Achieving Data Privacy through Secrecy Views and Null-Based Virtual Updates

Leopoldo Bertossi and **Lechen Li**
Carleton University, School of Computer Science, Ottawa, Canada

***Abstract*—We may want to keep sensitive information in a relational database hidden from a user or group thereof. We characterize sensitive data as the extensions of secrecy views. The database, before returning the answers to a query posed by a restricted user, is updated to make the secrecy views empty or a single tuple with null values. Then, a query about any of those views returns no meaningful information. Since the database is not supposed to be physically changed for this purpose, the updates are only virtual, and also minimal. Minimality makes sure that query answers, while being privacy preserving, are also maximally informative. The virtual updates are based on null values as used in the SQL standard. We provide the semantics of secrecy views, virtual updates, and secret answers to queries. The different instances resulting from the virtually updates are specified as the models of a logic program with stable model semantics, which becomes the basis for computation of the secret answers.**

***Index Terms*—Data privacy, views, query answering, null values, view updates, answer set programs, database repairs.**

I. Introduction

Database management systems allow for massive storage of data, which can be efficiently accessed and manipulated. However, at the same time, the problems of data privacy are becoming increasingly important and difficult to handle. For example, for commercial or legal reasons, administrators of sensitive information may not want or be allowed to release certain portions of the data. It becomes crucial to address database privacy issues.

In this scenario, certain users should have access to only certain portions of a database. Preferably, what a particular user (or class of them) is allowed or not allowed to access should be specified in a declarative manner. This specification should be used by the database engine when queries are processed and answered. We would expect the database to return answers that do not reveal anything that should be kept protected from a particular user. On the other side and at the same time, the database should return as informative answers as possible once the privacy conditions have been taken care of.

Some recent papers approach data privacy and access control on the basis of *authorization views* [27], [33]. View-based data privacy usually approaches the problem by specifying which views a user *is allowed* to access. For example, when the database receives a query from the user, it checks if the query can be answered using those views alone. More precisely, if the query can be rewritten in terms of the views, for every possible instance [27]. If no *complete rewriting* is possible, the query is rejected. In [33] the problem about the existence of a *conditional* rewriting is investigated, i.e. relative to an instance at hand.

Our approach to the data protection problem is based on specifications of what users are *not* allowed to access through query answers, which is quite natural. Data owners usually have a more clear picture of the data that are sensitive rather than about the data that can be publicly released. Dealing with our problem as "the complement" of the problem formulated in terms of authorization views is not natural, and not necessarily easy, since complements of database views would be involved [20], [21].

According to our approach, the information to be protected is declared as a *secrecy view*, or a collection of them. Their extensions have to be kept secret. Each user or class of them may have associated a set of secrecy views. When a user poses a query to the database, the system virtually updates some of the attribute values on the basis of the secrecy views associated to that user. In this work, we consider updates that modify attribute values through null values, which are commonly used to represent missing or unknown values in incomplete databases. As a consequence, in each of the resulting updated instances, the extension of each of the secrecy views either becomes empty or contains a single tuple showing only null values. Either way, we say that *the secrecy view becomes null*. Then, the original query is posed to the resulting class of updated instances. This amounts to: (a) Posing the query to each instance in the class. (b) Answering it as usual from each of them. (c) Collecting the answers that are shared by all the instances in the class. In this way, the system will return answers to the query that do not reveal the secret data. The next example illustrates the gist of our approach.

Example 1. Consider the following relational database D:

$Marks$	$studentID$	$courseID$	$mark$
	001	01	56
	001	02	90
	002	02	70

The *secrecy view* V_s defined below specifies that a student with her course mark must be kept secret when the mark is less than 60:

Contact author: bertossi@scs.carleton.ca. Faculty Fellow of the IBM CAS, Toronto.

$V_s(sid, cid, mark) \leftarrow Marks(sid, cid, mark), mark < 60$.[1]

The view extension on the given instance is $V_s(D) = \{\langle 001, 01, 56\rangle\}$, which is not null. Now, a user subject to this secrecy view wants to obtain the students' marks, posing the following query:

$$Q(sid, cid, mark) \leftarrow Marks(sid, cid, mark). \quad (1)$$

Through this query the user can obtain the first record $Mark(001, 01, 56)$, which is sensitive information. A way to solve this problem consists in *virtually* updating the base relation according to the definition of the secrecy view, making its extension null. In this way, the secret information, i.e. the extension of the secrecy view, cannot be revealed to the user. Here, in order to protect the tuple $Mark(001, 01, 56)$, the new instance D' below is obtained by virtually updating the original instance, changing the attribute value 56 into NULL.

Marks	*studentID*	*courseID*	*mark*
	001	01	NULL
	001	02	90
	002	02	70

Now, by posing the query about the secrecy view, i.e.

$$Q_1(sid, cid, mark) \leftarrow Marks(sid, cid, mark), \; mark < 60,$$

to D', the user gets an empty answer, i.e. now $V_s(D') = \emptyset$. This is because -in SQL databases- the comparison of NULL with any other value is not evaluated as true.

Now, query (1) will get from D' the first tuple with NULL instead of 56, which can only be -misleadingly, expectedly and intendedly- interpreted by the user as an unknown or missing value for that student in the instance at hand D (not D', which is fully hidden to the user). ■

Notice that, among other elements (cf. end of Section IV), there are two that are crucial for this approach to work: (a) The given database may contain null values and if it has them or not is not known to the user, and (b) The semantics of null values, including the logical operations with them. In this second regard, we can say for the moment and in intuitive terms, that we will base our work on the SQL semantics of nulls, or, more precisely, on a logical reconstruction of this semantics (cf. Sections II-A and II-B).

Hiding sensitive information is one of the concerns. Another one is about still providing as much information as possible to the user. In consequence, the virtual updates have to be minimal in some sense, while still doing their job of protecting data. In the previous example, we might consider virtually deleting the whole tuple $Marks(001, 05, 56)$ to protect secret information, but we may lose some useful information, like the student ID and the course ID. Furthermore, the user should not be able to guess the protected information by combing information obtained from different queries.

As illustrated above, null values will be used to virtually update the database instance. Null values and incomplete databases have received the attention of the database community [32], [29], [18], [23], [1], and may have several possible interpretations, e.g. as a replacement for a real value that is non-existent, missing, unknown, inapplicable, etc. Several formal semantics have been proposed for them. Furthermore, it is possible to consider different, coexisting null values. In this work, we will use a single null value, denoted as above and in the rest of this paper, by *null*. Furthermore, we will treat *null* as the NULL in SQL relational databases.

We want our approach to be applicable to, and implementable on, DBMSs that conform to the SQL Standard, and are used in database practice. We concentrate on that scenario and SQL nulls, leaving for possible future work the necessary modifications for our approach to work with other kinds of null values. Since the SQL standard does not provide a precise, formal semantics for NULL, we define and adopt here a formal, logical reconstruction of conjunctive query answering under SQL nulls (cf. Section II-B). In this direction, we introduce unary predicates $IsNull$ and $IsNotNull$ in logical formulas that are true only when the argument is, resp. is not, the constant NULL. This treatment of null values was first outlined in [9], but here we make it precise. It captures the logics and the semantics of the SQL NULL that are relevant for our work.[2] Including this aspect of nulls in our work is necessary to provide the basic scientific foundations for our approach to privacy.

In this paper, we consider only conjunctive secrecy views and conjunctive queries. The semantics of null-based virtual updates for data privacy that we provide is model-theoretic, in sense that the possible admissible instances after the update, the so-called *secrecy instances*, are defined and characterized. This definition captures the requirement that, on a secrecy instance, the extensions of the secrecy views contain only a tuple with null values or become empty. Furthermore, the secrecy instances do not depart from the original instance by more than necessary to enforce secrecy.

Next, the semantics of *secret answers* to a query is introduced. Those answers are invariant under the class of secrecy instances. More precisely, a ground tuple $\bar{t}$ to a first order query $\mathcal{Q}(\bar{x})$ is a secret answer from instance D if it is an answer to $\mathcal{Q}(\bar{x})$ in every possible secrecy instance for D. Of course, explicitly computing and materializing all the secrecy instances to secretely answer a query is too costly. Ways around this naive approach have to be found.

Actually, we show that the class of secrecy instances, for a given instance D and set of secrecy views $\mathcal{V}^s$, can be captured in terms of a disjunctive logic program with stable model semantics [15], [16]. More precisely, there is a one-to-one correspondence between the secrecy instances and the stable models of the program. As a consequence, the logic programs can be used to: (a) Compactly specify (axiomatize) the class of secrecy instances; and (b) Compute secret answers to queries by running the program on top of the original instance.

[1] We use Datalog notation for view definitions, and sometimes also for queries.

[2] The main issue in [9] was integrity constraint satisfaction in the presence of nulls, for database repair and consistent query answering [3].

Our work has some similarities with that on *database repairs* and *consistent query answering* (CQA) [3], [5]. In that case, the problem is about restoring consistency of a database wrt to a set of integrity constrains by means of minimal updates. The alternative consistent instances that emerge in this way are called *repairs*. They can be used to characterize the consistent data in an inconsistent database as the one that is invariant under the class of repairs. It is possible to specify the repairs of a database by means of disjunctive logic programs with stable model semantics (cf. [5] for references on CQA).

Summarizing, in this paper we make the following contributions: (a) We introduce *secrecy views* to specify what to hide from a given user. (b) We introduce the virtual *secrecy instances* that are obtained by minimally changing attribute values by nulls, to make the secrecy view extensions null. (c) We introduce the *secret answers* as those that are certain for the class of secrecy instances. Those are the answers returned to the user. (d) We establish that this approach works in the sense that the queries about the secrecy view contents always return meaningless answers; and furthermore, the user cannot reconstruct the original instance via secret answers to different queries. (e) We provide a precise logical characterization of query answering in databases with null values *à la* SQL. (f) We specify by means of logic programs the secrecy instances of a database, which allows for skeptical reasoning, and then, certain query answering, directly from the specification. (g) We establish sme connections between secret query answering and CQA in databases.

The structure of the rest of this paper is as follows. In Section II we introduce basic notation and definitions, including the semantics of conjunctive query answering in databases with nulls. In Section III, we introduce the secrecy instances and investigate the properties of secrecy. Section IV presents the notion of secret answer to a query. Section V presents secrecy logic programs. Section VI investigates the connection to database repairs and consistent query answering. Section VII discusses related work. In Section VIII we draw conclusions, and point to future work.

II. Preliminaries

Consider a relational schema $\Sigma = (\mathcal{U}, \mathcal{R}, \mathcal{B})$, where $\mathcal{U}$ is the possibly infinite database domain, with $null \in \mathcal{U}$, $\mathcal{R}$ is a finite set of database predicates, and $\mathcal{B}$ is a finite set of built-in predicates, say $\mathcal{B} = \{=, \neq, >, <\}$. For an n-ary predicate $R \in \mathcal{R}$, $R[i]$ denotes the ith position or attribute of R, with $1 \leq i \leq n$. The schema determines a language $L(\Sigma)$ of first-order (FO) predicate logic, with predicates in $\mathcal{R} \cup \mathcal{B}$ and *constants* in $\mathcal{U}$. A relational *instance* D for schema Σ is a finite set of ground atoms of the form $R(\bar{a})$, with $R \in \mathcal{R}$, and $\bar{a}$ a tuple of constants from $\mathcal{U}$ [1].

A query is a formula $\mathcal{Q}(\bar{x})$ of $L(\Sigma)$, with n free variables $\bar{x}$. $D \models \mathcal{Q}[\bar{c}]$ denotes that instance D makes $\mathcal{Q}$ true with the free variables taking values as in $\bar{c} \in \mathcal{U}^n$. In this case, $\bar{c}$ is an answer to the query. $\mathcal{Q}(D)$ denotes the set of answers to query $\mathcal{Q}$ from D. We will concentrate on *conjunctive queries*, that are $L(\Sigma)$-formulas consisting of a possibly empty prefix of existential quantifiers followed by a conjunction of (database or built-in) atoms.

