with the queries Q_1, Q'_1 and Q_2 as in Example 10, returns $ans^+_\Delta(Q_1) = \{(i_1, k, c)\}$, $ans^+_\Delta(Q'_1) = \emptyset$ and $ans^+_\Delta(Q_2) = \{(i_1, k, m), (i_1, k, m'), (i_2, k', m'), (i_2, k', m''), (i_3, k', m)\}$, as expected in this previous example.

As earlier noticed regarding $ans^+_\Delta(Q_2)$, for $x = (i_2, k', m')$ or $x = (i_2, k', m'')$, we have $v_\Delta(x) \neq v_{\Delta_{X_2}}(x)$. In this case, smart users could find it relevant to be informed of this situation, which can be done by labelling the two tuples (i_2, k', m') and (i_2, k', m'') by `inc`, that is, their truth value in Δ. We notice that this piece of information cannot be provided by any of the existing approaches.

Considering now the query $Q_3 : \mathsf{SELECT}\ M, C\ \mathsf{WHERE}\ K = k'$, Algorithm 4 discards the first two tuples of D^* (because their K-value is not equal to k'), and also the last tuple of D^* (as this tuple has no C-value). When processing the remaining four tuples in D^*, no functional dependency has to be taken care of, and so, we obtain $ans^+_\Delta(Q_3) = \{(m', c), (m', c'), (m'', c), (m'', c')\}$. □

5.3 Comparison with Repair-Based Approaches

Since we are not aware of any work from the literature addressing the issue of consistent query answering in the presence of null values, we propose to compare our approach with existing approaches to consistent query answering in the following context:

- We consider a database $\Delta = (D, \mathcal{FD})$ where D is a *relation* (*i.e.,* a set of tuples defined over U, with no nulls), and $\mathcal{FD}$ is a set of functional dependencies.
- Using our notation, we assume that $\Delta^* = \Delta$, where $\Delta^* = (D^*, \mathcal{FD})$. In other words, for every $X \to A$ in $\mathcal{FD}$ and all $t = qxa$ and $t' = q'xa'$ in D, $t_1 = q'xa$ and $t'_1 = qxa'$ are also in D.
- The repairs are defined based on set-theoretic inclusion as stated in Definition 8. We note in this respect that a more thorough study of repairs relying on various criteria, including for instance updates as in [13,22], lies outside the scope of this paper.

Definition 8 Given $\Delta = (D, \mathcal{FD})$ over universe U and satisfying the conditions set above, a *repair* of Δ is a relation R over U such that: (1) $R \subseteq D$, (2) R satisfies $\mathcal{FD}$, and (3) R is maximal among the relations satisfying (1) and (2). The set of all repairs of Δ is denoted by $Rep(\Delta)$.

Given a query $Q : \mathsf{SELECT}\ X\ [\mathsf{WHERE}\ \Gamma]$, the *repair-based consistent answer to* Q, denoted by $ans^{\mathsf{rep}}_\Delta(Q)$, is defined by: $ans^{\mathsf{rep}}_\Delta(Q) = \bigcap_{R \in Rep(\Delta)} \pi_X(\sigma_\Gamma(R))$. □

According to Definition 8, $ans^{\mathsf{rep}}_{\Delta}(Q)$ is obtained by evaluating Q against each repair and by taking the intersection of all these answers. This is precisely the usual way consistent query answering is defined in repair-based approaches using set-theoretic inclusion [12,23]. The following example illustrates Definition 8 in the context of our introductory example.

Example 12 In order deal with an example satisfying the context as stated above, we consider $\Delta_1 = (D_1, \mathcal{FD})$ where D_1 is shown below and where $\mathcal{FD} = \{Id \rightarrow K, Id \rightarrow C\}$ as in our previous examples. It should be clear that Δ_1 satisfies the conditions stated above, that is D_1 contains no nulls and $\Delta^* = \Delta$. In this case, we have two repairs R_1 and R_2 as shown below.

D_1	Id	K	M	C
	i_1	k	m	c
	i_1	k	m'	c
	i_2	k'	m'	c
	i_2	k'	m'	c'
	i_2	k'	m''	c
	i_2	k'	m''	c'

R_1	Id	K	M	C
	i_1	k	m	c
	i_1	k	m'	c
	i_2	k'	m'	c
	i_2	k'	m''	c

R_2	Id	K	M	C
	i_1	k	m	c
	i_1	k	m'	c
	i_2	k'	m'	c'
	i_2	k'	m''	c'

Recalling that the queries Q_1, Q'_1, Q_2 and Q_3 of Example 10 are defined by

$$Q_1 : \mathsf{SELECT}\ Id, K, C \quad ; \quad Q'_1 : \mathsf{SELECT}\ Id, K, C\ \mathsf{WHERE}\ C = c'$$
$$Q_2 : \mathsf{SELECT}\ Id, K, M \quad ; \quad Q_3 : \mathsf{SELECT}\ M, C\ \mathsf{WHERE}\ K = k',$$

we obtain the following answers based on these repairs:
- $ans^{\mathsf{rep}}_{\Delta_1}(Q_1) = \{(i_1, k, c)\}$;
- $ans^{\mathsf{rep}}_{\Delta_1}(Q'_1) = \emptyset$;
- $ans^{\mathsf{rep}}_{\Delta_1}(Q_2) = \{(i_1, k, m), (i_1, k, m'), (i_2, k', m'), (i_2, k', m'')\}$;
- $ans^{\mathsf{rep}}_{\Delta_1}(Q_3) = \emptyset$. □

As seen earlier, computing consistent answers based on repairs as stated in Definition 8, raises important computational difficulties. Moreover, we show that these answers are 'smaller' with respect to set-theoretic inclusion, than the answers as defined in Definition 7. Before stating this result, we show the following lemma.

