ACHIEVING DATA PRIVACY THROUGH VIRTUAL UPDATES

by

Lechen Li

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario

January, 2011

Contents

List of Tables

List of Figures

Abstract

There may be sensitive information in a relational database that we want to keep hidden from a user of group thereof. The sensitive data is characterized as the contents of a view. Whenever the user poses queries about these *secrecy views*, the set of answers either becomes empty or contains tuples with only null values. This is achieved through updates of the given instance. Since the database is not expected to be physically changed to produce this result, it is only virtually and minimally updated, and those changes are reflected in the view contents. Virtual updates are based on the use of null values as used in the SQL standard. The different ways to update the underlying database are specified as the models of logic programs with stable model semantics. The programs become the basis for the computation of the *secret answers* to queries, i.e. those that do not reveal the sensitive information. More precisely, secret answers can be computed by evaluating queries against the programs, e.g. using the disjunctive datalog system (DLV). An optimization approach is developed for evaluating queries, in such way only relevant data facts and program rules are involved in query evaluation.

Acknowledgements

It is a pleasure to thank the many people who made this thesis possible.

First of all, I would like to thank Professor Leopoldo Bertossi, my supervisor, for his constant encouragement, advice, support, lots of good ideas throughout my thesis. He has helped me a lot through all the stages of the writing of my thesis. Without his instructive advice and suggestion, the completion of this thesis would not have been possible.

I am also grateful to the School of Computer Science at Carleton University for providing us with a good environment and facilities to complete this research. I'd like to thank all my professors who have helped me to broaden my view and knowledge.

Last but not least, I would like to give special thanks to all my family for supporting me and being there whenever help was needed.

Chapter 1

Introduction

Database management systems allow for massive storage of data, which can be easily and efficiently accessed and manipulated. However, at the same time, the problems of data privacy are increasingly important. 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, authorization is required. More precisely, certain users can have access to only certain portions of a database. Hopefully, what a particular user (or class of them) is allowed or not allowed to access should be specified in a declarative manner. Currently, authorization mechanisms in commercial database management systems are at the level of columns, the whole relations, or on views. But there are many situations that demand a very fine-grained level, such as at the level of individual tuples or cells. For example, imagine a company's database that included information about an employee's health. If that employee has a disease, this must be kept as a secret. Otherwise, this health information could be publicly released.

A possible way to solve tuple-based access control is to create views for each user, or class of them. Those views contain only selected tuples of a database, those it allows a user (or class of them) to access. However, this approach is not practical since there may be a large number of users. Most of information systems in practice ignore tuple-based access control at the database level, and delegate this problem to the application level. In other words, application programs are used to ensure appropriate access rather than databases. Although widely used, this approach has serval disadvantages. For example, lots of source code has to be changed if any policy is changed. In addition, by putting access control into the application, you need to duplicate the access logics in each application that accesses the database, which will increase the cost and complexity of development. Therefore, it is better to address access control at the database level.

Some recent papers [37, 44] approach fine-grained access control in the database in terms of authorization views. 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 that 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 [37]. If no *complete rewriting* is possible, the query is rejected. In [44] 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 is sensitive rather than about the data that can be publicly released. Dealing with this problem as "the complement" of the problem formulated in terms of authorization views is not natural, and not necessarily easy, especially considering that complements of database views would be involved [27, 28].

According to our approach, the information to be protected is declared as a view, a secrecy view, or a collection of them. Each user or class of them may have associated a set of secrecy views. When a user poses a query on the database, the system will virtually update some of the attributes values in terms of a fixed set of secrecy views. In each of the resulting updated instances, the extension of secrecy views either becomes empty or contains tuples showing only null values. Then, the original query is posed to the updated instances, and is answered as usual. In this way, the system will return answers to the query that do not reveal the secret data. In this work, we consider updates that modify attribute values assigning null values.

Example 1.1. Consider the following relational database D:

Marks	*StudentID*	*CourseID*	*Mark*
	001	01	56
	001	02	90
	002	02	70

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

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

Now, a user of the database wants to see students' marks, using the following query:

$$Q(sid, cid, mark) \leftarrow Mark(sid, cid, mark). \tag{1.1}$$

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 in terms of view definitions, in such a way that the secret information, i.e. the extension of the secrecy view, can not be revealed to the user. Here, in order to protect the tuple $Mark(001, 01, 56)$, the new instance D' is obtained by virtually updating the original instance, changing the attribute value 56 to *null*. The virtually updated instance is D':

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

Marks	*StudentID*	*CourseID*	*Mark*
	001	01	*null*
	001	02	90
	002	02	70

Now, by posing the query

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

which corresponds to the secrecy view definition, to the "updated" database D', the user gets an empty answer, because the comparison of *null* with any value will be evaluated as *unknown*. Similarly, query (1.1) will get the first tuple with *null* instead of 56. The user cannot obtain the mark 56 by combining any answers obtained through other queries on D'. □

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 update has to be minimal in some sense, while still doing its 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. As expected, null values have received attention from the database community [43, 41, 24]. Null values may have several 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 thesis, by *null*. Furthermore, we will treat *null* as the `NULL` in SQL relational databases. Since the SQL standard does not provide a precise, formal semantics for `NULL`, we will adopt here the formal, logical reconstruction of SQL nulls proposed in [12]. It captures the semantics of the SQL `NULL` that are relevant for our work on privacy, namely integrity constraint satisfaction and query answering. This makes our approach to secrecy compatible with and implementable on top of commercial DBMSs.

1.1 Problem Statement and Contributions

The goal of this research is to develop methods to retrieve answers for first-order conjunctive queries that do not reveal any secret data from relational databases. In most practical cases, databases always contain null values. So, we will put special interest on databases with null values. The key features of our research are the following:

1. The notion of *secrecy views* is introduced. Basically, the sensitive information is protected via a fixed set of secrecy views. It is associated to a particular user or class of them. We have restricted ourselves mainly to the case of conjunctive secrecy views. The following is an example of a conjunctive secrecy view:

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

   Here, the secrecy view is specified using a Datalog rule. In general, we will use Datalog notation for views and queries.

2. In this thesis, we introduce *null*-based virtual updates on an instance, to keep the secrecy view extensions secret. The semantics of secrecy in the presence of null values that we provide is model-theoretic, in the sense that the possible admissible instances after the update, the so-called *secrecy instances*, are characterized and specified. On a secrecy instance, the extensions of the secrecy

views contain only tuples with *null* or become empty. Furthermore, the secrecy instances do not depart from the original instance by more than what is needed to protect the secret data. So, they minimally differ in a precise sense from the original instance.

3. The semantics of *secret answers* to a query is presented. 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.

4. In this thesis, we also show that our null-based semantics can be captured in terms of disjunctive logic programs with extra annotation constants, by establishing a correspondence between the *secrecy instances* of a relational database and the stable models of the program. The logic program can be used to specify the *secrecy instances* of the original database and to obtain *secret answers*. The basic idea behind the logic programming-based approach is that, since we need to reason simultaneously with all the secrecy instances of a database, we have to succinctly specify the class of secrecy instances. Then, different reasoning tasks could be performed, in particular, computation of secret answers to queries. We also show how to obtain a reduced secrecy view and a subset of sources for a query.

5. We also develop optimization techniques to reduce the amount of data involved in the computation of secret answers to queries, in such a way that only a subset of database facts and program rules are involved in query evaluation.

6. We discuss several additional issues in relation to our approach to privacy. There are some interesting connections between this work and *consistent query answering* [5, 8]. The secrecy instances can be seen as *repairs* of the original

instance that enforce semantic constraints on a given instance. These repairs minimally differ from the given instance under a repair semantics that is based on attribute updates with *null*. In this sense, secret query answering becomes a form of consistent query answering (CQA).