Example 2. Consider the following database instance D_1:

R	A	B
	a	b
	c	d
	e	$null$

S	B	C
	b	f
	d	g
	$null$	j

For the conjunctive query $\mathcal{Q}_1(x, z) : \exists y(R(x, y) \wedge S(y, z))$, it holds, e.g. $D_1 \models \mathcal{Q}_1[a, f]$. Actually, $\mathcal{Q}_1(D_1) = \{\langle a, f\rangle, \langle c, g\rangle, \langle e, j\rangle\}$. Notice that here, and for the moment, we are treating $null$ as any other constant in the domain. ■

Data will be protected via a fixed set $\mathcal{V}^s$ of secrecy views V_s. They are associated to a particular user or class of them.

Definition 1. A *secrecy view* V_s is defined by a Datalog rule of the form

$$V_s(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n),\ \varphi, \tag{2}$$

with $R_i \in \mathcal{R}, \bar{x} \subseteq \bigcup_i \bar{x}_i$ and $\bar{x}_i$ is a tuple of variables.[3] Formula φ is a conjunction of built-in atoms containing *terms*, i.e. domain constants or variables. ■

We can see that a secrecy view is defined by a conjunctive query with built-in predicates written in $L(\Sigma)$. The conjunctive query associated to the view in (2) is:

$$\mathcal{Q}^{V_s}(\bar{x}) :\ \exists \bar{y}(R_1(\bar{x}_1) \wedge \cdots \wedge R_n(\bar{x}_n) \wedge \varphi), \tag{3}$$

with $\bar{y} = (\bigcup \bar{x}_i) \smallsetminus \bar{x}$. $Conj(\Sigma)$ denotes the class of conjunctive queries of $L(\Sigma)$, and $V_s(D)$ the extension of view V_s computed on instance D for Σ. By definition, $V_s(D) = \mathcal{Q}^{V_s}(D)$.

Example 3. (example 2 cont.) For the given instance D_1, consider the secrecy view defined by $V_s(x) \leftarrow R(x, y), S(y, z)$. Here, the data protected by the view are those that belongs to its extension, namely $V_s(D_1) = \{\langle a\rangle, \langle c\rangle, \langle e\rangle\}$. Sometimes, to emphasize the view predicate involved, we write instead $V_s(D_1) = \{V_s(a), V_s(c), V_s(e)\}$. The corresponding conjunctive query is $\mathcal{Q}^{V_s}(x) : \exists y \exists z(R(x, y) \wedge S(y, z))$. ■

Finally, an *integrity constraint* (IC) is a sentence ψ of $L(\Sigma)$. $D \models \psi$ denotes that instance D satisfies ψ. For a fixed set $\mathcal{I}$ of ICs, we say that D is *consistent* when $D \models \mathcal{I}$, i.e. when D satisfies each element of $\mathcal{I}$.

For both of the notions of query answer and IC satisfaction above we are using the classic concept of satisfaction of predicate logic, denoted with $\models$. According to it, the constant $null$ is treated as any other constant of the database domain. We will use this notion at some places. However, in order to capture the special role of $null$ among those constants, as in SQL databases, we will introduce next a different notion, denoted with $\models_N$. In Example 2, under the new semantics, and due to the participation of $null$ in

[3] We will frequently use Datalog notation for view definitions and queries. When there is no possible confusion, we treat sequences of variables as set of variables. I.e. $x_1 \cdots x_n$ as $\{x_1, \ldots, x_n\}$.

join, the tuple $\langle e, j\rangle$ will not be an answer anymore, i.e. $D_1 \not\models_N \mathcal{Q}_1[e, j]$. The two notions, $\models$ and $\models_N$, will coexist and also be related (cf. Section II-B).

A. Null value semantics: The gist

In [12], Codd proposed a three-valued logic with truth values *true*, *false*, and *unknown* for relational databases with NULL. When a NULL is involved in a comparison operation, the result is *unknown*. This logic has been adopted by the SQL standard, and partially implemented in most common commercial DBMSs (with some variations). As a result, the semantics of NULL in both the SQL standard and the commercial DBMSs is not quite clear; in particular, for IC satisfaction in the presence of NULL.

The semantics for IC satisfaction with NULL introduced in [9], [10] presents a FO semantics for nulls in SQL databases. It is a reconstruction in classical logic of the treatment of NULL in SQL DBs. More precisely, this semantics captures the notion of satisfaction of ICs, and also of query answering for a broad class of queries in relational databases. In the rest of this section, we motivate and sketch some of the elements of the notion of query answer that we will use in the rest of this work. The details can be found in Section II-B. In the following, we assume that there is a single constant, *null*, to represent a null value.

A tuple $\bar{c}$ of elements of $\mathcal{U}$ is an answer to query $\mathcal{Q}(\bar{x})$, denoted $D \models_N \mathcal{Q}(\bar{c})$, if the formula (that represents) $\mathcal{Q}$ is *classically true* when the quantifiers on its *relevant* variables (attributes) run over $(\mathcal{U} \setminus \{null\})$; and those on of the non-relevant variables run over $\mathcal{U}$. The free relevant variables cannot take the value $null$ either. For a precise definition see Section II-B (and also [9], [10]).

Example 4. Consider the instance D_2 and query below:

R	A	B	C
	1	1	1
	2	*null*	*null*
	null	3	3

S	B
	null
	1
	3

$$\mathcal{Q}_2(x): \ \exists y \exists z (R(x, y, z) \wedge S(y) \wedge y > 2). \qquad (4)$$

A variable v (quantified or not) in a conjunctive query is *relevant* if it appears (non-trivially) twice in the formula after the quantifier prefix [9]. Occurrences of the form $v = null$ and $v \neq null$ do not count though. In query (4), the only relevant quantified variable is y, because it participates in a join and a built-in in the quantifier-free matrix of (4). So, there are two reasons for y to be relevant. The only free variable is x, which is not relevant. As for query answers, the only candidate values for x are: $null, 2, 1$. In this case, $null$ is a candidate value because x is a non-relevant variable.

First, $x = null$ is an answer to the query, because the formula $\exists y \exists z (R(x, y, z) \wedge S(y) \wedge y > 2)$ is true in D_2, with a non-null witness value for y and a witness value for z that combined make the (non-quantified) formula true. Namely, $y = 3, z = 3$. So, it holds $D_2 \models_N \mathcal{Q}_2[null]$.

Next, $x = 2$ is not an answer. For this value of x, because the candidate value for y, namely $null$ that accompanies 2 in P, makes the formula $(R(x, y, z) \wedge S(y) \wedge y > 2)$ false. Even if it were true, this value for y would not be allowed.

Finally, $x = 1$ is not an answer, because the only candidate value for y, namely 1, makes the formula false. In consequence, $null$ is the only answer. ■

This notion of query answer coincides with the classic FO semantics for queries and databases without null values [9], [10]. The next example with SQL queries and NULL provides additional intuition and motivation for the formal semantics of Section II-B. Notice the use in logical queries of the new unary predicates *IsNull* and *IsNotNull* that we also formally introduce in Section II-B.

Example 5. Consider the schema $\mathcal{S} = \{R(A, B)\}$ and the instance in the table below. In it NULL is the SQL null. If this instance is stored in an SQL database, we can observe the behavior of the following queries when they are directly translated into SQL and run on an SQL DB:

R	A	B
	a	b
	a	c
	d	NULL
	d	e
	u	u
	v	NULL
	v	r
	NULL	NULL

S	B	C
	b	h
	NULL	s
	l	m

(a) $\mathcal{Q}_1(x, y)$: $R(x, y) \wedge y = null$

SQL:
```
Select * from R
where B = NULL;
```
Result: No tuple

(b) $\mathcal{Q}'_1(x, y)$: $R(x, y) \ \wedge \ IsNull(y)$
SQL: Now uses `IS NULL`
Result: $\langle$d, NULL$\rangle$, $\langle$v, NULL$\rangle$, $\langle$NULL, NULL$\rangle$

(c) $\mathcal{Q}_2(x, y)$: $R(x, y) \wedge y \neq null$
SQL:
```
Select * from R where B <> NULL;
```
Result: No tuple

(d) $\mathcal{Q}'_2(x, y)$: $R(x, y) \wedge IsNotNull(y)$
SQL: Now uses `IS NOT NULL`
Answer: The five expected tuples

(e) $\mathcal{Q}_3(x, y) : R(x, y) \wedge x = y$
SQL:
```
Select * from R where A = B;
```
Result: $\langle$u, u$\rangle$

(f) $\mathcal{Q}_4(x, y) : R(x, y) \wedge x \neq y$
SQL:
```
Select * from R where A <> B;
```
Result: Four tuples: $\langle$a, b$\rangle$, $\langle$a, c$\rangle$, $\langle$d, e$\rangle$, $\langle$v, r$\rangle$

(g) $\mathcal{Q}_5(x, y, x, z) : R(x, y) \wedge R(x, z) \wedge y \neq z$
SQL:
```
Select * from R r1, R r2 where
    r1.A = r2.A and r1.B <> r2.B;
```
Result: $\langle$a, b, a, c$\rangle$, $\langle$a, c, a, b$\rangle$

(h) $\mathcal{Q}_6(x, y, z, t)$: $R(x, y) \wedge S(z, t) \wedge y = z$
SQL:
```
Select * from R r1, S s1
  where r1.B = s1.B;
```
Result: $\langle$a, b, b, h$\rangle$

(i) SQL:
```
Select * from R r1 join S s1
            on r1.B = s1.B;
```
Result:[4] $\langle$a, b, b, h$\rangle$

(j) $\mathcal{Q}_7(x, y, z, t) : R(x, y) \wedge S(z, t) \wedge y \neq z$

[4]The same result is obtained from DBMSs that do not require an explicitly equality together with the join.

SQL: `Select R1.A, R1.B, S1.B, S1.C from R R1, S S1 where R1.B <> S1.B';`
Result: $\langle$a, c, b, h$\rangle$, $\langle$d, e, b, h$\rangle$, $\langle$u, u, b, h$\rangle$, $\langle$v, r, b, h$\rangle$, $\langle$a, b, l, m$\rangle$, $\langle$a, c, l, m$\rangle$, $\langle$d, e, l, m$\rangle$, $\langle$u, u, l, m$\rangle$, $\langle$v, r, l, m$\rangle$ ■

B. Semantics of query answers with nulls

Here we introduce the semantics of FO conjunctive query answering in relational databases with null values.[5] More precisely, in SQL relational databases with a single null value, $null$, that is handled like the SQL `NULL`. The SQL queries are first reconstructed as queries in the FO language $L(\Sigma^{\mathtt{null}})$ associated to $\Sigma^{\mathtt{null}} = (\mathcal{U}, \mathcal{R}, \mathcal{B}^{\mathtt{null}})$, with $\mathcal{B}^{\mathtt{null}} = \mathcal{B} \cup \{IsNull(\cdot), IsNotNull(\cdot)\}$. The last two are new unary built-in predicates that correspond to the SQL predicates `IS NULL` and `IS NOT NULL`, used to check null values. Their intended semantics is as follows (cf. Definition 4): $IsNull(null)$ is true, but $IsNull(c)$ is false for any other constant c in the database domain. And, for any constant $d \in \mathcal{U}$, $IsNotNull(d)$ is true iff $IsNull(d)$ is false.

Introducing these predicates is necessary, because, as shown in Example 5, in the presence of `NULL`, SQL treats `IS NULL` and `IS NOT NULL` differently from $=$ and $\neq$, resp. For example, the queries $\mathcal{Q}(x) : \exists y(R(x, y) \wedge IsNull(y))$ and $\mathcal{Q}'(x) : \exists y(R(x, y) \wedge y = null)$ are both conjunctive queries of $L(\Sigma^{\mathtt{null}})$, but in SQL relational databases, they have different semantics.