Lemma 5 *Given a database $\Delta = (D, \mathcal{FD})$ satisfying the conditions set at the beginning of the section, for every q in D there exists R in $Rep(\Delta)$ such that q belongs to R.*

Proof It should be clear that the relation $\{q\}$ is a subset of D and satisfies $\mathcal{FD}$. Since D is a finite set of tuples, there exists at least one nonempty sub-set R of D, satisfying $\mathcal{FD}$, maximal with respect to set-theoretic inclusion and such that $\{q\} \subseteq R$. Since such a set R satisfies the conditions (1), (2) and (3) of Definition 8, it belongs to $Rep(\Delta)$, and the proof is complete. □

We are now ready to state the following proposition comparing $ans^{\mathsf{rep}}_{\Delta}(Q)$ and $ans^{+}_{\Delta}(Q)$, showing that the former is always a sub-set of the latter.

Proposition 5 *Let $\Delta = (D, \mathcal{FD})$ be a database satisfying the conditions set at the beginning of the section. For every query Q : SELECT X [WHERE Γ], the following holds: $ans^{\mathsf{rep}}_{\Delta}(Q) \subseteq ans^{+}_{\Delta}(Q)$.*

Proof Given t in $ans^{\mathsf{rep}}_{\Delta}(Q)$, by definition of $ans^{\mathsf{rep}}_{\Delta}(Q)$, every R in $Rep(\Delta)$ contains a tuple q such that q satisfies Γ and $q.X = t$. Moreover, as $Rep(\Delta) \subseteq D$ holds, q belongs to D thus to D^*, as $D = D^*$ is assumed.

On the other hand, if we assume that t is not in $ans^{+}_{\Delta}(Q)$, then, by Algorithm 4, for every q in D^*, one of the if-conditions lines 3, 4 or 5 fails. Since, D contains no nulls, so does D^*, and so, q satisfies the first condition. As q satisfies Γ, the test line 5 succeeds and so, the test line 4 fails. Hence, there exists $Y \rightarrow B$ in $\pi_X(\mathcal{FD})$ such that $YB \subseteq X$ and $q.Y \in inc(Y \rightarrow B)$. Therefore, for every tuple q in D^* satisfying Γ and such that $q.X = t$, D^* also contains a tuple q' such that $q.Y = q'.Y = t.Y$ and $q.B \neq q'.B$. Since B is in X, it also follows that $q'.X \neq t$.

As $D = D^*$, q and q' are in D, and so, by Lemma 5, $Rep(\Delta)$ contains a repair R containing q, and a repair R' containing q'. As R' satisfies $Y \rightarrow B$, B is in X, $q.B \neq q'.B$, and $q.X = t$, it is not possible that R' contains q'' such that $q''.X = t$. Thus, t is not in the answer to Q in R'. We therefore obtain a contradiction with the fact that t is in $ans^{\mathsf{rep}}_{\Delta}(Q)$, and the proof is complete. □

The following example illustrates Proposition 5 and the fact that the reverse inclusion does *not* hold in general.

Example 13 Considering Δ_1 as defined in Example 12, the consistent answers to the queries Q_1, Q'_1, Q_2 and Q_3 in Δ_1 are the following:

- $ans^{+}_{\Delta_1}(Q_1) = \{(i_1, k, c)\}$;
- $ans^{+}_{\Delta_1}(Q'_1) = \emptyset$;
- $ans^{+}_{\Delta_1}(Q_2) = \{(i_1, k, m),\ (i_1, k, m'),\ (i_2, k', m'),\ (i_2, k', m'')\}$;
- $ans^{+}_{\Delta_1}(Q_3) = \{(m', c),\ (m', c'),\ (m'', c),\ (m'', c')\}$.

We therefore obtain the following equalities or strict inclusions:

$$ans^{\mathsf{rep}}_{\Delta_1}(Q_1) = ans^{+}_{\Delta}(Q_1) \ \ ; \ \ ans^{\mathsf{rep}}_{\Delta_1}(Q'_1) = ans^{+}_{\Delta}(Q'_1)$$
$$ans^{\mathsf{rep}}_{\Delta_1}(Q_2) = ans^{+}_{\Delta}(Q_2) \ \ ; \ \ ans^{\mathsf{rep}}_{\Delta_1}(Q_3) \subset ans^{+}_{\Delta}(Q_3)$$

Moreover, we recall that when providing consistent answers, our approach additionally allows for pointing to the user possible problematic tuples, namely those tuples that are *inconsistent in Δ, although producing consistent tuples in the answer.* □

6 Concluding Remarks

In this paper we have introduced a novel approach to handle inconsistencies in a table with nulls and functional dependencies. Our approach uses set-theoretic semantics and relies on an extended version of the well known chase procedure to associate every possible tuple with one of the four truth values true, false, inconsistent and unknown. Moreover, we have seen that true and inconsistent tuples can be computed in time polynomial in the size of the input table. We have also seen that our approach applies to consistent query answering and we have shown that it provides larger answers than in the repair-based approaches.

Building upon these results, we currently pursue the following lines of research: (1) applying our approach to the particular but important case of key-foreign key constraints in the context of a star schema or a snow-flake schema; (2) extending

our approach to constraints other than functional dependencies, such as inclusion dependencies as done in [7], (3) investigating the issue of data quality in the framework of our approach; (4) further investigating the computation of false tuples, while extending our approach to the presence of tuples declared as *false*; and (5) last but not least, we plan to address an important issue *not* addressed in this paper: as our algorithms have to be re-run after every change either in the table or in the set of functional dependencies, we plan to design incremental algorithms for query evaluation.

Declarations

Author contributions: The two authors contributed to the study, conception and design. Both read and approved the submitted manuscript.

Funding: No funds, grants, or other support was received for conducting this study.

Financial interests: The authors declare they have no financial interests.

Non-financial interests: The second author is a member of the editorial board of the journal.

Data availability: Data sharing is not applicable to this article as no datasets were generated or analyzed during the current study.

Acknowledgement

The authors wish to sincerely thank the reviewers for their helpful comments and suggestions that helped us improving significantly previous versions of the paper.