7. In addition, IBM InfoSphere Master Data Management Server (MDM) provides a mechanism, named Suspect Duplicate Processing, to keep a single view of the customer information. Our privacy approach could make use of this mechanism to identify or detect duplicates for improving data quality.

This thesis is structured as follows. Chapter 2 introduces basic definitions and concepts needed in the later chapters. Chapter 3 discusses how null values are treated in the different DBMSs. Chapter 4 presents a precise semantics for secrecy instances and secret answers. Chapter 5 describes the specification of secrecy instances via logic programs and how to use them to compute secret answers to first-order conjunctive queries. Chapter 6 provides experimental results on various scenarios of user queries. Chapter 7 discuses several additional issues with respect to the previously introduced semantics, including connection with CQA and duplicate detection. Chapter 8 presents some conclusions and future work.

Chapter 2

Preliminaries

In this chapter we recall basic concepts of relational databases. We also introduce the database repairs and null-value semantics presented in [11], which is the underlying semantics that will be used in the rest of this work.

2.1 Databases and Integrity Constraints

In the context of relational databases, we assume that we have a fixed relational schema $\Sigma = (\mathcal{U}, \mathcal{R}, \mathcal{B})$, where $\mathcal{U}$ is the possibly infinite database domain, such that $null \in \mathcal{U}$, $\mathcal{R}$ is a fixed set of database predicates say, $\mathcal{R} = \{R_1, R_2, R_3, \ldots\}$, where each relation R has an a finite, ordered set of attributes $\mathcal{A}_R$; and $\mathcal{B}$ is a fixed set of built-in predicates, like comparison predicates, e.g.: $>, <, =$. $R[i]$ denotes the attribute in position i of predicate $R \in \mathcal{R}$. For $A \subseteq \mathcal{A}_R$, $R[A]$ denotes predicate R projected on the attributes in A.

The schema determines a language $\mathcal{L}(\Sigma)$ of first-order predicate logic. A database instance D compatible with Σ can be seen as a finite collection of ground atoms (or tuples) of the form $R(c_1, ..., c_n)$, where $R \in \mathcal{R}$ and the $c_1, ..., c_n$ are constants in $\mathcal{U}$. Built-in predicates have a fixed extension in every database instance.

A *query* is a first-order formula over language $\mathcal{L}(\Sigma)$. In this research we restrict ourselves to the class of conjunctive queries.

Definition 1. A conjunctive query $\mathcal{Q}(\bar{x})$ (CQ) is of the form:

$$\exists \bar{y} (\bigwedge_{i=1}^{n} R_i(\bar{x}_i, \bar{y}_i) \wedge \varphi), \tag{2.1}$$

where $R_i \in \mathcal{R}$, φ is a conjunction of built-ins whose variables appear in the R_is, and $\bar{x} = \bigcup_i \bar{x}_i$ which are the *free* variables of the query. If the query does not have free variables, i.e. $\bar{x}$ is the empty set, it is called a *boolean* query. □

Conjunctive queries can also be written as Datalog rules. The following is Datalog notation for the query (2.1):

$$Ans(\bar{x}) \leftarrow R_1(\bar{x}_1, \bar{y}_1), \ldots, R_n(\bar{x}_n, \bar{y}_n), \varphi, \tag{2.2}$$

where Ans is an intensional relation symbol, and the head $Ans(\bar{x})$ represents the results of the query $\mathcal{Q}(\bar{x})$.

Definition 2. Given a database instance D without *null*, a tuple of constants $\bar{t}$ is an answer to a query $\mathcal{Q}(\bar{x})$ of the form (2.1) in D iff $D \vDash \mathcal{Q}(\bar{t})$, i.e. $\mathcal{Q}(\bar{x})$ becomes true in D when the variables are replaced by the corresponding constants in $\bar{t}$. If $\mathcal{Q}$ is a sentence (boolean query), then *yes* is an answer iff $D \vDash \mathcal{Q}$. Otherwise, the answer is *no*. □

Example 2.1. Consider the database $D = \{P(1,1), Q(1)\}$. The query $\mathcal{Q}_1 : P(1,1)$ has *yes* as an answer. Query $\mathcal{Q}_2(x) : \exists y(P(x,y) \wedge Q(y))$ has 1 has an answer. □

With the purpose of answering first-order queries in a database with *null*, the notion of query answering will be modified (cf. Section 2.2).

Integrity Constraints (ICs) have been considered in the relational databases for adding semantics, and ensuring accuracy and consistency of data. The most common type of constraints are functional dependencies and inclusion dependencies.

A *functional dependency* [4] (FD) over a database schema Σ is an expression of the form $R : X \rightarrow Y$, where $R \in \mathcal{R}$ and $X,\ Y$ are sets of attributes associated to R.

A relation D over Σ satisfies $X \rightarrow Y$, denoted by $D \vDash X \rightarrow Y$, if for any two tuples that have the same values in the attributes in X, also have the same values in the attributes in Y. A set of attributes X is a *candidate key* of a relation R if for every attribute Y in R, it holds that $R : X \rightarrow Y$, and no subset of X has this property. One of the candidate keys can be chosen as the *primary key* (PK).

An *inclusion dependency* [4] (IND) over a database schema Σ is an expression of the form $R[X] \subseteq S[Y]$, where R, S are (possibly identical) relation names in Σ, and X, Y are sets of attributes from R and S, respectively. An instance D of Σ satisfies $R[X] \subseteq S[Y]$, denoted by $D \vDash R[X] \subseteq S[Y]$, if for each tuple $\bar{r}$ in $D(R)$[1], there exists a tuple $\bar{s}$ in $D(S)$, such that $\bar{s}[Y] = \bar{r}[X]$. An inclusion dependency is said to be *full* if $X = \mathcal{A}_R$, and *partial* if $X \subsetneqq \mathcal{A}_R$. A *foreign key constraint* is an inclusion dependency $S[Y] \subseteq R[X]$, where X is the primary key of R. More generally,

Definition 3. [12] An integrity constraints is a sentence of the form:

$$\forall \bar{x}(\bigwedge_{i=1}^{n} P_i(\bar{x}_i) \rightarrow \exists \bar{z}(\bigvee_{j=1}^{m} Q_j(\bar{y}_j, \bar{z}_j) \vee \varphi)), \tag{2.3}$$

where $P_i, Q_j \in R$, $\bar{x} = \bigcup_{i=1}^{m} \bar{x}_i$, $\bar{z} = \bigcup_{j=1}^{n} \bar{z}_i$, $\bar{y}_i \subseteq \bar{x}$, $\bar{x} \bigcap \bar{z} = \varnothing$, $\bar{z}_i \bigcap \bar{z}_j = \varnothing$ for $i \neq j$. Here, φ is a disjunction of built-in predicates from $\mathcal{B}$, whose variables appear in the antecedent of the implication. We will assume that there exists a propositional atom **false** $\in \mathcal{B}$ that is always false in a database. $\Box$

Definition 4. [12] A *universal integrity constraint* (UIC) is an sentence in $\mathcal{L}(\Sigma)$ of the form:

$$\forall \bar{x}(\bigwedge_{i=1}^{n} R_i(\bar{x}_i) \longrightarrow (\bigvee_{j=1}^{m} Q_j(\bar{y}_j) \vee \varphi)). \tag{2.4}$$

That is a formula of the form (2.3), without existential quantifiers. $\Box$

[1] $D(R)$ denotes the extension of predicate $R \in \mathcal{R}$ in an instance D.