In Example 5, each query $\mathcal{Q}$ is defined by the formula ψ on the right-hand side. Below, we will identify the query with its defining FO formula. Furthermore, we exclude from the SQL-like conjunctive queries those like (a) and (c) in Example 5.

Definition 2. (a) The class $Conj^{\mathtt{sql}}(\Sigma^{\mathtt{null}})$ contains all the *conjunctive queries* in $L(\Sigma^{\mathtt{null}})$ of the form

$$\mathcal{Q}(\bar{x}) : \ \exists \bar{y}(A_1(\bar{x}_1) \wedge \cdots \wedge A_n(\bar{x}_n)), \tag{5}$$

where $\bar{y} \subseteq \bigcup_i \bar{x}_i$, $\bar{x} = (\bigcup_i \bar{x}_i) \smallsetminus \bar{y}$, and the A_i are atoms containing any of the predicates in $\mathcal{R} \cup \mathcal{B}^{\mathtt{null}}$ plus terms, i.e. variables or constants in $\mathcal{U}$. Furthermore, those atoms are never of the form $t = null$, $null = t$, $t \neq null, null \neq t$, with t a term, $null$ or not.
(b) With $Conj(\Sigma^{\mathtt{null}})$ we denote the class of all conjunctive queries of the form (5), but without the restrictions on (in)equality atoms imposed on $Conj^{\mathtt{sql}}(\Sigma^{\mathtt{null}})$. ■

The idea here is to force conjunctive queries *à la* SQL, i.e. those in $Conj^{\mathtt{sql}}(\Sigma^{\mathtt{null}})$, that explicitly mention the null value in (in)equalities, to use the built-ins $InNull$ or $IsNotNull$. Notice that the class $Conj(\Sigma^{\mathtt{null}})$ includes both $Conj^{\mathtt{sql}}(\Sigma^{\mathtt{null}})$ and $Conj(\Sigma)$.

Definition 3. Consider a query in $Conj(\Sigma^{\mathtt{null}})$ of the form $\mathcal{Q}(\bar{x}) : \ \exists \bar{y}\psi(\bar{x}, \bar{y})$, with $\exists \bar{y}$ a possibly empty prefix of existential quantifiers, and ψ is a quantifier-free conjunction of atoms. A variable v is *relevant* for $\mathcal{Q}$ [10] if it occurs at least twice in ψ, without considering the atoms $IsNull(v)$, $IsNotNull(v)$, $v \ \theta \ null$, or $null \ \theta \ v$, with $\theta \in \mathcal{B}$. $\mathcal{V}^R(\mathcal{Q})$ denotes the set of relevant variables for $\mathcal{Q}$. ■

For example, for the query $\mathcal{Q}(x) : \exists y(P(x, y, z) \wedge Q(y) \wedge IsNull(y))$, $\mathcal{V}^R(\mathcal{Q}(x)) = \{y\}$, because y is used twice in the subformula $P(x, y, z) \wedge Q(y)$.

As usual in FO logic, we consider assignments from the set, Var, of variables to the underlying database domain $\mathcal{U}$ (that contains constant $null$), i.e. $s : Var \to \mathcal{U}$. Such an assignment can be extended to terms, as $\bar{s}$. It maps every variable x to $s(x)$, and every element c of $\mathcal{U}$ to c. For an assignment s, a variable y and a constant c, $s\frac{y}{c}$ denotes the assignment that coincides with s everywhere, possibly except on y, that takes the value c. Given a formula ψ, $\psi[s]$ denotes the formula obtained from ψ by replacing its free variables by their values according to s.

Now, given a formula (query) χ and a variable assignment function s, we verify if instance D satisfies $\chi[s]$ by assuming that the quantifiers on relevant variables range over $(\mathcal{U} \smallsetminus \{null\})$, and those on non-relevant variables range over $\mathcal{U}$. More precisely, we define, *by induction on* χ, when D satisfies χ with assignment s, denoted $D \models_N \chi[s]$.

Definition 4. Let χ be a query in $Conj(\Sigma^{\mathtt{null}})$, and s an assignment. The pair D, s satisfies χ under the null-semantics, denoted $D \models_N \chi[s]$, exactly in the following cases: (below $t, t_1, \ldots$ are terms; and x, x_1, x_2 variables)
1. (a) $D \models_N IsNull(t)[s]$, with $s(t) = null$. (b) $D \models_N IsNotNull(t)[s]$, with $s(t) \neq null$.
2. $D \models_N (t_1 < t_2)[s]$, with $\bar{s}(t_1) \neq null \neq \bar{s}(t_2)$, and $\bar{s}(t_1) < \bar{s}(t_2)$ (similarly for $>$).[6]
3. (a) $D \models_N (x = c)[s]$, with $s(x) = c \in (\mathcal{U} \smallsetminus \{null\})$. (or symmetrically).[7]
(b) $D \models_N (x_1 = x_2)[s]$, with $s(x_1) = s(x_2) \neq null$.
(c) $D \models_N (c = c)[s]$, with $c \in (\mathcal{U} \smallsetminus \{null\})$.
4. (a) $D \models_N (x \neq c)[s]$, with $null \neq s(x) \neq c \in (\mathcal{U} \smallsetminus \{null\})$. (or symmetrically).
(b) $D \models_N (c_1 \neq c_2)[s]$, with $c_1 \neq c_2$, and $c_1, c_2 \in (\mathcal{U} \smallsetminus \{null\})$.
5. $D \models_N R(t_1, \ldots, t_n)[s]$, with $R \in \mathcal{R}$, and $R(\bar{s}(t_1), \ldots, \bar{s}(t_n)) \in D$.
6. $D \models_N (\alpha \wedge \beta)[s]$, with α, β quantifier-free, $s(y) \neq null$ for every $y \in \mathcal{V}^R(\alpha \wedge \beta)$, and $D \models_N \alpha[s]$ and $D \models_N \beta[s]$.
7. $D \models_N (\exists y \ \alpha)[s]$ when: (a) if $y \in \mathcal{V}^R(\alpha)$, there is c in $(\mathcal{U} \smallsetminus \{null\})$ with $D \models_N \alpha[s\frac{y}{c}]$; or (b) if $y \notin \mathcal{V}^R(\alpha)$, there is c in $\mathcal{U}$ with $D \models_N \alpha[s\frac{y}{c}]$. ■

This semantics can be applied to conjunctive queries in $Conj^{\mathtt{sql}}(\Sigma^{\mathtt{null}})$. The notion of relevant attribute and this semantics of query satisfaction can be both extended to more complex formulas. In particular, they can be applied also to the satisfaction of integrity constraints under SQL null values [10], [9].

Definition 5. [10] Let $\mathcal{Q}(\bar{x}) : \exists \bar{y}\psi(\bar{x}, \bar{y})$ be a conjunctive query in $Conj(\Sigma^{\mathtt{null}})$, with $\bar{x} = x_1, \ldots, x_n$.

[5] This semantics can be extended to a broader class of queries and also to integrity constraint satisfaction. It builds upon a similar and more general semantics first introduced in [9], [10].

[6] Of course, when there is an order relation on $\mathcal{U}$.

[7] Here we use the symbols $=$ and $\neq$ both at the object and the meta levels, but there should not be a confusion since valuations are involved.

(a) A tuple $\langle c_1, \ldots, c_n \rangle \in \mathcal{U}^n$ is an *answer from D under the null query answering semantics* to $\mathcal{Q}$, in short, an N-*answer*, denoted $D \models_N \mathcal{Q}[c_1, \ldots, c_n]$, iff there exists an assignment s such that $s(x_i) = c_i$, for $i = 1, \ldots, n$; and $D \models_N (\exists \bar{y} \psi)[s]$.
(b) $\mathcal{Q}^N(D)$ denotes the set of N-answers to $\mathcal{Q}$ from instance D. Similarly, $V^N(D)$ denotes a view extension according to the N-answer semantics: $V^N(D) = (\mathcal{Q}^V)^N(D)$.
(c) If $\mathcal{Q}$ is a sentence (boolean query), the N-answer is *yes* iff $D \models_N \mathcal{Q}$, and *no*, otherwise. ■

Notice that $D \models_N (\exists \bar{y} \psi)[s]$ in (a) above requires, according to Definition 4, that the variables in the existential prefix $\exists \bar{y}$ that are relevant do not take the value $null$. The free variables x_i in $\mathcal{Q}(\bar{x})$ may take the value $null$ only when they are not relevant in the query. Example 4 illustrates this definition. In it, since the free variable x is not relevant, $\mathcal{Q}_2^N(D_2) = \{\langle null \rangle\}$. Similarly, in Example 2, it holds: $\mathcal{Q}_1^N(D_1) = \{\langle a, f \rangle, \langle c, g \rangle\} \subseteq \mathcal{Q}_1(D_1)$.

Actually, it is easy to prove that, for queries in $Conj(\Sigma^{\mathtt{null}})$, it holds in general: $\mathcal{Q}^N(D) \subseteq \mathcal{Q}(D)$. Furthermore, the N-query answering semantics coincides with classical FO query answering semantics in databases without null values [10], [9]. More precisely, if $null \notin \mathcal{U}$ (and then it does not appear in D or $\mathcal{Q}$ either): $D \models_N \mathcal{Q}[\bar{t}]$ iff $D \models \mathcal{Q}[\bar{t}]$.

Furthermore, every conjunctive query in $Conj(\Sigma^{\mathtt{null}})$ can be syntactically transformed into a new FO query for which the evaluation can be done by treating *null* as any other constant [10], [9]. (A similar transformation will be found in Proposition 1 below.)

More precisely, a conjunctive query $\mathcal{Q}(\bar{x}) \in Conj(\Sigma^{\mathtt{null}})$, i.e. of the form (5), can be rewritten into a classic conjunctive query, as follows:

$$\mathcal{Q}^{rw}(\bar{x}) : \exists \bar{y}(A_1(\bar{x}_1) \wedge \cdots \wedge A_n(\bar{x}_n) \wedge \bigwedge_{v \in \mathcal{V}^R(\mathcal{Q})} v \neq null). \quad (6)$$

It holds: $D \models_N \mathcal{Q}[\bar{c}]$ iff $D \models \mathcal{Q}^{rw}[\bar{c}]$. Here, on the right-hand side, we have classic FO satisfaction, and $null$ is treated as an ordinary constant in the domain. This transformation ensures that relevant variables range over $(\mathcal{U} \setminus \{null\})$. Query $\mathcal{Q}^{rw}(\bar{x})$ belongs to $Conj(\Sigma^{\mathtt{null}})$, and it may contain atoms of the form $IsNull(t)$ or $IsNotNull(t)$. However, replacing them by $t = null$ or $t \neq null$, resp., leads to a query in $Conj(\Sigma)$ that has the same answers as (6) (under the same classic semantics).

Example 6. (example 4 continued) Query $\mathcal{Q}$ in (4) can be rewritten as

$$\mathcal{Q}_2^{rw} : \ \exists y \exists z (P(x, y, z) \wedge Q(y) \wedge y > 2 \wedge y \neq null).$$

We had $D \not\models_N \mathcal{Q}_2[1]$. Now also $D \not\models \exists y \exists z (P(1, y, z) \wedge Q(y) \wedge y > 2 \wedge y \neq null)$ under classic query evaluation, with $null$ treated as an ordinary constant. Similarly, $D \not\models \mathcal{Q}_2^{rw}[2]$ due to the new conjunct $y = null$. Finally, $D \models \mathcal{Q}_2^{rw}[null]$ because $D \models (P(null, 3, 3) \wedge Q(3) \wedge 3 > 2 \wedge 3 \neq null)$. Since $null$ is treated as any other constant, we can compare it with 3. By the *unique names assumption*, it holds $null \neq 3$. ■

Although our framework provides a precise semantics for conjunctive queries in $Conj(\Sigma)$ or $Conj(\Sigma^{\mathtt{null}})$, in both cases possibly containing (in)equalities involving $null$, a usual conjunctive query in SQL should be first translated into a conjunctive query $\mathcal{Q}$ in $Conj^{\mathtt{sql}}(\Sigma^{\mathtt{null}})$ if we want to retain its intended semantics. After that $\mathcal{Q}^{rw}$ can be computed.