References

1. Foto N. Afrati and Phokion G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In Ronald Fagin, editor, *Database Theory - ICDT 2009, 12th International Conference,Proceedings*, volume 361 of *ACM International Conference Proceeding Series*, pages 31–41. ACM, 2009.
2. Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In Victor Vianu and Christos H. Papadimitriou, editors, *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Pennsylvania, USA*, pages 68–79. ACM Press, 1999.
3. Ofer Arieli, Marc Denecker, Bert Van Nuffelen, and Maurice Bruynooghe. Computational methods for database repair by signed formulae. *Ann. Math. Artif. Intell.*, 46(1-2):4–37, 2006.
4. Paolo Atzeni and Edward P. F. Chan. Efficient optimization of simple chase join expressions. *ACM Trans. Database Syst.*, 14(2):212–230, 1989.
5. Nuel D. Belnap. A useful four-valued logic. In J. Michael Dunn and George Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 5–37", isbn="978–94–010–1161–7, Dordrecht, 1977. Springer Netherlands.
6. Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

7. Loreto Bravo and Leopoldo E. Bertossi. Semantically correct query answers in the presence of null values. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Current Trends in Database Technology - EDBT 2006, EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Munich, Germany, March 26-31, 2006, Revised Selected Papers*, volume 4254 of *Lecture Notes in Computer Science*, pages 336–357. Springer, 2006.
8. François Bry. Query answering in information systems with integrity constraints. In Sushil Jajodia, William List, Graeme W. McGregor, and Leon Strous, editors, *Integrity and Internal Control in Information Systems*, volume 109 of *IFIP Conference Proceedings*, pages 113–130. Chapman Hall, 1997.
9. Andrea Calì, Domenico Lembo, and Riccardo Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In Frank Neven, Catriel Beeri, and Tova Milo, editors, *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pages 260–271. ACM, 2003.
10. Stavros S. Cosmadakis, Paris C. Kanellakis, and Nicolas Spyratos. Partition semantics for relations. *J. Comput. Syst. Sci.*, 33(2):203–233, 1986.
11. Ronald Fagin, Alberto O. Mendelzon, and Jeffrey D. Ullman. A simplified universal relation assumption and its properties. *ACM Trans. Database Syst.*, 7(3):343–360, 1982.
12. Paraschos Koutris and Jef Wijsen. Consistent query answering for self-join-free conjunctive queries under primary key constraints. *ACM Trans. Database Syst.*, 42(2):9:1–9:45, 2017.
13. Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. Computing optimal repairs for functional dependencies. *ACM Trans. Database Syst.*, 45(1):4:1–4:46, 2020.
14. Cedrine Madera and Anne Laurent. The next information architecture evolution: The data lake wave. In *Proceedings of the 8th International Conference on Management of Digital EcoSystems*, MEDES, pages 174–180, New York, NY, USA, 2016. ACM.
15. Francesco Parisi and John Grant. Inconsistency measures for relational databases. *CoRR*, abs/1904.03403, 2019.
16. Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.
17. Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pages 55–76, New York, 1977. Plemum Press.
18. Nicolas Spyratos. The partition model: A deductive database model. *ACM Trans. Database Syst.*, 12(1):1–37, 1987.
19. Nicolas Spyratos and Christophe Lécluse. Incorporating functional dependencies in deductive query answering. In *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*, pages 658–664. IEEE Computer Society, 1987.
20. Jeffrey D. Ullman. *Principles of Databases and Knowledge-Base Systems*, volume 1-2. Computer Science Press, 1988.
21. Moshe Y. Vardi. The universal-relation data model for logical independence. *IEEE Softw.*, 5(2):80–85, 1988.
22. Jef Wijsen. Database repairing using updates. *ACM Trans. Database Syst.*, 30(3):722–768, 2005.
23. Jef Wijsen. On the consistent rewriting of conjunctive queries under primary key constraints. *Inf. Syst.*, 34(7):578–601, 2009.

A Proof of Lemma 1

Lemma 1. *For every $\Delta = (D, \mathcal{FD})$, the sequence $(\mu_i)_{i\geq 0}$ has a unique limit μ^* that satisfies that $\mu^* \models \Delta$. Moreover:*

1. *For all a_1 and a_2 in the* same attribute domain *$dom(A)$, if $\mu^*(a_1) \cap \mu^*(a_2) \neq \emptyset$ then there exist $X \to A$ in $\mathcal{FD}$ and x over X such that $\mu^*(x) \neq \emptyset$ and $\mu^*(x) \subseteq \mu^*(a_1) \cap \mu^*(a_2)$.*
2. *For all α and β, $\Delta \vdash (\alpha \sqcap \beta)$ holds if and only if $\mu^*(\alpha) \cap \mu^*(\beta) \neq \emptyset$ holds.*

Proof We recall that the sequence $(\mu_i)_{i\geq 0}$ is defined by the following steps:

1. Associate each tuple t in D with an identifier, $id(t)$, called the *tuple identifier* of t. $id(t)$ is a natural number that identifies t uniquely.
2. Let μ_0 be the mapping defined for every domain constant a by:
   - $\mu_0(a) = \{id(t) \mid t \in D \text{ and } a \sqsubseteq t\}$.
3. Starting with $\mu_i = \mu_0$, while μ_i changes define μ_{i+1} as follows:
   - For every constant a, let $M(a)$ be the union of all $\mu_i(x)$ for which there exists $X \to A$ in $\mathcal{FD}$ such that $sch(x) = X$, a is in $dom(A)$, $\mu_i(xa) \neq \emptyset$ and $\mu_i(x) \not\subseteq \mu_i(a)$.
   - If $M(a) \neq \emptyset$, $\mu_{i+1}(a) = \mu_i(a) \cup M(a)$; otherwise, $\mu_{i+1}(a) = \mu_i(a)$.

The sequence $(\mu_i)_{i\geq 0}$ is increasing in the sense that for every α, $\mu_i(\alpha) \subseteq \mu_{i+1}(\alpha)$, and bounded in the sense that for every α, $\mu_i(\alpha) \subseteq \{id(t) \mid t \in D\}$. Hence the sequence has a unique limit. Moreover, for every t in D, $\mu^*(t) \neq \emptyset$ holds because $id(t)$ always belongs to $\mu^*(t)$, and $\mu^* \models \mathcal{FD}$, because otherwise μ^* would not be the limit of the sequence. Therefore $\mu^* \models \Delta$, which shows the first part of the lemma.

(1) Regarding the first item in the second part of the lemma, we first notice that by definition of μ_0, we have $\mu_0(a_1) \cap \mu_0(a_2) = \emptyset$, because it is not possible that a tuple in D has two distinct values over an attribute.