Definition 5. [12] A *referential integrity constraint* (RIC) is a sentence in $\mathcal{L}(\Sigma)$ of the form:

$$\forall \bar{x}(P(\bar{x}) \rightarrow \exists \bar{z}\ Q(\bar{y}, \bar{z})). \tag{2.5}$$

That is a formula of the form (2.3), with $m = n = 1$, without a φ, $\bar{y} \subseteq \bar{x}$, and $P, Q \in \mathcal{R}$. $\square$

The class of ICs of the form (2.3) contains most of the ICs commonly found in database practice. Functional dependencies can be expressed by several implications of the form (2.4), each of them with a single equality in the consequent. Partial inclusion dependencies are RICs, and full inclusion dependencies are UICs. A *denial constraint* can be expressed as $\forall \bar{x}(\bigwedge_{i=1}^{n} R_i(\bar{x}_i) \rightarrow \textbf{false})$.

Example 2.2. Consider a database schema $\Sigma = \{Employee(EId, Name, Dept), Dept(DId, Name)\}$. The following are ICs:

(a) The functional dependency (FD) $Employee : EId \rightarrow Name$, expressed in $L(\Sigma)$ by $\forall eid \forall n1 \forall n2 (Employee(eid, n1) \land Employee(eid, n2) \rightarrow n1 = n2)$.

(b) The inclusion dependency (IND) $Employee[Dept] \subseteq Dept[DeptID]$, expressed by $\forall eid \forall n \forall dept (Employee(eid, n, dept) \rightarrow \exists dn Dept(dept, dn))$. $\square$

We consider a fixed finite set IC of ICs of the form (2.3). A database D, possibly with null values, is *consistent* with respect to IC if it satisfies the given set IC; otherwise, we say D is *inconsistent.* The semantics of constraint satisfaction in presence of null values we consider is the one defined in [12, 11]. In order to present it, we need to introduce some concepts.

Definition 6. [12] For an IC $\psi \in \mathcal{L}(\Sigma)$ and a term t (i.e. a variable or a domain constant), $pos^R(\psi, t)$ is the set of positions in predicate $R \in \mathcal{R}$ where t appears in ψ.

The set of *relevant attributes* for an IC ψ of the form (2.3) is:

$\mathcal{A}(\psi) = \{R[i] \mid x$ is a variable present at least twice in $\mathcal{V}(\psi)$, and $i \in pos^R(\psi, x)\} \cup$

$\{R[i] \mid c$ is a constant in ψ and $i \in pos^R(\psi, c)\}$,

where $R[i]$ denotes a position in relation R. □

The relevant attributes include the attributes needed to check the satisfaction of the constraints, e.g. the attributes in joins, in built-ins, etc.

Definition 7. [12] Given a set of attributes $\mathcal{A}$, and a predicate $R \in \mathcal{R}$, we denote by $R^{\mathcal{A}}$ the predicate R restricted to (or projected onto) the attributes in $\mathcal{A}$. $D^{\mathcal{A}}$ denotes the database D with all its database atoms projected onto the attributes in $\mathcal{A}$, i.e. $D^{\mathcal{A}} = \{R^{\mathcal{A}}(\Pi_{\mathcal{A}}(\bar{t})) \mid R(\bar{t}) \in D\}$, where $\Pi_{\mathcal{A}}(\bar{t})$ is the projection on $\mathcal{A}$ of tuple $\bar{t}$. $D^{\mathcal{A}}$ has the same underlying domain $\mathcal{U}$ as D. □

Example 2.3. Consider a IC ψ of the form (2.3): $\forall x(P(x) \rightarrow \exists z Q(x, z))$, and the following database instance:

P	A
	a
	$null$

Q	A	B
	a	1
	b	2

Here x appears twice in ψ, therefore $\mathcal{A}(\psi) = \{P[1], Q[1]\}$. The value in the attribute B is not relevant to check satisfaction of the constraint ψ. This makes sense, since we just want values in the attribute A of P also to appear in the attribute A of Q, which is equivalent to checking if $\forall x(P^{\mathcal{A}(\psi)}(x) \rightarrow Q^{\mathcal{A}(\psi)}(x))$ is satisfied by $D^{\mathcal{A}(\psi)}$, where $D^{\mathcal{A}(\psi)}$ is:

$P^{\mathcal{A}(\psi)}$	A
	a
	$null$

$Q^{\mathcal{A}(\psi)}$	A
	a
	b

□

In a commercial SQL DBMS, a constraint is satisfied if any of the relevant attributes has a *null* value or the constraint is satisfied in the traditional way. More formally,

Definition 8. [12] A constraint ψ of the form (2.3) is satisfied in the database instance D, denoted $D \vDash_N \psi$, iff $D^{\mathcal{A}(\psi)} \vDash \psi^N$, where ψ^N is

$$\forall \bar{x}(\bigwedge_{i=1}^{m} R_i^{\mathcal{A}(\psi)}(\bar{x}_i) \rightarrow (\bigvee_{v_j \in (\mathcal{A}(\psi) \cap \bar{x})} v_j = null \ \vee \ \exists \bar{z}(\bigvee_{j=1}^{n} Q_j^{\mathcal{A}(\psi)}(\bar{y}_j, \bar{z}_j) \ \vee \ \varphi))), \tag{2.6}$$

where $\bar{x} = \cup_{i=1}^{m} \bar{x}_i$, and $\bar{z} = \cup_{j=1}^{n} \bar{z}_j$. Here, $D^{\mathcal{A}(\psi)} \vDash \psi^N$ refers to the classical first-order satisfaction, where *null* is treated as any other constant in the domain. □

We can see from Definition 8 that there are basically three cases for an integrity constraint of the form (2.3) satisfaction: (a) If *null* is in any of the relevant attributes in the antecedent, then the constraint is satisfied; (b) At least one of the conjunctive clauses has to be true. Theses clauses can be checked by the second disjunction in the consequent of formula (2.6), treating *null* as any other constant; (c) If the built-in formula φ is satisfied, then the constraint is satisfied.

Example 2.4. (example 2.3 continued) In order to check if $D \vDash_N \psi$, we need to check $D^{\mathcal{A}(\psi)} \vDash \psi^N$, with ψ^N: $\forall x(P^{\mathcal{A}(\psi)}(x) \rightarrow x = null \vee Q^{\mathcal{A}(\psi)}(x))$. For $x = a$, it holds that $D^{\mathcal{A}(\psi)} \vDash P^{\mathcal{A}(\psi)}(a)$. Since a is not a null value, we need to check if $D \vDash Q^{\mathcal{A}(\psi)}(a)$, which is true in this case. Now, for $x = null$, $D^{\mathcal{A}(\psi)} \vDash P^{\mathcal{A}(\psi)}(null)$, and $null = null$, so the constraint is satisfied. □

2.1.1 The *IsNull* Predicate

There is a very important constraint widely used in DBMSs: the NOT NULL constraint (NNC), which prevents certain attributes from taking the value *null*.

Definition 9. [12] A *NOT NULL*-constraint (NNC) is a denial constraint of the form:

$$\forall \bar{x}(P(\bar{x}) \wedge \mathit{IsNull}(x_i) \rightarrow \mathbf{false}), \tag{2.7}$$

where $x_i \in \bar{x}$ is in the position of the attribute that can not take null values. Here, we introduce a special predicate $\mathit{IsNull}(\cdot)$, with $\mathit{IsNull}(c)$ true iff c is *null*, instead of using the built-in comparison atom $c = \mathit{null}$, because in traditional DBMSs this equality would be always evaluated as *unknown*. $\square$

Notice that a NNC is not of the form (2.3), because it contains the special predicate *IsNull*.