III. Secrecy Instances

In this work we will make use of *null* to protect secret information. The basic idea that we develop in this and the next sections is that the extensions of the secrecy views, obtained as query answers, should contain only the tuple with *nulls* or become empty. In this case we will say that *the view is null*.

Definition 6. A query $\mathcal{Q}(\bar{x})$ *is null* on instance D if $\mathcal{Q}^N(D) \subseteq \{\langle null, \ldots, null \rangle\}$ (with the tuple inside with the same length as $\bar{x}$). A view $V(\bar{x})$ is null on D if the query defining it is null on D. ■

Example 7. (example 4 continued) Consider the secrecy view $V_s(x) \leftarrow R(x, y, z),\ S(y),\ y > 2$. Its corresponding FO query $\mathcal{Q}^{V_s}(x)$ in the one in (4), namely:

$$\mathcal{Q}_2(x) : \ \exists y \exists z (R(x, y, z) \wedge S(y) \wedge y > 2).$$

Under the semantics of secrecy in the presence of *null*, we expect the view to be null. This requires the values for attribute A associated with variable x in $\mathcal{Q}_2$ to be $null$, or the values in B associated with variable y in $\mathcal{Q}_2$ to be $null$, or the negation of the comparison to be $true$. These three cases correspond to the three assignments of Example 4. Thus, the view extension is $V_s(D_2) = \{\langle null \rangle\}$, which shows that the view is null on D_2. ■

In this example we are in an ideal situation, in the sense that we did not have to change the instance to obtain a "secret answer". However, this may be an exceptional situation, and we will have to virtually "distort" the given instance by replacing -as few as possible- non-null attribute values by $null$. More generally, since it does not necessarily holds that each secrecy becomes null on an instance D at hand, the view extensions will be obtained from an alternative, possibly virtual, version D' of D that does make each of those views null. In this sense, D' will be an *admissible* instance (cf. Definition 7 below). At the same time, we want D' to stay as close as possible to D (cf. Definition 11 below). Since there may be more that one such instance D', we query all of them simultaneously, and return the *certain answers* [18] (cf. Definition 12 below). Each of the query and view evaluations is done according to the notion of N-answer introduced in Section II-B.

First, we define the instances that make the secrecy views empty or null.

Definition 7. An instance D for schema Σ is *admissible* for a set $\mathcal{V}^s$ of secrecy views of the form (2) if under the N-answer semantics (cf. Definition 5), each $V_s(D)$ is empty or in all its tuples only *null* appears. $Admiss(\mathcal{V}^s)$ denotes the set of admissible instances. ■

As Example 7 shows, D_2 is admissible for the the given view. It also shows that there are some attributes that are particularly relevant for the view to be null, A and B in that case. In the following, we make precise this notion of *secrecy-relevant attribute* (cf. Definition 8(d) below). Before we used (plain) "relevance" associated to variables for query answering under nulls. Not surprisingly, the new notion is based on the previous one. This will allow us to provide an alternative and more operational characterization of secrecy instances (cf. Proposition 1 below).

Definition 8. Consider a view V_s defined as in (2).
(a) For $R \in \mathcal{R}$ in the body of (2) and a term t (i.e. a variable or constant), $pos^R(V_s, t)$ denotes the set of *positions* in R where t appears in the body of V_s's definition.
(b) The set of *combination attributes* for V_s is:
$\mathcal{C}(V_s) = \{R[i] \mid$ for a relevant variable v, $i \in pos^R(V_s, v)\}$.
(c) The set of *secrecy attributes* for V_s is: $\mathcal{S}(V_s) = \{R[i] \mid$ for an x in $V_s(\bar{x})$ in (2), $i \in pos^R(V_s, v)\}$.
(d) The set of *s-relevant attributes*[8] for a secrecy view V_s are those (associated to positions) in the set $\mathcal{A}(V_s) = \mathcal{C}(V_s) \cup \mathcal{S}(V_s)$. ■

Combination attributes for a secrecy view V_s are those involved in joins or built-in predicates (other than built-ins with explicit $null$). Secrecy attributes are those appearing in the head of V_s's definition, and accordingly, collect the query answers, which are expected to be secret. Hence, "secrecy attributes". They correspond to the free variables in the associated query $\mathcal{Q}^{V_s}$.

Example 8. (example 7 continued) Consider again the secrecy view $V_s(x) \leftarrow R(x, y, z),\ S(y),\ y > 2$. Here $\mathcal{C}(V_s) = \{R[2], S[1]\}$, because y is the only relevant variable; and $\mathcal{S}(V_s) = \{R[1]\}$, because x is the only free variable. In consequence, $\mathcal{A}(V_s) = \{R[1],\ S[1], R[2]\}$. Attribute C, i.e. $R[3]$, is not s-relevant. Actually, its value is not relevant to obtain the view extension. ■

The following proposition provides a characterization of admissible instance for a set of secrecy of views in terms of classic FO satisfaction (cf. [24, Proposition 1]). In it we use the notation $D \models \gamma$ for the classic notion of satisfaction by an instance D of FO formula γ, where *null* is treated as any other constant.

Proposition 1. Let $\mathcal{V}^s$ be a set of secrecy views, each of whose elements V_s is of the form (2), and has an expression $\mathcal{Q}^{V_s}(\bar{x}) : \exists\bar{y}(\bigwedge_{i=1}^n R_i(\bar{x}_i) \wedge \varphi)$ as a conjunctive query. For an instance D, $D \in \mathit{Admiss}(\mathcal{V}^s)$ iff for each $V_s \in \mathcal{V}^s$, $D \models$ *Null-*V^s, where *Null-*V^s is the following sentence associated to $\mathcal{Q}^{V_s}$:

$$\overline{\forall}(\bigwedge_{i=1}^{n} R_i(\bar{x}_i) \quad \longrightarrow \bigvee_{v \,\in\, \bigcup_i^n \bar{x}_i \,\cap\, \mathcal{C}(V_s)} v = null \quad \vee \bigwedge_{u \,\in\, \bigcup_i^n \bar{x}_i \,\cap\, \mathcal{S}(V_s)} u = null \quad \vee \neg\varphi). \tag{7}$$ ■

[8]For distinction from the notion of relevant attribute/variable used in Sections II-A and II-B.

In the theorem, $\overline{\forall}$ denotes the universal closure of the formula that follows it; and $v \in (\bigcup_i^n \bar{x}_i \cap \mathcal{C}(V_s))$ indicates that variable v appears in some of the atoms $R_i(\bar{x}_i)$ and in a combination attribute, etc.

Sentence *Null-*V^s in (7) originates in the FO rewriting $(\mathcal{Q}^{V_s})^{rw}$ as in (6) of the query $\mathcal{Q}^{V_s}$ associated to V^s, and the requirement that the latter becomes null on D.

Example 9. (example 8 continued) According to the above definition, in order to check whether the database instance D_2 is admissible, the following must hold:

$$D_2 \models \forall x \forall y \forall z(R(x, y, z) \wedge S(y) \quad \longrightarrow \quad x = null \ \vee \ y = null \ \vee \ y \leq 2).$$

When checking sentence on D_2, $null$ is treated as any other constant. Notice that the values for the non-s-relevant attributes do not matter.

For $x = 1, y = 1$, the antecedent of the implication is satisfied. For these values, the consequent is also satisfied, because $y = 1 < 2$. For $x = 2, y = null$, the consequent is satisfied since y is $null$. For $x = null,\ y = 3$, the antecedent is satisfied. For these values, the consequent is also satisfied, because $null = null$ is true. So, $D_2 \models_N \mathcal{Q}^{V_s}$, and instance D_2 is admissible. ■

The next step consists in selecting from the admissible instances those that are close to the database we are protecting. This requires introducing a notion of distance or an order relationship between instances for a same schema. This would allow us to talk about minimality of change. Since, in order to enforce privacy on an instance D, we will virtually change attribute values by $null$, the comparison of instances has to take this kind of changes and the presence of $null$ in tuples into account. Intuitively, a *secrecy instance* for D will be admissible and also minimally differ from D.

Definition 9. (a) The binary relation $\sqsubset$ on the database domain $\mathcal{U}$, is defined as follows: $c \sqsubset d$ iff $c = null$ and $d \neq null$. Its reflexive closure is $\sqsubseteq$.
(b) For $\bar{t}_1 = \langle c_1, \ldots, c_n\rangle$ and $\bar{t}_2 = \langle d_1, \ldots, d_n\rangle$ in $\mathcal{U}^n$: $\bar{t}_1 \sqsubseteq \bar{t}_2$ iff $c_i \sqsubseteq d_i$ for each $i \in \{1, \ldots, n\}$. Also, $\bar{t}_1 \sqsubset \bar{t}_2$ iff $\bar{t}_1 \sqsubseteq \bar{t}_2$ and $\bar{t}_1 \neq \bar{t}_2$. ■

This partial order relationship $\bar{t}_1 \sqsubseteq \bar{t}_2$ indicates that $\bar{t}_1$ is less or equally informative than $\bar{t}_2$. For example, tuple $(a, null)$ provides less information than tuple (a, b). Then, $(a, null) \sqsubset (a, b)$ holds.

In order to capture the fact that we are just modifying attribute values, but not inserting or deleting tuples, we will assume (sometimes implicitly) that database tuples have *tuple identifiers*. More precisely, each predicate has an additional, first, attribute *ID*, which is a key for the relation, and whose values are taken in $\mathbb{N}$ and not subject to changes. In consequence, tuples in an instance D will be of the form $R(k, \bar{t})$, with $k \in \mathbb{N}$, and $\bar{t} \in \mathcal{U}^n$, and $R \in \mathcal{R}$ is, implicitly, of arity $n + 1$. Below, we will consider only instances D' that are *correlated* to D, i.e. there is a surjective function κ from D to D', such that $\kappa(R(k, \bar{t})) = R(k, \bar{t}')$, for some $\bar{t}'$. This mapping respects the predicate name and the tuple

identifier. We say that D' is D-correlated (via κ). In the rest of this section, D is a fixed instance, the one under privacy protection. We will usually omit tuple identifiers.

Definition 10. (a) For database tuples $R_1(k_1, \bar{t}_1)$, $R_2(k_2, \bar{t}_2)$: $R_1(k_1, \bar{t}_1) \sqsubseteq R_2(k_2, \bar{t}_2)$ iff $R_1 = R_2$, $k_1 = k_2$, and $t_1 \sqsubseteq t_2$.
(b) For instances D_1, D_2: $D_1 \sqsubseteq D_2$ iff for every tuple $R_1(k_1, \bar{t}_1) \in D_1$, there is a tuple $R_2(k_2, \bar{t}_2)$ with $R_2(k, \bar{t}_2) \sqsubseteq R_1(k, \bar{t}_1)$.
(c) For D-correlated instances D_1, D_2: $D_1 \leq_D D_2$ iff: i. $D_1, D_2 \sqsubseteq D$, and ii. $D_2 \sqsubseteq D_1$. As usual, $D_1 <_D D_2$ iff $D_1 \leq_D D_2$, but not $D_2 \leq_D D_1$. ■

Notice that the condition (c)i. for the partial order $\leq_D$ forces D_1 and D_2 to be obtained from D by updating attribute values by $null$. Condition (c)ii. inverts the partial order $\sqsubseteq$ between tuples (and between instances). The reason is that we want secrecy instances to be *minimal* wrt the *set of changes* of attributes values by nulls (as customary for database repairs [5]). Informally, when $D_1 \leq_D D_2$, D_1 is obtained from D, in comparison with D_2, via "less" replacements of values by nulls, and then is close to D.