Since we assume that $\mu^*(a_1) \cap \mu^*(a_2) \neq \emptyset$, there exists $i_0 \geq 0$ such that $\mu_{i_0}(a_1) \cap \mu_{i_0}(a_2) = \emptyset$ and $\mu_{i_0+1}(a_1) \cap \mu_{i_0+1}(a_2) \neq \emptyset$. By definition of the sequence $(\mu_i)_{i\geq 0}$, for $j = 1, 2$, $\mu_{i_0+1}(a_j) = \mu_{i_0}(a_j) \cup M(a_j)$ where $M(a_j)$ is the union of all $\mu_{i_0}(x_j)$ such that $X_j \to A$ is in $\mathcal{FD}$, $\mu_{i_0}(x_j) \cap \mu_{i_0}(a_j) \neq \emptyset$ and $\mu_{i_0}(x_j) \not\subseteq \mu_{i_0}(a_j)$. Hence,

$$\begin{aligned}\mu_{i_0+1}(a_1) \cap \mu_{i_0+1}(a_2) &= (\mu_{i_0}(a_1) \cup M(a_1)) \cap (\mu_{i_0}(a_2) \cup M(a_2)) \\ &= (\mu_{i_0}(a_1) \cap \mu_{i_0}(a_2)) \cup (\mu_{i_0}(a_1) \cap M(a_2)) \ \cup \\ &\qquad (M(a_1) \cap \mu_{i_0}(a_2)) \cup (M(a_1) \cap M(a_2))\end{aligned}$$

Since $\mu_{i_0+1}(a_1) \cap \mu_{i_0+1}(a_2) \neq \emptyset$, at least one of the four terms of the above union is not empty. But since $\mu_{i_0}(a_1) \cap \mu_{i_0}(a_2) = \emptyset$, only the last three cases are investigated below.

(*i*) If $\mu_{i_0}(a_1) \cap M(a_2) \neq \emptyset$, $M(a_2)$ contains x_2 such that $\mu_{i_0}(a_1) \cap \mu_{i_0}(x_2) \neq \emptyset$. Thus, there exists $X_2 \to A$ is in $\mathcal{FD}$ such that $X_2 = sch(x_2)$, $\mu_{i_0}(a_1) \cap \mu_{i_0}(x_2) \neq \emptyset$ and $\mu_{i_0}(a_2) \cap \mu_{i_0}(x_2) \neq \emptyset$. Since both a_1 and a_2 are in $dom(A)$, we have $\mu_{i_0+1}(x_2) \subseteq \mu_{i_0+1}(a_1)$ and $\mu_{i_0+1}(x_2) \subseteq \mu_{i_0+1}(a_2)$. Thus $\mu^*(x_2) \subseteq \mu^*(a_1) \cap \mu^*(a_2)$.

(*ii*) If $\mu_{i_0}(a_2) \cap M(a_1) \neq \emptyset$, it can be shown in a similar way that there exist $X_1 \to A$ is in $\mathcal{FD}$ and x_1 over X_1 such that $\mu^*(x_1) \subseteq \mu^*(a_1) \cap \mu^*(a_2)$. The proof is omitted.

(*iii*) If $M(a_1) \cap M(a_2) \neq \emptyset$, for $j = 1, 2$, $M(a_j)$ contains x_j such that $\mu_{i_0}(x_1) \cap \mu_{i_0}(x_2) \neq \emptyset$. Thus, for $j = 1, 2$, there exist $X_j \to A$ in $\mathcal{FD}$ such that $X_j = sch(x_j)$, $\mu_{i_0}(x_j) \cap \mu_{i_0}(a_j) \neq \emptyset$ and $\mu_{i_0}(x_1) \cap \mu_{i_0}(x_2) \neq \emptyset$. Hence, $\mu_{i_0+1}(x_j) \subseteq \mu_{i_0+1}(a_j)$, for $j = 1, 2$ and $\mu_{i_0+1}(x_1) \cap \mu_{i_0+1}(x_2) \neq \emptyset$. It follows that, when computing μ_{i_0+2}, we obtain the additional inclusions $\mu_{i_0+2}(x_1) \subseteq \mu_{i_0+2}(a_2)$ and $\mu_{i_0+2}(x_2) \subseteq \mu_{i_0+2}(a_1)$, which implies that for $j = 1, 2$, $\mu^*(x_j) \subseteq \mu^*(a_1) \cap \mu^*(a_2)$ holds. This part of the proof is thus complete.

(2) Regarding the second item in the second part of the lemma, assume first that $\Delta \vdash (\alpha \sqcap \beta)$. Since $\mu^* \models \Delta$, we obviously have that $\mu^*(\alpha) \cap \mu^*(\beta) \neq \emptyset$.

Conversely, assuming that $\mu^*(\alpha) \cap \mu^*(\beta) \neq \emptyset$, we show that $\Delta \vdash (\alpha \sqcap \beta)$, that is, for every μ such that $\mu \models \Delta$, $\mu(\alpha) \cap \mu(\beta) \neq \emptyset$. The proof is by induction on the steps of the construction of μ^*, assuming α in $dom(A)$ and β in $dom(B)$.

• The result holds for $i = 0$. Indeed, if $\mu_0(\alpha) \cap \mu_0(\beta) \neq \emptyset$ then there exists u in D such that $\alpha \sqsubseteq u$ and $\beta \sqsubseteq u$. Hence for every μ such that $\mu \models \Delta$, we have $\mu(u) \neq \emptyset$ and $\mu(u) \subseteq \mu(\alpha) \cap \mu(\beta)$, implying that $\mu(\alpha) \cap \mu(\beta) \neq \emptyset$ holds.

• For $i_0 > 0$, assuming that μ_{i_0} satisfies that for all ζ and η such that $\mu_{i_0}(\zeta) \cap \mu_{i_0}(\eta) \neq \emptyset$, we have $\mu(\zeta) \cap \mu(\eta) \neq \emptyset$ for every μ such that $\mu \models \Delta$, we show that the result holds for μ_{i_0+1}.