Definition 10. [11] A NNC ψ of the form (2.7), is satisfied by a database D with *null*, denoted $D \models_N \psi$, iff

$$D \models \forall x((P(\bar{x}) \wedge x_i = \mathit{null}) \rightarrow \mathbf{false}), \tag{2.8}$$

with *null* treated as any other constant. $\square$

Example 2.5. Consider the NNC $\psi : \forall xy(P(x,y) \wedge \mathit{IsNull}(x) \rightarrow \mathbf{false})$. This constraint is satisfied if $D \models \forall xy(P(x,y) \wedge x = \mathit{null} \rightarrow \mathbf{false})$. $\square$

2.1.2 Primary Keys

In [11], they also propose a precise formalization in first-order logic of the notions of primary key and unique key satisfaction in databases that conform to the SQL standard SQL-2003.

Definition 11. [11] Given a predicate $R(x_1, \ldots, x_n)$ and its *primary key* $\{R[1], \ldots, R[m]\}$[2], the primary key can be logically expressed as the following set of formulas:

$\forall \bar{x}\bar{y}(R(x_1, \ldots, x_m, x_{m+1}, \ldots, x_n) \wedge R(x_1, \ldots, x_m, y_{m+1}, \ldots, y_n) \rightarrow x_j = y_j),$

for $j = m+1, \ldots, n$.

$\forall \bar{x}\bar{y}(R(x_1, \ldots, x_m, x_{m+1}, \ldots, x_n) \wedge R(x_1, \ldots, x_m, y_{m+1}, \ldots, y_n) \wedge \mathit{IsNull}(x_j)$

$\rightarrow \mathit{IsNull}(y_j)),$ for $j = m+1, \ldots, n$.

$\forall \bar{x}\bar{y}(R(x_1, \ldots, x_m, x_{m+1}, \ldots, x_n) \wedge \mathit{IsNull}(x_j) \rightarrow \textbf{false}),$ for $j = 1, \ldots, m$.

The third set of rules, are NNCs for all the attributes in the key. A *unique key* can be logically expressed by using only the first two set of rules. □

Example 2.6. Consider a schema with relations $R(X, Y)$, with *primary key* (PK) $R[1]$. The following database instance D violates the PK:

R	X	Y
	a	b
	a	$null$

By Definition 11, the PK can be written as $\forall xyz(R(x,y) \wedge R(x,z) \rightarrow y = z)$, $\forall xyz(R(x,y) \wedge R(x,z) \wedge \mathit{IsNull}(y) \rightarrow \mathit{IsNull}(z))$, and $\forall xy(R(x,y) \wedge \mathit{IsNull}(x) \rightarrow \textbf{false})$.

In order to check if D satisfies the PK, we need to check the following three formulas:

$$D \models \forall xyz(R(x,y) \wedge R(x,z) \rightarrow x = null \vee y = null \vee z = null \vee y = z). \tag{2.9}$$

$$D \models \forall xyz(R(x,y) \wedge R(x,z) \wedge y = null \rightarrow x = null \vee z = null). \tag{2.10}$$

$$D \models \forall xyz(R(x,y) \wedge x = null \rightarrow \textbf{false}). \tag{2.11}$$

For $x = a, y = null$, and $z = b$, the antecedent of the rule is satisfied since $R(a,b) \in D$ and $R(a, null) \in D$. For these values the consequent is not satisfied, because $z = null$ is false. Therefore, Formula (2.10) does not hold. Thus, the database instance D is

[2]Without loss of generality we will assume the primary key to be the first m attributes of R.

inconsistent. □

2.2 Query Answering in Databases with Null Values

In order to answer first-order queries in databases with null values, we will use the *null* query answering semantics introduced in [11]. It extends the semantics of IC satisfaction we just presented.

We assume that all the qualifiers in a first-order query are over different variables. For example, $\forall x P(x,y) \wedge \forall x Q(x)$ can be rewritten as the equivalent query $\forall x P(x,y) \wedge \forall z Q(z)$. The queries may contain the special predicate *IsNull*, which captures the SQL query with `IS NULL` and `IS NOT NULL` expressions in first-order logic.

Example 2.7. Consider a table $P(X,Y)$, and the SQL query $\mathcal{Q}$:

```
SELECT P.X
FROM   P
WHERE  Y IS NULL
```

This query $\mathcal{Q}$ can be written in first-order logic as $\exists y(P(x,y) \wedge \mathit{IsNull}(y))$. □

Definition 12. [11] The set of *restricted relevant variables* of a first-order query φ are: $\mathcal{V}^R(\varphi) = \{x \mid x$ is present at least twice in φ, except for the variables in the *IsNull* predicate $\}$ □

Example 2.8. For query $\mathcal{Q}$: $\exists y(P(x,y,z) \wedge Q(y) \wedge \mathit{IsNull}(y))$, $\mathcal{V}^R(Q) = \{y\}$, since y is used twice in $\mathcal{Q}$. □

Definition 13. [11] A *variable assignment function* s, denoted by $s[x|a]$, is a function from the set of variables to the underlying database domain $\mathcal{U}$, by setting $s(x)$ to take the value a. A *term assignment function* $\bar{s}$ is defined as follows: (a) If term t is a variable x, then $\bar{s}(t) = s(x)$. (b) If term t is a constant c or *null*, then $\bar{s}(t) = c$. Given a formula ϕ, $\phi[s]$ denotes the formula obtained from ϕ by replacing its free variables by its value according to s. □

Given a variable assignment function $\mathbf{s}$, we check if D satisfies $\phi[\mathbf{s}]$ by assuming that the quantifiers of *restricted relevant* variables over $(\mathcal{U} \setminus \{null\})$ and of *non-relevant* variables over $\mathcal{U}$. Formally:

Definition 14. [11] Let ϕ be a first-order formula, and $\mathbf{s}$ be a variable assignment function and $\mathcal{B} = \{<, >, =, \mathbf{false}\}$. We define, by induction on ϕ, when D satisfies ϕ with assignment $\mathbf{s}$ with respect to the null-value semantics, denoted $D \vDash_N^q \phi[\mathbf{s}]$. Then, $D \vDash_N^q \phi[\mathbf{s}]$ when ϕ is of one of the following forms:

1. $t_1 \diamond t_n$ for $\diamond \in \{<, >, =\}$, $\bar{\mathbf{s}}(t_1) \neq null$, $\bar{\mathbf{s}}(t_2) \neq null$, and $D \vDash t_1 \diamond t_n$.

2. $R(t_1, \ldots, t_n)$, with $R \in \mathcal{R}$ and $R(\bar{\mathbf{s}}(t_1), \ldots, \bar{\mathbf{s}}(t_n)) \in D$.

3. $\neg\alpha$, and $D \nvDash_N^q \alpha[\bar{\mathbf{s}}]$.

4. $(\alpha \vee \beta)$, and $D \vDash_N^q \alpha[\bar{\mathbf{s}}]$ or $D \vDash_N^q \beta[\bar{\mathbf{s}}]$.

5. $(\alpha \wedge \beta)$, and $D \vDash_N^q \alpha[\bar{\mathbf{s}}]$ and $D \vDash_N^q \beta[\bar{\mathbf{s}}]$.

6. $(\forall y)(\alpha)$, and one of the following holds:

   (a) $y \in \mathcal{V}^R(\alpha)$, and for all a in $(\mathcal{U} \setminus \{null\})$, $D \vDash_N^q \alpha[\bar{\mathbf{s}}[y|a]]$.

   (b) $y \notin \mathcal{V}^R(\alpha)$, and for all a in $\mathcal{U}$, $D \vDash_N^q \alpha[\bar{\mathbf{s}}[y|a]]$.

7. $(\exists y)(\alpha)$, and one of the following holds:

(a) $y \in \mathcal{V}^R(\alpha)$, and there exists an a in $(\mathcal{U} \smallsetminus \{null\})$ with $D \models_N^q \alpha[\bar{\mathbf{s}}[y|a]]$.

(b) $y \notin \mathcal{V}^R(\alpha)$, and there exists an a in $\mathcal{U}$ with $D \models_N^q \alpha[\bar{\mathbf{s}}[y|a]]$.