Definition 11. An instance D_s is a *secrecy instance* for D wrt a set $\mathcal{V}^s$ of secrecy views iff: (a) $D_s \in Admiss(\mathcal{V}^s)$, and (b) D_s is $\leq_D$-minimal in the class of D-correlated database instances that satisfy (a). (I.e. there is no instance D' in that class with $D' <_D D_s$.) $Sec(D, \mathcal{V}^s)$ denotes the set of all the secrecy instances for D wrt $\mathcal{V}^s$. ■

Notice that a secrecy instance nullifies all the secrecy views, is obtained from D by changing attribute values by $null$, and the set of changes is minimal wrt set inclusion.[9]

Example 10. Consider the instance $D = \{P(1,2), R(2,1)\}$ for schema $\mathcal{R} = \{P(A,B),\ R(B,C)\}$. With tuple identifiers (underlined), it takes the form $D = \{P(\underline{1}, 1, 2), R(\underline{1}, 2, 1)\}$. Consider also the *secrecy view*:

$$V_s(x,z) \leftarrow P(x,y), R(y,z),\ y < 3.^{10}$$

D itself is not admissible (it does not nullify the secrecy view), and then it is not a secrecy instance either. Now, consider the following alternative updated instances D_i:

D_1	$\{P(\underline{1}, null, 2), R(\underline{1}, 2, null)\}$
D_2	$\{P(\underline{1}, 1, null), R(\underline{1}, 2, 1)\}$
D_3	$\{P(\underline{1}, 1, 2), R(\underline{1}, null, 1)\}$
D_4	$\{P(\underline{1}, 1, null), R(\underline{1}, null, 1)\}$

For example, for D_1 the set of changes can be identified with the set of changed positions: $U_1 = \{P[1], R[2]\}$ (*ID* has position 0). The D_i are all admissible, that is (cf. (7)):

$$D_i \models \forall x \forall y \forall z (P(x,y) \wedge R(y,z) \longrightarrow (y = null \vee (x = null \wedge z = null) \vee y \geq 3).$$

D_1, D_2, and D_3 are the only three secrecy instances, i.e. they are $\leq_D$-minimal: The sets of changes U_1, $U_2 = \{P[2]\}$, and $U_3 = \{R[1]\}$ are all incomparable under set inclusion. D_4 is not minimal, because $U_4 = \{P[2], R[1]\} \supsetneq U_3$, which is also reflected in the fact that $P(\underline{1}, 1, null) \sqsubset P(\underline{1}, 1, 2)$; and then, $D_3 <_D D_4$. ■

[9] As opposed to minimizing the cardinality of that set. Cf. [5] for a discussion of different forms of "repairs" of databases.

[10] It would be easy to consider tuple ids in queries and view definition, but they do not contribute to the final result and will only complicate the notation. So, we skip tuple ids whenever possible.

IV. Privacy Preserving Query Answers

Now we want to define and compute the *secret answers to queries* from a given database D that is subject to privacy constraints, as represented by the nullification of the secrecy views. They will be defined on the basis of the class of secrecy instances for D. This class will be queried instead of directly querying D. In this sense, we may consider the class of secrecy instances as representing a *logical database*, given through its models. In such a case, the intended answers are those that are true of all the instances in the class, and become the so-called *certain answers* [18].

Definition 12. Let $\mathcal{Q}(\bar{x}) \in Conj(\Sigma^{\mathtt{null}})$. A tuple $\bar{c}$ of constants in $\mathcal{U}$ is a *secret answer* to $\mathcal{Q}$ from D wrt to a set of secrecy views $\mathcal{V}^s$ iff $\bar{c} \in \mathcal{Q}^N(D_s)$ for each $D_s \in Sec(D, \mathcal{V}^s)$. $SA(\mathcal{Q}, D, \mathcal{V}^s)$ denotes the set of all secret answers. ■

Example 11. (example 10 continued). Consider the query $\mathcal{Q}(x,z) : \exists y(P(x,y) \wedge R(y,z) \wedge y < 3)$. According to Definition 5, it holds: $\mathcal{Q}^N(D_1) = \{\langle null, null\rangle\}$, $\mathcal{Q}^N(D_2) = \emptyset$, and $\mathcal{Q}^N(D_3) = \emptyset$. These answers can also be obtained by first rewriting $\mathcal{Q}$, as in (6), into the query $\mathcal{Q}^{rw}(x,z) : \exists y(P(x,y) \wedge R(y,z) \wedge y < 3 \wedge y \neq null)$, which can be evaluated on each of the secrecy instances treating *null* as any other constant.

We obtain $SA(\mathcal{Q}, D, \{V_s\}) = \mathcal{Q}^N(D_1) \cap \mathcal{Q}^N(D_2) \cap \mathcal{Q}^N(D_3) = \emptyset$. This is as expected, because in this example, $\mathcal{Q}$ is $\mathcal{Q}^{V_s}$, the query associated to the secrecy view. ■

The idea behind answering queries from the secrecy instances (SIs) for D is that the answers are still close to those we would have obtained from D (because SIs are maximally close to D). Furthermore, since all the secrecy views become null on the SIs, the answers returned to any query, not necessarily to a secrecy view computation, will take this property into account. In the query answering part we are using a *skeptical or cautious semantics*, that sanctions as true what is simultaneously true in a whole class of models, or instances in our case (the SIs). Now we analyze to what extent this approach does protect the sensitive data. A restricted user may try to pose several queries to obtain sensitive information.

Example 12. Consider instance $D = \{P(1,2),\ P(3,4),\ R(2,1),\ R(3,3)\}$ for schema $\mathcal{R} = \{P(A,B), R(B,C)\}$, and the secrecy view $V_s(x,z) \leftarrow P(x,y), R(y,z)$. In this case, $V_s^N(D) = \{\langle 1,1\rangle\}$. D has the following SIs:

D_1	$\{P(null, 2), P(3,4), R(2, null), R(3,3)\}$
D_2	$\{P(1, null), P(3,4), R(2,1), R(3,3)\}$
D_3	$\{P(1,2), P(3,4), R(null, 1), R(3,3)\}$

The user may pose the queries $\mathcal{Q}_1(x,y) : P(x,y)$ and $\mathcal{Q}_2(x,y) : R(x,y)$, trying to reconstruct D. It holds

$\mathcal{Q}_1^N(D_1) = \{\langle null, 2\rangle,\ \langle 3,4\rangle\}$, $\mathcal{Q}_1^N(D_2) = \{\langle 1, null\rangle,\ \langle 3,4\rangle\}$, $\mathcal{Q}_1^N(D_3) = \{\langle 1,2\rangle, \langle 3,4\rangle\}$. Then, $SA(\mathcal{Q}_1, D, \{V_s\}) = \{\langle 3,4\rangle\}$. Now, $\mathcal{Q}_2^N(D_1) = \{\langle 2, null\rangle, \langle 3,3\rangle\}$, $\mathcal{Q}_2^N(D_2) = \{\langle 2,1\rangle,\ \langle 3,3\rangle\}$, $\mathcal{Q}_2^N(D_3) = \{\langle null, 1\rangle,\ \langle 3,3\rangle\}$. Then, $SA(\mathcal{Q}_2,\ D, \{V_s\}) = \{\langle 3,3\rangle\}$.

By combining the secret answers to $\mathcal{Q}_1$ and $\mathcal{Q}_2$, it is not possible to obtain $V_s^N(D)$. For the user who poses the queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$, the relations look as follows:

P	A	B
	3	4

R	B	C
	3	3

■

Now, we establish in general the impossibility of obtaining the contents of the secrecy views through the use of secret answers to atomic queries (as in the previous example). Open atomic queries are the "broader" queries we may ask; other queries are obtained from them by conjunctive combinations.

Definition 13. Let $\mathcal{V}^s$ be a set of secrecy views V_s. The *secrecy answer instance* for $\mathcal{V}^s$ from D is $D_{\mathcal{V}^s} = \{R(\bar{c}) \mid R \in \mathcal{R} \text{ and } \bar{c} \in SA(R(\bar{x}), D, \mathcal{V}^s)\}$. ■

Here, we are building a database instance by collecting the secret answers (SAs) to all the atomic queries of the form $\mathcal{Q}(\bar{x}) : R(\bar{x})$, with $R \in \mathcal{R}$. This instance has the same schema as D.

Example 13. (example 12 continued) Consider the secrecy view $V_s(x,z) \leftarrow P(x,y),\ R(y,z)$. It holds: $D_{\{V_s\}} = \{P(3,4)\} \cup \{R(3,3)\} = \{P(3,4), R(3,3)\}$. Notice that $V_s^N(D_{\{V_s\}}) = \emptyset = SA(\mathcal{Q}^{V_s}, D, \{V_s\}) = \bigcap_{i=1}^{3}(\mathcal{Q}^{V_s})^N(D_i) = \{\langle null, null\rangle\} \cap \emptyset \cap \emptyset$. ■

Proposition 2. For every V_s of the form (2) in $\mathcal{V}^s$, $SA(\mathcal{Q}^{V_s}, D, \mathcal{V}^s) = V_s(D_{\mathcal{V}^s})$. ■

This proposition tells us that by combining SAs to queries, trying to reconstruct the original instance, we cannot obtain more information that the one provided by the SAs (cf. [24, Proposition 2] for a proof).

The original database D may contain null values, and users have to count on that. A restricted user will receive as query answers the SAs, which are defined and computed through null values. This user could obtain nulls from a query, and hopefully he will not know if they were already in D or were (virtually) introduced for privacy purposes. This is fine and accomplishes our goals. However, as long as the user does not have other kind of information.

Example 14. Consider the instance $D = \{P(1,1)\}$, and the secrecy view $V_s(x) \leftarrow P(x,y), x = 1$. D has only one secrecy instance D_s:

P	A	B
	$null$	1

For the query $\mathcal{Q}(x) : \exists y(P(x,y) \wedge x = 1)$ associated to the secrecy view, the secrecy answer to $\mathcal{Q}(x)$ on D is $\emptyset$. Now, the secrecy answer to $\mathcal{Q}'(x) : \exists y P(x,y)$ is $\{\langle null\rangle\}$. A user who receives this answer will not know if the null value was introduced to protect data.

However, if the user knows from somewhere else that there is an SQL's `NOT NULL` constraint or a key constraint on the first attribute, and that it is satisfied by D, then he will know that the received null was not originally in D. Furthermore, that it is replacing a non-null value. If he also knows that there is exactly one tuple in the relation (a `COUNT` query), and also the secrecy view definition, he will infer that $\langle 1\rangle \in V_s^N(D)$. ■

In summary, for our approach to work, we rely on the following assumptions:

(a) The user interacts via conjunctive query answering with a possibly incomplete database, meaning that the latter may contain null values, and this is something the former is aware of, and can count on (as with databases used in common practice). In this way, if a query returns answers with null values, the user will not know if they were originally in the database or were introduced for protection at query answering time.

(b) The queries request data, as opposed to schema elements, like integrity constraints and view definitions. Knowing the ICs (and about their satisfaction) in combination with query answers could easily expose the data protection policy. The most clear example is the one of a `NOT NULL` SQL constraint, when we see nulls where there should not be any.

(c) In particular, the user does not know the secrecy view definitions. Knowing them would basically reveal the data that is being protected and how.

These assumptions are realistic and make sense in many scenarios, for example, when the database is being accessed through the web, without direct interaction with the DBMS via complex SQL queries, or through an ontology that offers a limited interaction layer. After all, protecting data may require additional measures, like withholding from certain users certain information that is, most likely, not crucial for many applications. From these assumptions and Proposition 2, we can conclude that the user cannot obtain information about the secrecy views through a combination of SAs to conjunctive queries. Therefore, there is not leakage of sensitive information.

V. Secrecy Instances and Logic Programs

The updates leading to the secrecy instances (SIs) should not physically change the database. Also, different users may be restricted by different secrecy views. Rather, the possibly several SIs have to be virtual, and used mainly as an auxiliary notion for the secret answer semantics. We expect be able to avoid computing all the SIs, materializing them, and then cautiously querying the class they form. We would rather stick to the original instance, and use it as it is to obtain the secret answers.