Indeed, let i_0 such that $\mu_{i_0}(\alpha) \cap \mu_{i_0}(\beta) = \emptyset$ and $\mu_{i_0+1}(\alpha) \cap \mu_{i_0+1}(\beta) \neq \emptyset$. By definition of the sequence $(\mu_i)_{i\geq 0}$, and as in (1) just above, $\mu_{i_0+1}(\alpha) = \mu_{i_0}(\alpha) \cup M(\alpha)$ where $M(\alpha)$ is the union of all $\mu_{i_0}(x)$ such that $X \to A$ is in $\mathcal{FD}$, $\mu_{i_0}(x) \cap \mu_{i_0}(\alpha) \neq \emptyset$ and $\mu_{i_0}(x) \not\subseteq \mu_{i_0}(\alpha)$. Similarly, $\mu_{i_0+1}(\beta) = \mu_{i_0}(\beta) \cup M(\beta)$ where $M(\beta)$ is the union of all $\mu_{i_0}(y)$ such that $Y \to B$ is in $\mathcal{FD}$, $\mu_{i_0}(y) \cap \mu_{i_0}(\beta) \neq \emptyset$ and $\mu_{i_0}(y) \not\subseteq \mu_{i_0}(\beta)$. Thus:

$$\begin{aligned}\mu_{i_0+1}(\alpha) \cap \mu_{i_0+1}(\beta) &= (\mu_{i_0}(\alpha) \cup M(\alpha)) \cap (\mu_{i_0}(\beta) \cup M(\beta)) \\ &= (\mu_{i_0}(\alpha) \cap \mu_{i_0}(\beta)) \cup (\mu_{i_0}(\alpha) \cap M(\beta)) \ \cup \\ &\qquad (M(\alpha) \cap \mu_{i_0}(\beta)) \cup (M(\alpha) \cap M(\beta))\end{aligned}$$

Since $\mu_{i_0+1}(\alpha) \cap \mu_{i_0+1}(\beta) \neq \emptyset$, at least one of the four terms of the above union is non empty. But since $\mu_{i_0}(\alpha) \cap \mu_{i_0}(\beta) = \emptyset$, only the last three cases are investigated below.

(*i*) If $\mu_{i_0}(\alpha) \cap M(\beta) \neq \emptyset$, there exist $Y \to B$ in $\mathcal{FD}$ and y over Y such that $\mu_{i_0}(\alpha) \cap \mu_{i_0}(y) \neq \emptyset$ and $\mu_{i_0}(\beta) \cap \mu_{i_0}(y) \neq \emptyset$. By our induction hypothesis, for every μ such that $\mu \models \Delta$, we have $\mu(\alpha) \cap \mu(y) \neq \emptyset$ and $\mu(y) \subseteq \mu(\beta)$, which implies that $\mu(\alpha) \cap \mu(\beta) \neq \emptyset$.

(*ii*) If $\mu_{i_0}(\beta) \cap M(\alpha) \neq \emptyset$, the case is similar to (*i*) above. The proof is omitted.

(*iii*) If $M(\alpha) \cap M(\beta) \neq \emptyset$, there exist $X \to A$ and $Y \to B$ in $\mathcal{FD}$, x over X and y over Y, such that $\mu_{i_0}(x) \cap \mu_{i_0}(y) \neq \emptyset$, $\mu_{i_0}(\alpha) \cap \mu_{i_0}(x) \neq \emptyset$ and $\mu_{i_0}(\beta) \cap \mu_{i_0}(y) \neq \emptyset$. By our induction hypothesis, for every μ such that $\mu \models \Delta$, we have $\mu(x) \cap \mu(y) \neq \emptyset$, $\mu(x) \subseteq \mu(\alpha)$ and $\mu(y) \subseteq \mu(\beta)$. Hence, $\mu(\alpha) \cap \mu(\beta) \neq \emptyset$ also holds in this case, and the proof is complete. □

B Proof of Lemma 2

Lemma 2. *Let $\Delta = (D, \mathcal{FD})$ and t a tuple. Then Algorithm 1 computes correctly the closure t^+ of t.*

Proof In this proof, we denote by $cl(t)$ the output of Algorithm 1, and we show that $cl(t) = t^+$, that is that $cl(t) \subseteq t^+$ and $t^+ \subseteq cl(t)$ both hold. Before proceeding to these proofs, we draw attention on that for every $\mathcal{T}$-mapping μ such that $\mu(t) \neq \emptyset$, $\mu \models \Delta$ if and only if $\mu \models \Delta_t$, where Δ_t is defined by the statement line 1 in Algorithm 1. Indeed:
• If $\mu \models \Delta_t$ then for every $q \in D_t$ $\mu(q) \neq \emptyset$ and $\mu \models \mathcal{FD}$. Since $D \subset D_t$, $\mu(q) \neq \emptyset$ for every $q \in D$, implying that $\mu \models \Delta$ holds.
• Conversely, if $\mu \models \Delta$ then as $\mu(t)$ is supposed to be nonempty, $\mu(q) \neq \emptyset$ for every q in D_t. Since $\mu \models \mathcal{FD}$ holds, $\mu \models \Delta_t$ also holds.

To first prove that $cl(t) \subseteq t^+$, we consider a $\mathcal{T}$-mapping μ such that $\mu \models \Delta$, and we prove that $\mu(t) \subseteq \mu(a)$ for every a in $cl(t)$. We first observe that if $\mu(t) = \emptyset$ then $\mu(t) \subseteq \mu(\alpha)$ holds for every constant α. Therefore, $\mu(t) \subseteq \mu(a)$ holds.

Now, if $\mu(t) \neq \emptyset$ then $\mu \models \Delta_t$, as shown above. The proof that $\mu(t) \subseteq \mu(a)$ is done by induction on the steps of the execution of Algorithm 1. Denoting by cl^0, cl^1, ... the sequence of the assignments of $cl(t)$ during execution, the following holds for every a in $cl(t)$.
• If a is in cl^0 as computed on line 2, a occurs in t. It is thus clear that $\mu(t) \subseteq \mu(a)$.
• We now assume that, for $j \geq 0$, every α in cl^j is such that $\mu(t) \subseteq \mu(\alpha)$ and we show that this holds for a in cl^{j+1} but not in cl^j. In this case, according to the condition in line 5 of Algorithm 1, there exist $X \to A$ in $\mathcal{FD}$ and x over X such that $\Delta_t \vdash xa$ and for every b in x, $b \in cl^j$. Thus $\mu(x) \cap \mu(a) \neq \emptyset$ (because $\mu \models \Delta_t$) and $\mu(t) \subseteq \mu(b)$ for every b in x (by our induction hypothesis, because b is in cl^j). Hence $\mu(t) \subseteq \mu(x)$ and $\mu(x) \subseteq \mu(a)$ hold, thus implying that $\mu(t) \subseteq \mu(a)$.