For all database instance D, $D \not\models_N^q$ **false**. □

Definition 15. [11] A variable assignment $\mathbf{s}$ is *null-valid* with respect to ϕ if for every relevant variable x in ϕ, $\mathbf{s}(x) \neq null$. □

Definition 16. [11] (a) A tuple $(t_1, \ldots, t_n)$ with values in $\mathcal{U}$ is an answer from a database D under the *null query answering semantics* to a FO query $\mathcal{Q}$ with free variables $(x_1, \ldots, x_n)$ iff there exists a null-valid assignment $\mathbf{s}$ for $\mathcal{Q}$, such that $\mathbf{s}(x_i) = t_i$, for $i = 1, \ldots, n$, and $D \models_N^q \mathcal{Q}[\mathbf{s}]$. (b) $Ans^N(\mathcal{Q}, D)$ denotes the set of answers to $\mathcal{Q}$ obtained from database D under the semantics in (a). (c) If $\mathcal{Q}$ is a sentence (boolean query), the answer under the null query answering semantics is *yes* iff $D \models_N^q \mathcal{Q}$ and *no* otherwise. □

Example 2.9. Consider $D = \{P(1,1,1),\ P(2, null, null),\ P(null, 3, 3),\ Q(null),\ Q(1),\ Q(3)\}$, and the query $\mathcal{Q}$: $\exists yz(P(x,y,z) \wedge Q(y) \wedge y > 2)$.

For this query, the restricted relevant attribute is y, and the only free variable is x. $\mathbf{s_1}(x) = null$ is an answer to the query posed to D. In fact, the formula $\exists yz(P(x,y,z) \wedge Q(y) \wedge y > 2)$ is true in D, since there is a value for y which is not *null* and a value for z that make the formula true. Namely, $y = 3, z = 3$. So, it holds that $D \models_N^q \mathcal{Q}[null]$.

An assignment $\mathbf{s_2}(x) = 2$ is not an answer to the query. The formula $\exists yz(P(x,y,z) \wedge Q(y) \wedge y > 2)$ is not true in D, since the only value for y that makes the formula true is *null*.

Now, $\mathbf{s_3}(x) = 1$ is not an answer to the query sine there is not a value for y which is not *null* that make $y > 2$. In all, $Ans^N(\mathcal{Q}, D) = \{null\}$. □

In [11], it is shown that this null query answering semantics coincides with the classical first-order semantics for queries and databases without null values [12].

On the other hand, conjunctive queries can be syntactically transformed into new FO formulas for which the evaluation can be done by treating *null* as any other constant.

Definition 17. A query of the form (2.1) can be transformed into a new query $\mathcal{Q}_N$ written as $\exists \bar{y}(\bigwedge_{i=1}^{n} R_i(\bar{x}_i, \bar{y}_i) \wedge \varphi \wedge \bigwedge_{v \in \mathcal{V}^R(\mathcal{Q})} v \neq null)$, where $\mathcal{V}^R(\mathcal{Q})$ are *restricted relevant variables* (cf. Definition 12). In fact, this ensures that relevant variables are evaluated only over values in $(\mathcal{U} \setminus \{null\})$. Then, the new query $\mathcal{Q}_N$ can be evaluated treating *null* as any other constant in $\mathcal{U}$.

Example 2.10. (example 2.9 continued) The query $\mathcal{Q}$ in Example 2.9 can be rewritten as Q_N: $\exists yz(P(x,y,z) \wedge Q(y) \wedge y > 2 \wedge y \neq null)$.

To check if $D \models \mathcal{Q}_N(1)$, we need to check if $D \models \exists yz(P(1,y,z) \wedge Q(y) \wedge y > 2 \wedge y \neq null)$. Now, for $y = 1$ and $z = 1$, $D \models P(1,1,1)$ and $D \models Q(1)$, but $y = 1$ is less than 2. Therefore (1) is not an answer to the query $\mathcal{Q}_N$.

(2) is not an answer to the query $\mathcal{Q}_N$. Although $D \models P(2, null, null)$ and $D \models Q(null)$, $y = null$ contradicts with $y \neq null$. Therefore (2) is not an answer.

Now, (3) is an answer to the query $\mathcal{Q}_N$, since it holds that $D \models (P(null, 3, 3) \wedge Q(3) \wedge 3 > 2 \wedge 3 \neq null)$. □

2.3 Database Repairs and Consistent Query Answering

Given a inconsistent database instance D with a set IC of integrity constraints, we can restore consistency by deleting and/or inserting tuples. As a result, a new database instance, so-called *repair*, has the same schema as D that satisfies the IC and minimally differs from D under set inclusion. In order to formally define the repairs, we need a notion of distance between the database and its repaired version.

Definition 18. [5] Let D, D' be database instances over the same schema and domain. The distance, $\Delta(D, D')$, between D and D' is the symmetric difference $\Delta(D, D') = (D \smallsetminus D') \cup (D' \smallsetminus D)$. □

Definition 19. [12] Let D, D', D'' be database instances over the same schema and domain $\mathcal{U}$. It holds that $D' \leq_D D''$ iff :

(a) For every atom $P(\bar{a}) \in \Delta(D, D')$, with $\bar{a} \in (\mathcal{U} \smallsetminus \{null\})$, it holds that $P(\bar{a}) \in \Delta(D, D'')$.

(b) For every atom $P(\bar{a}, \overline{null}) \in \Delta(D, D')$, with $\bar{a} \in (\mathcal{U} \smallsetminus \{null\})$, there exists a $\bar{b} \in \mathcal{U}$, such that $P(\bar{a}, \bar{b}) \in \Delta(D, D'')$ and $P(\bar{a}, \bar{b}) \notin \Delta(D, D')$. □

Definition 20. [12] Given a database instance D, a set IC of ICs of the form (2.3), and NNCs of the form (2.7), a repair of D wrt IC is a database instance D' over the same schema of D, such that:

(a) $D' \models_N IC$,

(b) D' is $\leq_D$-*minimal* in the class of database instances that satisfy IC wrt $\models_N$, i.e. there is no database D'' in this class with $D'' <_D D'$. $D' <_D D''$ means $D' \leq_D D''$, but not $D'' \leq_D D'$.

The set of repairs of D wrt IC is denoted by $Rep(D, IC)$. □

Example 2.11. Consider the database instance $D = \{P(a)\}$, and the integrity constraint IC: $\forall x(P(x) \rightarrow \exists z Q(x, z))$. This database D is inconsistent wrt IC. Consistency can be restored by deleting $P(a)$ or inserting $Q(a, null)$. So there are two repairs: $D_1 = \{\}$ and $D_2 = \{P(a), Q(a, null)\}$. Notice that the database instance $D_3 = \{P(a), Q(a, b)\}$ is consistent wrt IC, but it is not a repair, we have $D_1 \leq_D D_3$, therefore D_3 is not $\leq_D$-*minimal*. □

A consistent answer to a FO query posed to a possibly inconsistent database D wrt a set IC of ICs is defined as follows:

Definition 21. [12] Given a database D and a set IC of ICs, a ground tuple $\bar{t}$ is a *consistent answer* to a query $\mathcal{Q}(\bar{x})$ wrt IC in D iff for every $D' \in Rep(D, IC)$, $D' \models^q_N \mathcal{Q}[\bar{t}]$. If $\mathcal{Q}$ is a sentence (boolean query), then *yes* is a consistent answer iff $D' \models^q_N \mathcal{Q}$ for every repair D' of D. Otherwise, the consistent answer is *no*. □

In this formulation of CQA we are using a notion $\models^q_N$ of answering first-order queries in databases with *null*.

2.4 Views

As indicated above, secret data is characterized as the contents of a view. In this research, we will restrict ourselves to a syntactic class of views.