One way to approach this problem is via query rewriting. Ideally, a query $\mathcal{Q}$ posed to D and expecting secret answers should be rewritten into another query $\mathcal{Q}'$. This new query would be posed to D, and the usual answers returned by D to $\mathcal{Q}'$ should be the secret answers to $\mathcal{Q}$. We would like $\mathcal{Q}'$ to be still a simple query, that can be easily evaluated. For example, if $\mathcal{Q}'$ is FO, it can be evaluated in polynomial

time in data. However, this possibility is restricted by the intrinsic complexity of the problem of computing or deciding secret answers, which is likely to be higher than polynomial time in data (cf. Section VI). In consequence, $\mathcal{Q}'$ may not even a FO query, let alone conjunctive.

An alternative approach is to specify the SIs in a compact manner, by means of a logical theory, and do reasoning from that theory, which is in line with skeptical query answering. This will not decrease a possibly high intrinsic complexity, but can be much more efficient than computing all the secrecy instances and querying them in turns. Wrt the kind of logical specification needed, we can see that secret query answering (SQA) is a *non-monotonic* process.

Example 15. Consider $D = \{P(a)\}$, the secrecy view $V(x) \leftarrow P(x), R(x)$, and the query $\mathcal{Q}: \ Ans(x) \leftarrow P(x)$. Here, $V(D) = \emptyset$, and then, D itself is its only SI. Therefore, $SA(\mathcal{Q}, D, \{V\}) = \{\langle a \rangle\}$.

Let us update D to $D_1 = \{P(a), R(a)\}$. Now, $V(D_1) = \{\langle a \rangle\}$. The SIs for D_1 are: $D_1' = \{P(null), R(a)\}$ and $D_1'' = \{P(a), R(null)\}$. It holds, $\mathcal{Q}(D_1') = \{\langle null \rangle\}$ and $\mathcal{Q}(D_1'') = \{\langle a \rangle\}$. Then, $SA(\mathcal{Q}, D_1, \{V\}) = \emptyset$. The previous secret answer is lost. ■

The non-monotonicity of SQA requires a non-monotonic formalism to logically specify the SIs of a given instance. Actually, they can be specified as the stable models of a disjunctive logic program, a so-called *secrecy program*.

Secrecy programs use annotation constants with the intended, informal semantics shown in the table below. More precisely, for each database predicate $R \in \mathcal{R}$, we introduce a copy of it with an extra, final attribute (or argument) that contains an annotation constant. So, a tuple of the form $R(\bar{t})$ would become an annotated atom of the form $R(\bar{t}, \mathbf{a})$.[11] The annotation constants are used to keep track of virtual updates, i.e. of old and new tuples:

Annotation	Atom	The tuple $R(\bar{a})$ ...
$\mathbf{u}$	$R(\bar{a}', \mathbf{u})$	is being updated
$\mathbf{bu}$	$R(\bar{a}, \mathbf{bu})$	has been updated
$\mathbf{t}$	$R(\bar{a}, \mathbf{t})$	is new or old
$\mathbf{s}$	$R(\bar{a}, \mathbf{s})$	stays in the secrecy instance

In $R(\bar{a}, \mathbf{bu})$, annotation $\mathbf{bu}$ means that the atom $R(\bar{a})$ has already been updated, and $\mathbf{u}$ should appear in the new, updated atom, say $R(\bar{a}', \mathbf{u})$. For example, consider a tuple $R(a, b) \in D$. A new tuple $R(a, null)$ is obtained by updating b into $null$. Therefore, $R(a, b, \mathbf{bu})$ denotes the old atom before updating, while $P(a, null, \mathbf{u})$ denotes the new atom after the update.

The logic program uses these annotations to go through different steps, until its stable models are computed. Finally, the atoms needed to build an SI are read off by restricting a model of the program to atoms with the annotation $\mathbf{s}$. As expected, the official semantics of the annotations is captured through the logic program; the table above is just for motivation. In Section V-A we provide the general form of $\Pi(D, \mathcal{V}^s)$, the *secrecy logic program* that specifies the SIs for an instance D subject to set of secrecy views $\mathcal{V}^s$. The following example illustrates the main ideas and issues.

[11] We should use a new predicate, e.g. R', but to keep the notation simple, we will reuse the predicate. We also omit tuple ids.

Example 16. (example 10 continued) Consider $\mathcal{R} = \{P(A, B), \ R(B, C)\}$, $D = \{P(1, 2), \ R(2, 1)\}$ and the secrecy view $V_s(x, z) \leftarrow P(x, y), R(y, z), y < 3$.

The secrecy instance program $\Pi(D, \{V_s\})$ is as follows:

1. $P(1, 2). \quad R(2, 1).$ (initial database)

2. $P(null, y, \mathbf{u}) \vee P(x, null, \mathbf{u}) \vee R(null, z, \mathbf{u})$
$\leftarrow P(x, y, \mathbf{t}), \ R(y, z, \mathbf{t}), y < 3, \ y \neq null, aux(x, z).$
$R(y, null, \mathbf{u}) \vee P(x, null, \mathbf{u}) \vee R(null, z, \mathbf{u})$
$\leftarrow P(x, y, \mathbf{t}), \ R(y, z, \mathbf{t}), y < 3, \ y \neq null, aux(x, z).$
$aux(x, z) \leftarrow P(x, y, \mathbf{t}), \ R(y, z, \mathbf{t}), y < 3, x \neq null.$

$aux(x, z) \leftarrow P(x, y, \mathbf{t}), \ R(y, z, \mathbf{t}), y < 3, z \neq null.$

3. $P(x, y, \mathbf{bu}) \leftarrow P(x, y, \mathbf{t}), R(y, z, \mathbf{t}), y < 3, y \neq null,$
$aux(x, z), P(null, y, \mathbf{u}), x \neq null.$
$R(y, z, \mathbf{bu}) \leftarrow P(x, y, \mathbf{t}), R(y, z, \mathbf{t}), y < 3, y \neq null,$
$aux(x, z), R(y, null, \mathbf{u}), z \neq null.$
$P(x, y, \mathbf{bu}) \leftarrow P(x, y, \mathbf{t}), R(y, z, \mathbf{t}), y < 3, y \neq null,$
$aux(x, z), P(x, null, \mathbf{u}).$
$R(y, z, \mathbf{bu}) \leftarrow P(x, y, \mathbf{t}), R(y, z, \mathbf{t}), y < 3, y \neq null,$
$aux(x, z), R(null, z, \mathbf{u}).$

4. $P(x, y, \mathbf{t}) \leftarrow P(x, y). \qquad P(x, y, \mathbf{t}) \leftarrow P(x, y, \mathbf{u}).$
$R(x, y, \mathbf{t}) \leftarrow R(x, y). \qquad R(x, y, \mathbf{t}) \leftarrow R(x, y, \mathbf{u}).$

5. $P(x, y, \mathbf{s}) \leftarrow P(x, y, \mathbf{t}), \ not \ P(x, y, \mathbf{bu}).$
$R(x, y, \mathbf{s}) \leftarrow R(x, y, \mathbf{t}), \ not \ R(x, y, \mathbf{bu}).$

The facts in 1. belong to the initial instance D, and become annotated right away with $\mathbf{t}$ by rules 4. The most important rules of the program are those in 2. and 3. They enforce the update semantics of secrecy in the presence of $null$ and using $null$. Rules in 2. capture in the body the violation of secrecy (i.e. a non-null view contents); and in the head, the intended way of restoring secrecy: We can either update a combination of (combination) attributes or single secrecy attributes with $null$. In this example, we need to update, with $null$, values in attribute B or in attributes A and C, simultaneously.

Since disjunctive programs do not allow conjunctions in the head, the intended head $(P(null, z) \wedge P(y, null)) \vee P(x, null) \vee Q(null, z) \leftarrow Body$ is represented by means of two rules, as in 2.: $P(null, z) \vee P(x, null) \vee Q(null, z) \leftarrow Body$ and $P(y, null) \vee P(x, null) \vee Q(null, z) \leftarrow Body$.

Furthermore, we need to restore secrecy only if the given database is not already a secrecy instance, which happens when the combination attribute B is not null, the secrecy attributes A and C are not null, and formula φ is true. Predicate $aux(x, z)$ defined in 2. captures the condition $not \ (x \neq null \wedge z \neq null)$.

The rules in 3. collect the tuples in the database that have already been updated and (virtually) no longer exist in the database. Rules 4. annotate the original the atoms and also the new version of updated atoms. Rules in 5. collect the

tuples that stay in the final state of the updated database: They are original or new, but have never been updated. ■

The secrecy instances are in one-to-one correspondence with the restrictions to $\mathbf{s}$-annotated atoms of the stable models of $\Pi(D, \mathcal{V}^s)$.[12]

Example 17. (example 16 continued) The program has three stable models (the facts in 1. are omitted):

$M_1 = \{P(1,2,\mathbf{t}),\ R(2,1,\mathbf{t}),\ aux(1,1), \underline{P(1,2,\mathbf{s})},$
$R(2,1,\mathbf{bu}), R(null,1,\mathbf{u}), R(null,1,\mathbf{t}), \underline{R(null,1,\mathbf{s})}\}.$
$M_2 = \{P(1,2,\mathbf{t}),\ R(2,1,\mathbf{t}),\ aux(1,1), P(1,2,\mathbf{bu}),$
$\underline{R(2,1,\mathbf{s})}, P(1,null,\mathbf{u}), P(1,null,\mathbf{t}), \underline{P(1,null,\mathbf{s})}\}.$
$M_3 = \{P(1,2,\mathbf{t}),\ R(2,1,\mathbf{t}),\ aux(1,1), P(1,2,\mathbf{bu}),$
$R(2,1,\mathbf{bu}), P(null,2,\mathbf{u}), R(2,null,\mathbf{u}), P(null,2,\mathbf{t}),$
$R(2,null,\mathbf{t}), aux(1,null), aux(null,1), \underline{P(null,2,\mathbf{s})},$
$\underline{R(2,null,\mathbf{s})}\}.$

The secrecy instances are built by selecting the underlined atoms, obtaining: $D_1 = \{P(1,2),\ R(null,1)\}$, $D_2 = \{P(1,null), R(2,1)\}$, and $D_3 = \{P(null,2), R(2,null)\}$. They coincide with those in Example 10. ■

In order to compute secret answers to a query, it is not necessary to explicitly compute all the stable models. Instead, the query can be posed directly on top of the program and answered according to the skeptical semantics. This will return the secret answers to the query. The query has to be formulated as a top-layer program, with $\mathbf{s}$-annotated atoms, that are those that affect the query. A system like *DLV* can be used. It computes the disjunctive stable-model semantics, with an interface to commercial DBMSs [22].

Example 18. (example 17 continued) We want the secret answers to the conjunctive query

$$\mathcal{Q}(x,z) : \exists y(P(x,y) \wedge R(y,z) \wedge y < 3).$$

This requires first rewriting it, as in (6), into $\mathcal{Q}^{rw}(x,y) : \exists y(P(x,y) \wedge R(y,z) \wedge y < 3 \wedge y \neq null)$. This new query can be evaluated against instances with *null* treated as any other constant. In its turn, $\mathcal{Q}^{rw}$ is transformed into a query program with all the database atoms using annotation $\mathbf{s}$:

$$Ans(x,z) \leftarrow P(x,y,\mathbf{s}), R(y,z,\mathbf{s}),\ y < 3,\ y \neq null.$$

This one is evaluated in combination with the secrecy program in Example 16, under the skeptical semantics. In this evaluation, *null* is treated as an ordinary constant. ■

A. The general secrecy logic program

To provide the general form of secrecy logic program, we need to introduce some notation first. We recall that our view definitions are of the form

$$V_s(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n),\ \varphi. \tag{8}$$

[12]The proof of this claim is rather long, and is similar in spirit to the proof of the fact that database repairs wrt integrity constraints [3] can be specified by means of disjunctive logic programs with stable model semantics (cf. [10], [2]).

Some of the variables[13]in atoms in the body of the definitions are relevant, as in Definition 8, and their values will be replaced by $null$. As expected, and illustrated in Example 10, those atoms and variables play a crucial role in the program.