As a consequence, we have shown that for every μ such that $\mu \models \Delta$, for every a in $cl(t)$, $\mu(t) \subseteq \mu(a)$. Therefore, by Definition 4, $cl(t) \subseteq t^+$ holds.

Conversely, $t^+ \subseteq cl(t)$ is shown by contraposition: assuming that $a \notin cl(t)$, we prove that $a \notin t^+$. To this end, we exhibit a $\mathcal{T}$-mapping μ_t such that $\mu_t \models \Delta$ and $\mu_t(t) \not\subseteq \mu_t(a)$.

We denote by μ_t^* the $\mathcal{T}$-mapping built up as μ^*, but starting from Δ_t as defined line 1 in Algorithm 1. Thus, $\mu_t^* \models \Delta_t$, and since $\mu_t^*(t) \neq \emptyset$, it has been seen above that $\mu_t^* \models \Delta$.

Thus, if $\mu_t^*(t) \not\subseteq \mu_t^*(a)$ then μ_t^* is the $\mathcal{T}$-mapping we are looking for, and thus, we set $\mu_t = \mu_t^*$. Assuming that $\mu^*(t) \subseteq \mu^*(a)$, let k be a natural number not in $\mu_t^*(\alpha)$ for any α occurring in Δ_t, and let μ_t be the $\mathcal{T}$-mapping defined for every constant α by:
– $\mu_t(\alpha) = \mu_t^*(\alpha) \cup \{k\}$, if $\alpha \in cl(t)$
– $\mu_t(\alpha) = \mu_t^*(\alpha)$, otherwise.
We show that μ_t satisfies that: (1) $\mu_t(t) \not\subseteq \mu_t(a)$ and (2) $\mu_t \models \Delta$.

(1) Since every α in t is in $cl(t)$, k is in $\mu_t(t)$ and since a is not in $cl(t)$, k is not in $cl(a)$. It thus follows that $\mu_t(t) \not\subseteq \mu_t(a)$.

(2) Since for every constant α, $\mu_t^*(\alpha) \subseteq \mu_t(\alpha)$ holds, for every q in D, it holds that $\mu_t^*(q) \subseteq \mu_t(q)$, which implies $\mu_t(q) \neq \emptyset$, because $\mu_t^*(q) \neq \emptyset$ holds as a consequence of $\mu_t^* \models \Delta$.

To prove that $\mu_t \models Y \to B$ for every $Y \to B$ in $\mathcal{FD}$, let y over Y and b in $dom(B)$ such $\mu_t(y) \cap \mu_t(b) \neq \emptyset$. To show that $\mu_t(y) \subseteq \mu_t(b)$, we consider the two cases according to which $\mu_t^*(y) \cap \mu_t^*(b)$ is or not empty.
• If $\mu_t^*(y) \cap \mu_t^*(b) = \emptyset$, then by definition of μ_t, for $\mu_t(y) \cap \mu_t(b)$ to be nonempty, it must be that $\mu_t(y) = \mu_t^*(y) \cup \{k\}$ and $\mu_t(b) = \mu_t^*(b) \cup \{k\}$. Writing y as $\beta_1 \dots \beta_p$, this implies that every β_i $(i = 1, \dots, p)$, and b are in $cl(t)$. Then, as we know that $cl(t) \subseteq t^+$ holds, all these

constants are in t^+, implying that $\mu_t^*(t) \subseteq \mu_t^*(\beta_i)$ $(i = 1, \ldots, p)$ and $\mu_t^*(t) \subseteq \mu_t^*(b)$, because $\mu_t^* \models \Delta$. Since $\mu_t^*(t) \neq \emptyset$, we have $\mu_t^*(y) \cap \mu_t^*(b) \neq \emptyset$, which contradicts our hypothesis that $\mu_t^*(y) \cap \mu_t^*(b) = \emptyset$. This case in thus not possible.
• If $\mu_t^*(y) \cap \mu_t^*(b) \neq \emptyset$, then as $\mu_t^* \models \mathcal{FD}$, $\mu_t^*(y) \subseteq \mu_t^*(b)$ holds, and by Lemma 1 applied to Δ_t, we also have that $\Delta_t \vdash yb$. Since $\mu_t^*(y) \subseteq \mu_t^*(b)$ holds, assuming that $\mu_t(y) \subseteq \mu_t(b)$ does not hold implies that k belongs to $\mu_t(y)$ but not to $\mu_t(b)$. Hence, every β_i $(i = 1, \ldots, p)$ is in $cl(t)$ whereas b is not. This is a contradiction with line 5 of Algorithm 1, where it is stated that β is inserted into $cl(t)$ (because $\Delta_t \vdash yb$ and every β_i $(i = 1, \ldots, p)$ is in $cl(t)$). Thus, $\mu_t(y) \subseteq \mu_t(b)$ holds showing that $\Delta_t \models Y \to B$. The proof is therefore complete. □

C Proof of Lemma 3

Lemma 3. *Algorithm 2 applied to $\Delta = (D, \mathcal{FD})$ always terminates. Moreover, for every tuple t, $\mu^*(t) \neq \emptyset$ holds if and only if t is in $\mathsf{LoCl}(D^*)$.*

Proof The tuples inserted into D^* when running the while-loop line 4 of Algorithm 2 are built up using only constants occurring in Δ. Thus, the number of these tuples is finite, and so, Algorithm 2 terminates.