Definition 22. A *view* V_s is defined by a Datalog rule of the form

$$V_s(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n),\ \varphi, \tag{2.12}$$

with $R_i \in \mathcal{R}, \bar{x} \subseteq \bigcup_i \bar{x}_i$, and $\bar{x}_i$ is a tuple of variables. Formula φ is a conjunction of built-ins atoms containing variables in R_is or domain constants. The conjunction in the body is denoted by $B(V_s)$. $\square$

Thus, in this thesis views are defined by conjunctive queries with built-in predicates. $V_s[D]$ denotes the extension of view V_s when computed on an instance D for Σ. Sometimes we simply use V_s if the instance is understood from the context.

Example 2.12. Consider the database instance $D = \{R(a,b), R(c,d), S(b,f), S(d,g), S(e,e)\}$, and the view by $V_s(x) \leftarrow R(x,y), S(y,z)$. Here, the data in the extension of the view $V_s[D] = \{(a),(c)\}$. Sometimes, to emphasize the predicate involved, we write $V_s[D] = \{V_s(a), V_s(c)\}$. $\square$

Our views can be expressed as $L(\Sigma)$-queries. More precisely, $B(V_s)$ in (2.12) can be written as the conjunctive query: $\exists \bar{y}\ (R_1(\bar{x}_1) \wedge \ldots \wedge R_n(\bar{x}_n) \wedge \varphi)$, with $\bar{x} = (\bigcup_i \bar{x}_i) - \bar{y}$.

Chapter 3

Null Semantics for SQL Queries and ICs

This research will address the problem of characterizing and obtaining *secret answers* to queries, i.e. that do not reveal secret information as specified by secrecy views over incomplete databases that may contain the single null value *null*, as `NULL` in SQL relational databases. In order to obtain *secret answers* in different DBMSs that (possibly only partially) follow the SQL standard that use null values in practice, there are several issues that need to be addressed.

First, we will make use of *null* to protect secret information contained in secrecy views. The idea is that the extensions of the secrecy views contain only tuples with null or become empty. Our view evaluation corresponds to conjunctive query evaluation, which will be based on the notion of the null query answering semantics (cf. Chapter 4). We need to show how different DBMSs interpret the null query answering semantics.

Second, we can have integrity constraints on the base schema. In this case, privacy has the extra condition of satisfying the ICs, because ICs could be violated through the process of updating the database instance with nulls. IC satisfaction with null values introduced in [12, 11] (cf. Section 2.1) give us a repairing strategy to restore consistency. Therefore, we need to show how different DBMSs interpret the IC satisfaction with null values, since we expect to adopt this semantics to restore consistency.

As a result of this chapter, we are able to apply the notion of the null query answering semantics in different DBMSs, and we also give some guidelines to adopt the semantics of IC satisfaction with null values proposed in [12, 11] in different DBMSs.

In this chapter, we do not consider secrecy instances or secret answers. These topics will be addressed in Chapter 4.

3.1 Interpretations of Null Values

A number of research [24, 26, 43, 41] has concentrated on the topic of incomplete databases and null values in relational databases. Incomplete databases in the classic sense that some information is represented using null values. Basically, a *null* is a place-holder for an attribute whose value can not be represented by an usual constant. Formal treatment of nulls usually include many interpretations, however, they can can be classified in two categories [43]. These are:

(a) `Unknown`: the attribute's value exists, but it is missing or unknown. For example, last name of an employee is missing because every employee has a last name or the salary of employee John is unknown.

(b) `Inapplicable`: the attribute's value is inapplicable or does not exist, i.e. salary for an employee, who just enters the company, is not valid during the first month. The *null* for salary is not applicable for at least one month. If an employee has no middle name, a special domain value *null* could be used to represent the middle name.

A formal treatment of null values under "unknown" interpretation was first proposed by Codd [17] and is defined by the usual *true* and *false*, and also the additional value *unknown*, which is used for *null*. In Codd's approach, a comparison expression, such as $X > 100$, evaluates to *true* or *false* in the usual way if X is not *null*. However, if X is *null*, then this expression evaluates to *unknown*. The truth tables for the negation, conjunction, and disjunction in three-valued logic (3VL) have also been addressed. Table 3.1 is the truth table for conjunction (*And*) while Table 3.2 is the true table for disjunction (*Or*). 3VL for negation (*Not*) is defined in the true table shown in Table 3.3.

Table 3.1: Truth Table for Conjunction

AND	True	False	Unknown
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

Table 3.2: Truth Table for Disjunction

OR	True	False	Unknown
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

Table 3.3: Truth Table for Negation

P	Not P
True	False
False	True
Unknown	Unknown

3VL was adopted in the SQL 2003 standard (SQL-2003) [1, 2]. In general, when a given query is evaluated, only those combinations of tuples that evaluate the logical expression of the query to *true* are selected as answers. In addition, SQL-2003 suggests the use of *Is Null* predicate when comparing values in a equality comparison against *null*. For example, a *Is Null* predicate tests whether one or more columns have null values. This predicate returns *true* if every value of attributes is *null*, otherwise it returns *false*.

The implemented semantics for query answering as specified in the SQL standard follows this three-valued logic, however it has been criticized in the database literature for its shaky logical basis. Some tuples will not be returned as answers to some tautological queries [43]. For example if we ask for all the employees with *salary* > 2000 or *salary* ≤ 2000, any employee whose salary is *null* will not be the answer. That is, if the expression p is *unknown*, then $\neg p$ is *unknown* as well, which in turn means $p \vee \neg p$ is *unknown* instead to *true*.

In addition, both *inapplicable* and *unknown* values are represented by null values. The SQL standard and DBMSs won't be able to distinguish them. Logically cleaner versions of null value semantics have been proposed by the research community, i.e. four-valued logic was proposed [22] to distinguish applicable and inapplicable data. But they have not been adopted by commercial systems due to the criticisms they have received.

The semantics introduced in [12, 11] provides a partial logical reconstruction in first-order predicate logic of the way nulls are handled in the SQL standard. More precisely, this semantics addresses the notion of IC satisfaction in the presence of *null* (cf. Section 2.1) and query answering for a broad classes of queries in databases with/without null values (cf. Section 2.2).

In our work, we will adopt this first-order reconstruction of *null*, because it captures the semantics of the SQL NULL that are relevant for our work on privacy, namely integrity constraint satisfaction and (conjunctive) query answering (cf. Section 4).

3.2 Null Interpretation in SQL queries

SQL query answering is the implemented semantics currently used in DBMSs for query answering, which is not an extension of the null query answering semantics. In order to check whether the null query answering semantics conforms to the SQL query answering in DBMSs, we basically compare answers of conjunctive queries given by DBMSs with the result of the null query answering semantics.

A first-order conjunctive query $\mathcal{Q}(\bar{x})$ of the form (2.1) can be always expressed as a SQL query $\mathcal{Q}^{SQL}$ as follows:

SELECT $x'_1, \ldots, x'_m$

FROM $R_1, \ldots, R_n$

WHERE $R_i.v = R_j.v$, AND...AND $R_l.u = R_k.u$, AND φ^{SQL}

Here, $\bar{x} = \{x_1, \ldots, x_m\}$, x'_i is an attribute associated with variable x_i in $\mathcal{Q}(\bar{x})$, the conditions in the WHERE clause represent the joins, and φ^{SQL} replaces each variable x in φ by its x' version.

We use the following running example to illustrate how the null query answering semantics is interpreted in DBMSs. This is done by creating tables and running queries in major commercial database management systems. In particular, we compare answers obtained by the null query answering semantics with those obtained by SQL query answering in Oracle 11g, IBM DB2 9.7 Enterprize Edition, Microsoft SQL Server 2008. In the following, we will use the abbreviations Oracle, DB2, and SQL Server to refer to above DBMSs.