For an atom of the form $R(\bar{x})$ and variables $\bar{y} \subseteq \bar{x}$, $R(\bar{x})\frac{\bar{y}}{null}$ denotes $R(\bar{x})$ with all the variables in $\bar{y}$ replaced by $null$. In reference to (8), with this notation, we define:

$$\begin{aligned}\mathcal{CP}(V_s) &= \{R_i(\bar{x}_i)\frac{\bar{y}}{null} \mid R_i(\bar{x}_i) \text{ is in body of (8)},\\ &\qquad \bar{y} = \{y_1, ..., y_n\} \subseteq \bar{x},\ \text{and } y_i \in \mathcal{C}(V_s))\}.\\ \mathcal{SP}(V_s) &= \{R_i(\bar{x}_i)\frac{\bar{y}}{null} \mid R_i(\bar{x}_i) \text{ is in body of (8)},\\ &\qquad \bar{y} = \{y_1, ..., y_n\} \subseteq \bar{x},\ \text{and } y_i \in \mathcal{S}(V_s))\}.\end{aligned}$$

For the sets of predicate positions, $\mathcal{C}(V_s)$ and $\mathcal{S}(V_s)$, see Definition 8. The atom sets $\mathcal{CP}(V_s)$ and $\mathcal{SP}(V_s)$ will be used in the head of the disjunctive rules that change some relevant attribute values into nulls (rules 2. in Example 10).

Example 19. For the secrecy view $V_s(x,z,w) \leftarrow P(x,y), Q(y,z,w)$, it holds: $\mathcal{C}(V_s) = \{P[2], Q[1]\}$ and $\mathcal{S}(V_s) = \{P[1], Q[2], Q[3]\}$. Thus, $\mathcal{CP}(V_s) = \{P(x,null), Q(null,z,w)\}$, and $\mathcal{SP}(V_s) = \{P(null,y), Q(y,null,null)\}$. ■

Given a database instance D, a set $\mathcal{V}^s$ of secrecy views V_ss, each of them of the form (8), the secrecy program $\Pi(D, \mathcal{V}^s)$ contains the following rules:

1. Facts: $R(\bar{c}, \mathbf{t})$ for each atom $R(\bar{c}) \in D$.

2. For every V_s of the form (8), if $\mathcal{SP}(V_s) = \{R^1(\bar{x}_1), \ldots, R^a(\bar{x}_a)\}$, and $\mathcal{CP}(V_s) = \{R^1(\bar{x}_1), ..., R^b(\bar{x}_b)\}$, then the program contains the rules:
(a) If $\mathcal{S}(V_s) \cap \mathcal{C}(V_s) \neq \emptyset$, the rule:

$$\bigvee_{R^c \in \mathcal{CP}(V_s)} R^c(\bar{x}_c, \mathbf{u}) \ \leftarrow\ \textstyle\bigwedge_{i=1}^n R_i(\bar{x}_i, \mathbf{t}),\ \varphi,\ \bigwedge_{v_l \in \mathcal{C}(V_s)} v_l \neq null.$$

(b) If $\mathcal{S}(V_s) \cap \mathcal{C}(V_s) = \emptyset$, for each $R^d \in \mathcal{SP}(V_s)$, $1 \leq d \leq a$, the rule:

$$R^d(\bar{x}_d, \mathbf{u}) \vee \bigvee_{R^c \in \mathcal{CP}(V_s)} R^c(\bar{x}_c, \mathbf{u}) \ \leftarrow\ \textstyle\bigwedge_{i=1}^n R_i(\bar{x}_i, \mathbf{t}),\ \varphi,\ \bigwedge_{v_l \in \mathcal{C}(V_s)} v_l \neq null,\ aux_{V_s}(\bar{x}).$$

Plus rules defining the auxiliary predicates: If $\mathcal{S}(V_s) = \{x^1, ..., x^k\}$ and $\bar{x} = \langle x^1, \ldots, x^k \rangle$, then for each $1 \leqslant i \leqslant k$, the rule

$$aux_{V_s}(\bar{x}) \leftarrow \textstyle\bigwedge_{i=1}^n R_i(\bar{x}_i, \mathbf{t}) \wedge \varphi \wedge x^i \neq null.$$

3. The old tuple collecting rules:
(a) For each $R^j \in \mathcal{SP}(V_s)$, $1 \leq j \leq a$:

$$R^j(\bar{x}_j, \mathbf{bu}) \leftarrow \textstyle\bigwedge_{i=1}^n R_i(\bar{x}_i, \mathbf{t}),\ \varphi,\ aux_{V_s}(\bar{x}), \bigwedge_{v_l \in \mathcal{C}(V_s)} v_l \neq null,\ R^j(\bar{x}_j, \mathbf{u}),\ \bigwedge_{v_l \in \mathcal{S}(V_s) \cap \bar{x}_j} v_l \neq null.$$

(b) For each $R^c \in \mathcal{CP}(V_s)$, $1 \leq c \leq b$:

$$R^c(\bar{x}_c, \mathbf{bu}) \leftarrow \textstyle\bigwedge_{i=1}^n R_i(\bar{x}_i, \mathbf{t}),\ \varphi,\ aux_{V_s}(\bar{x}), \bigwedge_{v_l \in \mathcal{C}(V_s)} v_l \neq null,\ R^c(\bar{x}_c, \mathbf{u}).$$

[13]To be more precise, we should talk about variables in relevant positions or arguments, as we did before, e.g. in Section III, but the description would be less intuitive.

4. For each $R \in \mathcal{R}$, the rule: $R(\bar{x}, \mathbf{t}) \leftarrow R(\bar{x}, \mathbf{u})$.

5. For each $R \in \mathcal{R}$, the rule:
$$R(\bar{x}, \mathbf{s}) \leftarrow R(\bar{x}, \mathbf{t}),\ \mathit{not}\ R(\bar{x}, \mathbf{bu}).$$

Rules in 1. create program facts from the initial instance. Rules in 2. are the most important and express how to impose secrecy by changing attribute values into nulls. Notice that, by definition, $\mathcal{CP}(V_s)$ and $\mathcal{SP}(V_s)$ already already include those changes. The body of the rule becomes true when the database instance does not nullify the view, and the head captures the intended ways of imposing secrecy. Rules in 3. collect the tuples in the database that have already been updated and (virtually) no longer exist in the database. Rules 4. capture the atoms that are part of the database or updated atoms in the process of imposing secrecy. Rules in 5. collect the tuples in the secrecy instance, as those that did not become old.

The same secrecy program can be used with different queries. However, available optimization techniques can be used to specialize the program for a given query (cf. [11], [5] for this kind of optimizations for repair logic programs).

VI. The CQA Connection

Consider a database instance D that fails to satisfy a given set of integrity constraints IC. It still contains useful and some semantically correct information. The area of *consistent query answering* (CQA) [3], [5] has to do with: (a) Characterizing the information in D that is still semantically correct wrt IC, and (b) Characterizing, and computing, in particular, the semantically correct, i.e. consistent, answers to a query $\mathcal{Q}$ from D wrt IC. The first goal is achieved by proposing a *repair semantics*, i.e. a class of alternative instances to D that are consistent wrt IC and minimally depart from D. The consistent information in D is the one that is invariant under all the repairs in the class. This applies in particular to the consistent answers: They should hold in every minimally repaired instance.

There are some connections between CQA and our treatment of privacy preserving query answering. Notice that every view definition of the form (2) can be seen as an integrity constraint expressed in the FO language $L(\Sigma \cup \{V_s\})$:

$$\forall \bar{x}(V_s(\bar{x}) \ \longleftrightarrow\ \exists \bar{y}(R_1(\bar{x}_1) \wedge \cdots \wedge R_n(\bar{x}_n) \wedge \varphi)), \quad (9)$$

with $\bar{y} = (\bigcup \bar{x}_i) \smallsetminus \bar{x}$. From this perspective, the problem of *view maintenance*, i.e. of maintaining the view defined by (9) synchronized with the base relations [17] becomes a problem of *database maintenance*, i.e. maintenance of the consistency of the database wrt (9) seen as an IC. This also works in the other direction since every IC can be associated to a violation view, which has to stay empty for the IC to stay satisfied.

Actually, we want more than maintaining the view defined in (9). We want it to be empty or returning only tuples with null values. In consequence, we have to impose the following ICs on D, which are obtained from the RHS of (9): If $\bar{x}$ is $x^1, \ldots, x^k$, then for $1 \leq i \leq k$,

$$\forall \bar{x}\bar{y}\neg(R_1(\bar{x}_1) \wedge \cdots \wedge R_n(\bar{x}_n) \wedge \varphi \wedge x^i \neq \mathit{null}). \quad (10)$$

That is, from each view definition (9) we obtain k *denial constraints* (DCs), i.e. prohibited conjunctions of (positive) database atoms and built-ins. DCs have been investigated in CQA under several repair semantics [14], [5].

In our case, the secrecy instances correspond to the repairs of D wrt the set DCs in (10). These repairs are defined according to the null-based (and attribute-based [5]) repair semantics of Section III, i.e. $\leq_D$-minimality (cf. Example 10). Through this correspondence we can benefit from concepts and techniques developed for CQA.

Example 20. The secrecy view defined by
$$V_s(x, z) \leftarrow P(x, y), R(y, z), y < 3$$
gives rise to the following denial constraints: $\neg\exists xyz(P(x, y) \wedge R(y, z) \wedge y < 3 \wedge x \neq \mathit{null})$ and $\neg\exists xyz(P(x, y) \wedge R(y, z) \wedge y < 3 \wedge z \neq \mathit{null})$. A instance D has to be minimally repaired in order to satisfy them. ■

VII. Related Work

Other researchers have investigated the problem of data privacy and access control in relational databases. We described in Section I the approach based on authorization views [27], [33]. In [19], the privacy is specified through values in cells within tables that can be accessed by a user. To answer a query $\mathcal{Q}$ without violating privacy, they propose the table and query semantics models, which generate masked versions of the tables by replacing all the cells that are not allowed to be accessed with `NULL`. When the user issues $\mathcal{Q}$, the latter is posed to the masked versions of the tables, and answered as usual. The table semantics is independent of any queries, and views. However, the query semantics takes queries into account. [19] shows the implementation of two models based on query rewriting.

Recent work [30] has presented a labeling approach for masking unauthorized information by using two types of special variables. They propose a secure and sound query evaluation algorithm in the case of cell-level disclosure policies, which determine for each cell whether the cell is allowed to be accessed or not. The algorithm is based on query modification, into one that returns less information than the original one. Those approaches propose query rewiring to enforce fine-grained access control in databases. Their approach is mainly algorithmic.

Data privacy and access control in incomplete propositional databases has been studied in [6], [7], [31]. They take a different approach, *control query evaluation* (CQE), to fine-grained access control. It is policy-driven, and aims to ensure confidentiality on the basis of a logical framework. A security policy specifies the facts that a certain user is not allowed to access. Each query posed to the database by that user is checked, as to whether the answers to it would allow the user to infer any sensitive information. If that is the case, the answer is distorted by either *lying* or *refusal* or *combined lying and refusal*. In [8], they extend

CQE to restricted incomplete FO logic databases via a transformation into a propositional language. This approach seem to be incomparable to ours. They do not use null values, and the issue of maximality of answers that do not compromise privacy is not explicitly addressed.

Our approach is based on producing virtual updates on the database, by forcing the secrecy views to become null. This is clearly reminiscent of the older, but still challenging database problem of updating a database through views [13]. Here we confront new difficulties, namely the occurrence of SQL nulls with a special semantics, and the minimality of null-based changes on the base relations.

In [9] a null-based repair semantics was introduced, but it differs from the one introduced in Section III. The former was proposed for enforcing satisfaction of sets of ICs that include referential ICs, which require the possible insertion of new tuples with nulls. The comparison between instances is based onsets of full tuples and also on the occurrence of nulls in them. Here, we enforce secrecy by changes of attributes values only.