The proof that for every t in $\mathsf{LoCl}(D^*)$, $\mu^*(t) \neq \emptyset$ holds is conducted by induction on the steps of Algorithm 2. If $(D_k)_{k\geq 0}$ denotes the sequence of the states of D^* during the execution, we first note that since $D_0 = D$, for every t in $\mathsf{LoCl}(D_0)$, $\mu^*(t) \neq \emptyset$ holds.

Assuming now that for $i > 0$, for every t in $\mathsf{LoCl}(D_i)$, $\mu^*(t) \neq \emptyset$, we prove the result for every t in $\mathsf{LoCl}(D_{i+1})$. Indeed, let t' in D_{i+1} such that $t \sqsubseteq t'$. If t' is in D_i, the proof is immediate; we thus now assume that t' is not in D_i, that is that t' occurs in D_{i+1} when running Algorithm 2, that is, there exist $X \to A$ in $\mathcal{FD}$, t_1 and t_2 in D_i such that $t_1.X = t_2.X = x$, $t_1.A = a$ and either (i) $t_2.A$ is not defined or (ii) $t_2.A$ is defined but not equal to $t_1.A$. Writing t_1 as $t_1'xa$, we have the following:

(i) If $t_2.A$ is not defined, then t_2 is written as $t_2'x$ and, according to the statement line 9, t' is of the form $t_2'xa$. By our induction hypothesis, $\mu^*(t_1)$ and $\mu^*(t_2)$ are nonempty, and thus $\mu^*(x) \cap \mu^*(a) \neq \emptyset$. Hence, $\mu^*(x) \subseteq \mu^*(a)$ (because $\mu^* \models X \to A$), and so, $\mu^*(t') = \mu^*(t_2') \cap \mu^*(x) \cap \mu^*(a) = \mu^*(t_2') \cap \mu^*(x)$, showing that $\mu^*(t') = \mu^*(t_2)$. Hence $\mu^*(t') \neq \emptyset$, and so, $\mu^*(t) \neq \emptyset$ also holds, since $\mu^*(t') \subseteq \mu^*(t)$.
(ii) If $t_2.A$ is defined but $t_1.A \neq t_2.A$. for $i = 1, 2$, t_i is written as $t_i'xa_i$ where $a_i = t_i.A$. According to statement line 12, t' is one of the tuples $t_1'xa_2$ or $t_2'xa_1$, and each of these cases can be treated as (i) above.

We therefore have shown that if t is in $\mathsf{LoCl}(D^*)$ as computed by the main loop line 4 of Algorithm 2, then $\mu^*(t) \neq \emptyset$. Since the last loop line 14 does not change this set $\mathsf{LoCl}(D^*)$, this part of the proof is complete.

Conversely, we show that for every t, if $\mu^*(t) \neq \emptyset$ then t is in $\mathsf{LoCl}(D^*)$. The proof is done by induction on the construction of μ^*. By definition of μ_0, it is clear that if $\mu_0(t) \neq \emptyset$ then t is in $\mathsf{LoCl}(D)$ and thus in $\mathsf{LoCl}(D^*)$. Now, if we assume that for every $i > 0$ and every t, if $\mu_i(t) \neq \emptyset$ then t belongs to $\mathsf{LoCl}(D^*)$, we prove that this result holds for μ_{i+1}.

Let t be such that $\mu_i(t) = \emptyset$ and $\mu_{i+1}(t) \neq \emptyset$. For every α, writing $\mu_{i+1}(\alpha)$ as $\mu_i(\alpha) \cup M(\alpha)$, where $M(\alpha)$ is the union of all $\mu_i(x)$ such that x is a tuple over X, where $X \to A \in \mathcal{FD}$, $\alpha \in dom(A)$, $\mu_i(x) \cap \mu_i(\alpha) \neq \emptyset$, and $\mu_i(x) \not\subseteq \mu_i(\alpha)$, we have the following:

$$\begin{aligned}
\mu_{i+1}(t) &= \textstyle\bigcap_{\alpha \sqsubseteq t} \mu_{i+1}(\alpha) \\
&= \textstyle\bigcap_{\alpha \sqsubseteq t} (\mu_i(\alpha) \cup M(\alpha)) && (1) \\
&= \mu_i(t) \cup \Big(\bigcup_{t=t_1t_2} \Big(\mu_i(t_1) \cap \Big(\bigcap_{\beta \sqsubseteq t_2} M(\beta) \Big) \Big) \Big) \cup \Big(\bigcap_{\alpha \sqsubseteq t} M(\alpha) \Big) && (2)
\end{aligned}$$

Equality (2) above is obtained from (1) by applying the distributivity of intersection over union with the convention that $t = t_1t_2$ refers to any split of t into two tuples t_1 and t_2. Assuming $\mu_i(t) = \emptyset$ and $\mu_{i+1}(t) \neq \emptyset$ implies that in Equality (2) either the second or the last term of the union is nonempty.
• If $\bigcup_{t=t_1t_2} \Big(\mu_i(t_1) \cap \Big(\bigcap_{\beta \sqsubseteq t_2} M(\beta) \Big) \Big) \neq \emptyset$, there exist t_1 and t_2 such that $t = t_1t_2$ and $\mu_i(t_1) \cap \Big(\bigcap_{\beta \sqsubseteq t_2} M(\beta) \Big) \neq \emptyset$. Given such a split of t, writing t_2 as $\beta_1 \ldots \beta_p$ implies that, for $k = 1, \ldots, p$, $M(\beta_k)$ contains y_k such that $Y_k \to B_k$ is in $\mathcal{FD}$ and $\mu_i(y_k) \cap \mu_i(\beta_k) \neq \emptyset$. Moreover,

we have that $\mu_i(t_1) \cap \left(\bigcap_{k=1}^{k=p} \mu_i(y_k)\right) \neq \emptyset$. Thus by our induction hypothesis, $\mathsf{LoCl}(D^*)$ contains a tuple of the form $q_1 t_1 y_1 \dots y_p$ and p tuples of the form $q'_k y_k \beta_k$ $(k = 1, \dots, p)$.

Now, given $k = 1, \dots, p$, if $q_1 t_1 y_1 \dots y_p$ is not defined over B_k, $q_1 t_1 y_1 \dots y_p \beta_k$ appears in D^* due to the statement line 9 of Algorithm 2. Assume now that $q_1 t_1 y_1 \dots y_p$ is defined over B_k but with a value different than β_k, say β'_k.