Example 3.1. The employee and department data in Tables 3.4 and 3.5 will be used serval times throughout the rest of this chapter. Imagine a company database D storing information for employees and departments as shows in the following two tables. The attributes *Dept* and *Salary* in the *Employee* table may take null values. Also the attributes *Name* in the *Dept* table are allowed to contain null values.

Table 3.4: Employee Table

Eno	*Name*	*Dept*	*Salary*
100	*Mike*	1	40,000
101	*Mary*	1	35,000
102	*Tim*	1	*null*
103	*John*	2	42,000
104	*Anna*	*null*	45,000

Table 3.5: Dept Table

Deptno	*Name*
1	*HR*
2	*IT*
3	*Sales*
5	*Marketing*

□

The following example shows whether SQL query answering coincides with the null query answering semantics for a conjunctive query that contains joins and built-ins.

Example 3.2. Consider the conjunctive query $\mathcal{Q} : \exists xzwu(Employee(x,y,z,w) \wedge Dept(z,u) \wedge u = \text{“}IT\text{”}))$. It asks for the names of all employees who work at IT department. According to the null query answering semantics as presented in Section 2.2, $Ans^N(\mathcal{Q}, D) = \{(John)\}$. There are two ways to evaluate this query in DBMSs.

First, this query $\mathcal{Q}$ can be expressed as the following SQL query $\mathcal{Q}^{SQL}$, which is evaluated in DBMSs as usual:

```
SELECT Employee.Name
FROM   Dept, Employee
WHERE  Dept = Deptno
       AND
       Dept.Name=“IT”
```

This type of queries is very common in database praxis. If the query $\mathcal{Q}^{SQL}$ is executed in three DBMSs as usual, all of them return the tuple $\{(John)\}$. Therefore, we can see that the answers to a query of this kind obtained from both the null query answering semantics and SQL DBMSs coincide. In fact, they all implement 3VL for the boolean operator *And* as we showed above.

Furthermore, query $\mathcal{Q}$ can be transformed into a new SQL query $\mathcal{Q}_N^{SQL}$ which can be evaluated treating *null* as any other constant but making sure that variables in joins do not take the value *null*. This transformation is done by first rewriting query $\mathcal{Q}$ into a new query $Q_N : \exists xzwu(Employee(x,y,z,w) \wedge Dept(z,u) \wedge u = \text{“}IT\text{”} \wedge z \neq null \wedge u \neq null)$, according to Definition 17 in Chapter 2. Next, this query is expressed in SQL using SQL notation. The following is the new transformed SQL query $\mathcal{Q}_N^{SQL}$:

```
SELECT Employee.Name
FROM   Dept, Employee
WHERE  Dept = Deptno AND Dept.Name='IT'
       AND
       Dept ≠ 'Null' AND Deptno ≠ 'Null'
       AND Dept.Name ≠ 'Null'
```

For the purpose of evaluation of query $\mathcal{Q}_N^{SQL}$ on DBMSs treating *null* as any other constant, we modified the company database D by updating SQL NULL with the constant *Null*. We run this SQL query in our testing DBMSs, every DBMS returns the tuple $(John)$, as we expected. □

The following example shows whether SQL query answering coincides with the null query answering semantics for a conjunctive query that is open, and has built-ins.

Example 3.3. Consider the query $\mathcal{Q}$: $\exists xzw(Employee(x,y,z,w) \wedge w < 42000)$, that asks for the names of employees whose salary is less than 42,000. According to the null query answering semantics in Section 2.2, $Ans^N(\mathcal{Q},D) = \{(Mike),(Mary)\}$.

Again, query $\mathcal{Q}$ can be written as the following SQL query $\mathcal{Q}^{SQL}$, and then evaluate $\mathcal{Q}^{SQL}$ in DBMSs as usual:

```
SELECT Name
FROM   Employee
WHERE  Salary < 42000
```

The set of tuples $\{(\mathit{Mike}), (\mathit{Mary})\}$ are returned by running the query $\mathcal{Q}^{SQL}$ in our three DBMSs. Notice that (Tim) is not an answer to the SQL query. In this case, the answers given by SQL DBMSs and the null query answering semantics coincides, which means comparison *null* with any value will be evaluated as *unknown* in both commercial DBMSs and null querying answering semantics.

Similarly, we can transform query $\mathcal{Q}$ into a new SQL query $\mathcal{Q}_N^{SQL}$ for which evaluation can be done by treating *null* as any other constant.

```
SELECT Name
FROM   Employee
WHERE  Salary < 42000 AND
       AND
       Salary ≠ 'Null'
```

The set of tuples $\{(\mathit{Mike}), (\mathit{Mary})\}$ are returned by running the query $\mathcal{Q}_N^{SQL}$ in our three DBMSs. □

As indicated above, there is a very important predicate *Is Null* to test certain attributes from taking the value *null*.

Example 3.4. Consider the following conjunctive query $\mathcal{Q}$ with a *Is Null* predicate: $\exists xzw(\mathit{Employee}(x, y, z, w) \wedge \mathit{IsNull}(z))$, and the corresponding SQL query $\mathcal{Q}^{SQL}$:

```
SELECT Name
FROM   Employee
WHERE  Dept IS Null
```

For the null query answering semantics (cf. Section 2.2), it holds that $\mathit{Ans}^N(\mathcal{Q}, D) = \{(\mathit{Anna})\}$. A tuple containing *null* for *dept* will be the shown in the answer of the SQL query $\mathcal{Q}^{SQL}$. As for this case, (Anna) is the answer in all three DBMSs. If we insert tuple $(105, \mathit{Smith}, \ \ , 34,000)$ into *Employee* table by using the following SQL statement:

```
INSERT INTO Employee
VALUES (105,‘Smith’,’ ’,34000)
```

DB2 and SQL Server return the tuple $(Anna)$, while Oracle returns the set of tuples $\{(Anna), (Smith)\}$. In SQL-2003 and the null query answering semantics, *null* is not the same as an empty character string, i.e. a string of blank characters. However, Oracle treats a string value with a length of zero as *null*. $\square$

In the above examples, we are showing that, the null query answering semantics fully captures the way null values handled on DB2 and SQL Server for first-order conjunctive queries. However, for Oracle, since *null* means both “unknown ” as well as the zero-length empty string, answers to conjunctive queries in Oracle may be different from the null query answering semantics. Therefore, we need to treat the empty string as *null* in database in order to capture the semantics of *null* in Oracle. For example, the database instance $D = \{P(1,'\ \ '), Q(null)\}$ is equivalent to $D = \{P(1, null), Q(null)\}$ in Oracle.

3.3 IC Satisfaction in Databases with Null Values

In this section, we start with an overview of the semantics of IC satisfaction in the SQL standard SQL-2003. Then, we compare the semantics of IC satisfaction with null values provided in [12, 11] and major commercial database management systems that all follow SQL-2003. Here, we focus on the most important ICs available in major commercial databases, such as *unique*, *primary key*, and *foreign key* constraints. We therefore provide some starting points towards modifying the semantics for satisfaction of constraints in databases with null values in order to unify the semantics currently used in different DBMSs.

3.3.1 Semantics of Unique Constraints

According to SQL-2003, a unique constraint `UNIQUE(`$x_1, \ldots, x_n$`)` for a relation $R \in \mathcal{R}$ is satisfied iff there are no two rows t_1, t_2 in R such that the values of all their attributes x_i have the same non-null values. More formally [40],

$$\forall t_1 t_2 \in R : (\bigwedge_{i=1}^{n} (t_1.x_i \neq null \wedge t_2.x_i \neq null) \rightarrow \bigvee_{i=1}^{n} (t_1.x_i \neq t_2.x_i)) \tag{3.1}$$

Example 3.5. Consider a database with the table *Person*, where there is a unique constraint on both *Name* and *Phone* together, denoted by `UNIQUE(Name,Phone)`. The following database instance is accepted as a consistent state in SQL-2003.