A representation of null values in logic programs with stable model semantics is proposed in [28], whose aim is to capture the intended semantics of null values *à la* Reiter, i.e. as found in his logical reconstruction of relational databases [26]. Two remarks have to be made here. First, Reiter reconstructs "logical" nulls, but not SQL nulls. In our work we use the latter, as done in database practice. Second, we take care of nulls by proposing a new query answering semantics that can be captured in classic logical terms via query rewriting. The rewritten queries are the input to a logic program, which then treats them as ordinary constants (without having to give a logical account of them).

VIII. Conclusions

In this work, we have developed a logical framework and a methodology for answering conjunctive queries that do not reveal secret information as specified by secrecy views. Our work is of a foundational nature, and attempts to provide a theoretical basis, or at least part of that basis, for possible technological developments. Implementation efforts and experiments, beyond the proof-of-concept examples we have run with *DLV*, are left for future work.

We have concentrated on conjunctive secrecy views and conjunctive queries. We have assumed that the databases may contain nulls, and also nulls are used to protect secret information, by virtually updating with nulls some of the attribute values. In each of the resulting alternative virtual instances, the secrecy views either become empty or contain a tuple showing only null values. The queries can be posed against any of these virtual instances or cautiously against all of them, simultaneously. The latter guarantees privacy.

The update semantics enforces (or captures) two natural requirements. That the updates are based on null values, and that the updated instances stay close to the given instance. In this way, the query answers become implicitly maximally informative, while not revealing the original contents of the secrecy views.

The null values are treated as in the SQL standard, which in our case, and for conjunctive query answering, is reconstructed in classical logic. This reconstruction captures well the "semantics" of SQL nulls (which in not clear or complete in the standard), at least for the case of conjunctive query answering, and some extensions thereof. This is the main reason for concentrating on conjunctive queries and views. In this case, queries and views can be syntactically transformed into conjunctive queries and views for which the evaluation or verification can be done by treating nulls as any other constant.

The secret answers are based on a skeptical semantics. In principle, we could consider instead the more relaxed *possible* or *brave* semantics: an answer would be returned if it holds *in some* of the secrecy instances. The *possibly secret answers* would provide more information about the original database than the (certainly) secret answers. However, they are not suitable for our the privacy problem.

Example 21. (example 10 continued) A *possibly secret answer* to the query $\mathcal{Q}_1(x, y) : P(x, y)$ is $\langle 1, 2\rangle$, obtained from D_3. Similarly, $\langle 2, 1\rangle$ is a possibly secret answer to $\mathcal{Q}_2(x, y) : R(x, y)$. From these possibly secret answers, the user can obtain the contents of the secrecy view. ■

We introduced disjunctive logic programs with stable model semantics to specify the secrecy instances. This is a single program that can be used to compute secret answers to any conjunctive query. This provides a general mechanism, but may not be the most efficient way to go for some classes of secrecy views and queries. *Ad hoc* methods could be proposed for them, as has been the case in CQA [4], [5].

Our work leaves several open problems, and they are matter of ongoing and future research. Complexity issues have to be explored. For example, of deciding whether or not a particular instance is a secrecy instance of an original instance. Also, of deciding if a tuple is a secret answer to a query. The connection with CQA, where similar problems have been investigated, looks very promising in this regard.

Another problem is about query rewriting, i.e. about the possibility of rewriting the original query into a new FO query, in such a way that the new query, when answered by the given instance, returns the secret answers. From the connection with CQA we can predict that this approach has limited applicability, but whenever possible, it should be used, for its simplicity and lower complexity.

For future work, it would be interesting to investigate the connections with *view determinacy* [25], that has to do with the possible determination of extensions of query answers by a set of views with a fixed contents. The occurrence of SQL nulls and their semantics introduces a completely new dimension into this problem.

A natural extension of this work would go in the direction of freeing ourselves from the assumptions listed at the end of Section IV. Their relaxation would create a challenging new scenario, and most likely, would require a non-straightforward modification of our approach. One of these possible relaxations consists in the addition of ICs

to the schema. If they are known to the user, and, most importantly, that they are satisfied by the database, then privacy could be compromised. Also the updates leading to the virtual updates should take these ICs into account, to produce consistent secrecy instances.

It would also be interesting to investigate more expressive queries and secrecy views, going beyond the conjunctive case. However, if we allow negation, the challenges become intrinsically more difficult. On one side, in the case of secrecy views, negation becomes a fundamental complication for privacy [27], [33]. On the other, the query rewriting methodology that captures nulls as ordinary constants (cf. Section II-B) that we have used in our work does not include the combination of nulls and negation. The extension of our privacy approach to queries or secrecy views with negation would make it necessary to first attempt an extension of this kind of query rewriting. However, this requires to agree on a sensible semantics for SQL nulls in the context of such more expressive queries, something that is definitely worth investigating.

Acknowledgements: This research started when Leo Bertossi was spending his sabbatical at the TU Vienna. Support from Georg Gottlob, Thomas Eiter and a Pauli Fellowship of the "Wolfgang Pauli Institute, Vienna" is highly appreciated. We are indebted to Thomas Eiter and Loreto Bravo for technical conversations at an early stage of this research, and to Sina Ariyan for some computational experiments. Research funded by NSERC Discovery and NSERC/IBM CRDPJ/371084-2008.

References

[1] Abiteboul, S., Hull, R. and Vianu, V. *Foundations of Databases*, Addison-Wesley, 1995.

[2] Barcelo, P. Applications of Annotated Predicate Calculus and Logic Programs to Querying Inconsistent Databases. MSc Thesis PUC, 2002. http://people.scs.carleton.ca/~ bertossi/papers/tesisk.pdf

[3] Bertossi, L. Consistent Query Answering in Databases. *ACM Sigmod Record*, June 2006, 35(2):68-76.

[4] Bertossi, L. From Database Repair Programs to Consistent Query Answering in Classical Logic (extended abstract). In Proc. The Alberto Mendelzon International Workshop on Foundations of Data Management (AMW'09), CEUR-WS, Vol-450, 15 pp.

[5] Bertossi, L. *Database Repairing and Consistent Query Answering*, Morgan & Claypool, Synthesis Lectures on Data Management, 2011.

[6] Biskup, J. and Weibert, T. Confidentiality Policies for Controlled Query Evaluation. In *Data and Applications Security*, Springer LNCS 4602, 2007, pp. 1-13.

[7] Biskup,J. and Weibert. Keeping Secrets in Incomplete Datbabases. *International Journal of Information Sercurity*, 2008, 7(3):199-217.

[8] Biskup, J., Tadros, C. and Wiese, L. Towards Controlled Query Evaluation for Incomplete First-Order Databases. In Proc. FoIKS'10, Springer LNCS 5956, 2010, pp. 230-247.

[9] Bravo, L. and Bertossi, L. Semantically Correct Query Answers in the Presence of Null Values. Proc. EDBT WS on Inconsistency and Incompleteness in Databases (IIDB'06), J. Chomicki and J. Wijsen (eds.), Springer LNCS 4254, 2006, pp. 336-357

[10] Bravo, L. Handling Inconsistency in Databases and Data Integration Systems. PhD. Thesis, Carleton University, Department of Computer Science, 2007. http://people.scs.carleton.ca/~ bertossi/papers/Thesis36.pdf

[11] Caniupan, M. and Bertossi, L. The Consistency Extractor System: Answer Set Programs for Consistent Query Answering in Databases. *Data & Knowledge Engineering*, 2010, 69(6):545-572.

[12] Codd, E.F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 1979, 4(4):397-434.

[13] Cosmadakis, S. and Papadimitriou, Ch. Updates of Relational Views. *Journal of the ACM*, 1984, 31(4):742-760.

[14] Chomicki, J. and Marcinkowski, J. Minimal-Change Integrity Maintenance Using Tuple Deletions. *Information and Computation*, 2005, 197(1-2):90-121.

[15] Gelfond, M. and Lifschitz, V. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 1991, 9:365-385.

[16] Gelfond, M. and Leone, N. Logic Programming and Knowledge Representation: The A-Prolog Perspective. *Artificial Intelligence*, 2002, 138(1-2):3-38.

[17] Gupta, A. and Singh Mumick, I. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 1995, 18(2):3-18.

[18] Imielinski, T. and Lipski, W. Jr. Incomplete Information in Relational Databases. *Journal of the ACM*, 1984, 31(4):761-791.

[19] LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y. and DeWitt, D. Limiting Disclosure in Hippocratic Databases. In *Proc. International Conference on Very large Data Bases* (VLDB'04), 2004, pp. 108-119.

[20] Lechtenbörger, J. and Vossen, G. On the Computation of Relational View Complements. *Proc. ACM Symposium on Principles of Database Systems* (PODS'02), 2002, pp. 142-149.

[21] Lechtenbörger, J. The Impact of the Constant Complement Approach towards View Updating. *Proc. ACM Symposium on Principles of Database Systems* (PODS'03), 2003, pp. 49-55.

[22] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 2006, 7(3):499-562.

[23] Levene, M. and Loizou, G. *A Guided Tour of Relational Databases and Beyond*. Springer, 1999.

[24] Li, L. Achieving Data Privacy Through Virtual Updates. MSc. Thesis, Carleton University, Department of Computer Science, 2011. http://people.scs.carleton.ca/~ bertossi/papers/thesisLechen.pdf

[25] Nash, A., Segoufin, L. and Vianu, V. Views and Queries: Determinacy and Rewriting. *ACM Transactions on Database Systems*, 2010, 35(3).

[26] Reiter, R. Towards a Logical Reconstruction of Relational Database Theory. In *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos and J.W. Schmidt (eds.), Springer, 1984, pp. 191–233.

[27] Rizvi, S., Mendelzon, A., Sudarshan, S. and Roy, P. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proc. Proc. ACM International Conference on Management of Data* (SIGMOD'04), 2004, pp. 551-562.

[28] Traylor, B. and Gelfond, M. Representing Null Values in Logic Programming. In *Logical Foundations of Computer Science, Proc. LFCS'94*. Springer LNCS 813, 1994, pp. 341-352.

[29] Vassiliou, Y. Null Values in Data Base Management: A Denotational Semantics Approach. In *Proc. ACM International Conference on Management of Data* (SIGMOD'79), 1979, pp. 162-169.

[30] Wang, Q., Yu, T., Li, N., Lobo, J., Bertino, E., Irwin, K. and Byun, J.-W.. On the Correctness Criteria of Fine-Grained Access Control in Relational Databases. In *Proc. International Conference on Very large Data Bases* (VLDB'07), 2007, pp. 555-566.

[31] Weibert, T. A Framework for Inference Control in Incomplete Logic Databases. PhD thesis, Technische Universität Dortmund, 2008.

[32] Zaniolo, C. Database Relations with Null Values. In *Proc. ACM Symposium on Principles of Database Systems* (PODS'82), 1982, pp. 27-33. ACM.

[33] Zhang, Z. and Mendelzon, A. Authorization Views and Conditional Query Containment. In *Proc. International Conference on Database Theory* (ICDT'05), Springer LNCS 3363, 2005, pp. 259-273.

Leopoldo Bertossi has been Full Professor at the School of Computer Science, Carleton University (Ottawa, Canada) since 2001. He is Faculty Fellow of the IBM Center for Advanced Studies. He obtained a PhD in Mathematics from the Pontifical Catholic University of Chile (PUC) in 1988. Until 2001 he was professor at the Department of Computer Science, PUC; and also the President of the Chilean Computer Science Society (SCCC) in 1996 and 1999-2000. His research interests include database theory, data integration, peer data management, intelligent information systems, data quality, knowledge representation, and

answer set programming.

Lechen Li was born in Sichuan, China in 1985. She received a Bachelor in Computer Engineering from the Sichuan Normal University, Chengdu, China, and a MSc degree in computer science in 2011 from Carleton University, Ottawa, Canada, under the supervision of Prof. L. Bertossi. Her master's research was in the area of data privacy.