By construction of t_1 and t_2, B_k is not in $sch(t_1)$, and so, B_k is either in $sch(q_1)$ or in Y_i for some $i = 1, \dots, p$. In any case, denoting $sch(q_1 y_1 \dots y_p)$ by Q, we write $q_1 t_1 y_1 \dots y_p$ as $r^k t_1 b'_k$ where $r^k = (q_1 y_1 \dots y_p).(Q \setminus B_k)$. Considering that $r^k t_1 b'_k$ and $q'_k y_k \beta_k$ have the same Y_k-value y_k, the statement line 12 of Algorithm 2 applies and $r^k t_1 \beta_k$ is inserted in D^*. During the subsequent iterations, a similar argument shows that D^* contains a tuple of the form $r t_1 \beta_1 \dots \beta_p$, that is $r t_1 t_2$ or rt. It thus follows that t is in $\mathsf{LoCl}(D^*)$.

• If $\bigcap_{\alpha \sqsubseteq t} M(\alpha) \neq \emptyset$, the same reasoning as above applies considering that t_1 is empty and $t_2 = t$. After the iterations, D^* contains a tuple of the form $r\beta_1 \dots \beta_p$, that is rt. Thus, in this case again, t is in $\mathsf{LoCl}(D^*)$, and the proof is complete. □

D Proof of Proposition 2

Proposition 2. Let $\Delta = (D, \mathcal{FD})$ and t be such that $\Delta \vdash t$. For every tuple q and every a in $dom(A)$ such that $q \sqsubseteq t$ and $a \sqsubseteq t$, a belongs to q^+ if and only if A belongs to Q^+.

Proof Assuming first a in q^+, we show by induction on the steps of Algorithm 1 that A is in Q^+. It is important to notice that since $q \sqsubseteq t$ and $\Delta \vdash t$, $\Delta \vdash q$ holds. Hence when running Algorithm 1 with Δ and q as input, as shown in the proof of Lemma 2, $\mu \models \Delta$ holds if and only if $\mu \models \Delta_q$ holds. Thus, for every tuple τ, $\Delta \vdash \tau$ holds if and only if $\Delta_q \vdash \tau$ holds.

If a is in q^+ because of line 2 in Algorithm 1, then A is in Q, showing that A is in Q^+. If a is inserted in q^+ because of line 5, then there exist $X \to A$ in $\mathcal{FD}$ and x over X such that every b in x belongs to q^+ and $\Delta_q \vdash xa$, that is $\Delta \vdash xa$. Assuming that the proposition holds for every b in x implies that every B in X is in Q^+. Thus, $X \subseteq Q^+$ holds, and so A is in Q^+.

Conversely, let A be in Q^+. If A is in Q, then $q.A = a$, and so, a is in q^+. Let us now assume that A is not in Q, and let us show by induction on the execution of the loop computing Q^+ that a belongs to q^+. Indeed, denoting by Q' the current value of Q^+ when A is inserted in Q^+, there exists $X \to A$ in $\mathcal{FD}$ such that $X \subseteq Q'$. Thus, by our induction hypothesis, every α in $q.X$ is in q^+. Moreover, since $\Delta \vdash t$ and $xa = t.XA$, $\Delta \vdash xa$. Hence, $\Delta_q \vdash xa$, and by the statement line 5 of Algorithm 1, a belongs to q^+. The proof is therefore complete. □

E Proof of Lemma 4

Lemma 4. *Given $\Delta = (D, \mathcal{FD})$, a tuple t is inconsistent in Δ if and only if $t \in \mathsf{Inc}(\Delta)$.*

Proof We note first that for every x in $inc(X \to A)$ there exist $a_1, \dots, a_k$ $(k \geq 2)$ in $dom(A)$ such that for every $i = 1, \dots, k$, $xa_i \in \mathsf{LoCl}(D^*)$, thus such that $\Delta \vdash xa_i$. Therefore, for every $i = 1, \dots, k$, a_i belongs to x^+, and so, $\Delta \vdash (x \preceq a_1 \sqcap \dots \sqcap a_k)$ holds, showing that x is inconsistent in Δ.

We now prove that if q belongs to $\mathsf{Inc}(\Delta)$ then q is inconsistent in Δ. Indeed, by Algorithm 3, there exist t in D^*, $X \to A$ in $\mathcal{FD}$, such that $Q \subseteq T$, $t.Q = q$, $t.X \in inc(X \to A)$, and $X \subseteq Q^+$. Since $\Delta \vdash t$, Proposition 2 applies, showing that for every α in x, α belongs to q^+, where $q = t.Q$. Hence, every a_i in x^+ is also in q^+, and thus for every $i = 1, \dots, k$, $\Delta \vdash (q \preceq a_i)$, implying that q is inconsistent in Δ.

Conversely, if q is inconsistent in Δ, then $\Delta \vdash q$ and $\Delta \nvdash q$. Thus, there exist A in U and a and a' in $dom(A)$ such that $\Delta \vdash (q \preceq a \sqcap a')$, implying that $\Delta \vdash qa$ and $\Delta \vdash qa'$. By Lemma 3, D^* contains two rows t and t' such that $qa \sqsubseteq t$ and $qa' \sqsubseteq t'$. This implies that A can not be in Q because otherwise, we would for instance have $qa = q$ and thus $qa' = qaa'$, which does not define a tuple. Since, by Definition 4, $\Delta \vdash (q \preceq a \sqcap a')$ implies that a and a' are in q^+, by Proposition 2, A is in Q^+. Since A is not in Q, $\mathcal{FD}$ contains $X \to A$ such that $X \subseteq Q^+$. It follows that A is in X^+, $t.XA = xa$ and $t'.XA = xa'$. Therefore x belongs to $inc(X \to A)$.

Summing up, we have found a tuple t in D^* and $X \to A$ in $\mathcal{FD}$ such that $t.X$ belongs to $inc(X \to A)$, $q \sqsubseteq t$ and $X \subseteq Q^+$. It thus follows from line 7 of Algorithm 3 that q belongs to $\mathsf{Inc}(\Delta)$, which completes the proof. □