Person	*Name*	*Phone*	*Age*
	Mary	000 – 1000	10
	Mary	000 – 1001	20
	null	000 – 1000	30

This is because *null* in *Name* attribute is not relevant to check IC satisfaction. More precisely, for these values in *Name* or *Phone* which are *null* make the antecedent of Formula (3.1) false. Therefore the whole formula is true. □

Example 3.6. To simplify the presentation, we further assume that the name of department has to be unique. The table of definition in Example 3.1 was extended by adding the following unique constraint:

```
ALTER TABLE Dept
ADD CONSTRAINT name_Unique UNIQUE(Name)
```
□

As discussed in Section 2.1.2, the `name_Unique` can be written as a set of rules according to the semantics proposed in [11]:

$$\forall xyz(Dept(y,x) \wedge Dept(z,x) \wedge IsNull(y) \rightarrow IsNull(z)),$$
$$\forall xyz(Dept(y,x) \wedge Dept(z,x) \rightarrow y = z).$$

In order to check if the company database D satisfies the `name_Unique`, by Definition 8, we need to check the following formulas:

$$D \models \forall xyz(Dept(y,x) \wedge Dept(z,x) \wedge y = null \rightarrow x = null \vee z = null). \tag{3.2}$$

$$D \models \forall xyz(Dept(y,x) \wedge Dept(z,x) \rightarrow x = null \vee y = null \vee z = null \vee y = z). \tag{3.3}$$

DB2 disallows the unique constraint `name_Unique`, since DB2 does not support unique constraints on columns that contain null values, while both Oracle and SQL Server allow it. The following insertion is accepted by Oracle, SQL server, and the semantics in [11]:

```
INSERT INTO Dept
VALUES(6, NULL)
```

However, if we try to insert another null value into *Name* in the *Dept* table, the insertion will be rejected by SQL Server due to violation of the unique constraint `name_Unique`. Basically, in SQL Server, if a unique constraint is defined upon a column containing *null*, only one null value will be allowed in that column, while in Oracle and [11] a unique constraint allows more than one null values. Therefore, the semantics of a unique constraint defined in Oracle is less restrictive. More formally, `UNIQUE(`$x_1, \ldots, x_n$`)` for a relation R holds in DB2 iff the following holds [40]:

$$\forall t \in R : (\bigwedge_{i=1}^{n} t.x_i \neq null) \wedge \forall t_1 t_2 \in R : (\bigvee_{i=1}^{n} t_1.x_i \neq t_2.x_i). \tag{3.4}$$

`UNIQUE(`$x_1, \ldots, x_n$`)` for a relation R holds in SQL Server iff the following holds:

$$\forall t_1 t_2 \in R : (\bigvee_{i=1}^{n} t_1.x_i \neq t_2.x_i)^1. \tag{3.5}$$

[1]Here, null is treated as a database constant value.

`UNIQUE`$(x_1,\dots,x_n)$ for a relation R holds in Oracle iff the following holds [40]:

$$\forall t_1 t_2 \in R : (\bigvee_{i=1}^{n}(t_1.x_i \neq null \vee t_2.x_i \neq null) \rightarrow \bigvee_{i=1}^{n} t_1.x_i \neq t_2.x_i). \tag{3.6}$$

3.3.2 Semantics of Primary Key Constraints

In SQL-2003, a primary key constraint is a combination of a unique constraint and one or more not null constraints. A primary key constraint `PRIMARY KEY`$(\mathtt{x_1},\dots,\mathtt{x_n})$ for a relation $R \in \mathcal{R}$ is satisfied iff the following formula holds [40]:

$$\forall t \in R : (\bigwedge_{i=1}^{n} t.x_i \neq null) \wedge \forall t_1 t_2 \in R : (\bigvee_{i=1}^{n} t_1.x_i \neq t_2.x_i) \tag{3.7}$$

In DB2, SQL Server, Oracle, and [11] the definition of a primary key implicitly defines not null constraints on the corresponding uniqueness columns which coincides with the semantics of a primary key proposed in SQL-2003, as shown in the following example:

Example 3.7. (example 3.6 continued) The table of definition in Example 3.6 was extended by adding the following primary key:

```
ALTER TABLE Dept
ADD CONSTRAINT deptno_PK PRIMARY KEY(Deptno)
```
□

According to [11], the primary key can be written as $\forall xyz(Dept(x,y) \wedge Dept(x,z) \rightarrow y = z)$, $\forall xyz(Dept(x,y) \wedge Dept(x,z) \wedge IsNull(y) \rightarrow IsNull(z)$, and the NNC $\forall xyz(Dept(x,y) \wedge IsNull(x) \rightarrow \mathbf{false})$.

In order to check if the company database D satisfies the `deptno_PK`, we need to check the following formulas (cf. Section 2.1.2):

$$D \models \forall xyz(Dept(x,y) \wedge Dept(x,z) \wedge y = null \rightarrow x = null \vee z = null). \tag{3.8}$$

$$D \models \forall xyz(Dept(x,y) \wedge Dept(x,z) \rightarrow x = null \vee y = null \vee z = null \vee y = z). \tag{3.9}$$

$$D \models \forall xyz(Dept(x,y) \wedge x = null \rightarrow \mathbf{false}). \tag{3.10}$$

The following insertion is rejected due to a violation of the primary key `deptno_PK` in DB2, Oracle and SQL Server.

```
INSERT INTO Dept
VALUES(NULL,'Consulting')
```

The `deptno_PK` will be violate the primary key semantics proposed in [11] if there is tuple $(null, consulting)$ in the *Dept* table. This is because the tuple $Dept(null, consulting)$ will not be satisfied Formula (3.10).

3.3.3 Semantics of Foreign Key Constraints

Referential constraints permit the comparison of values from different columns of one or more relations. A referential constraint (or an inclusion dependency), denoted by $R[X] \subseteq S[Y]$ (cf. Section 2.1), is satisfied, if every value of X in R that refers to a value of the corresponding attribute in Y in the relation S.[2] Here, R is the referencing table and S is the referenced table. In the case of referential constraints, SQL-2003 suggests three different semantics, namely *simple*, *partial*, and *full*, respectively.

If it is *simple-match* semantics [2] for each row t_1 of the referencing table R, either at least one of the values of the referencing columns in R shall be a null value, or the value of each referencing column in R shall be equal to the value of the corresponding referenced column in some row of the referenced table S.

If *partial-match* semantics [2] is specified, for each row t_1 of the referencing table R, there shall be some row t_2 of the referenced table S such that the value of each referencing column in t_1 is either *null* or is equal to the value of the corresponding referenced column in t_2.

If *full-match* semantics [2] is specified, for each row t_1 of the referencing table R, either the value of every referencing column shall be a null value, or the value of every referencing column in t_1 shall not be *null* and there shall be some row t_2 of the

[2]Referential constraints could be defined on the same table.

referenced table such that the value of each referencing column in t_1 is equal to the value of the corresponding referenced column in t_2.

Example 3.8. Consider a referenial constraint ψ: $R(X,Y) \subseteq S(X,Y)$ and the database $D = \{R(null,a), S(a,a)\}$. (a) D satisfies ψ wrt simple match, since there is a null value in the referencing table R. (b) D does not satisfy ψ wrt partial match, since there is no tuple $R(a,a)$ or $R(null,null)$ in R. (c) D does not satisfy ψ wrt full match, since any of null values can not be in the referencing table R. □

Foreign key is a special case of a referential constraint. It requires the primary key on the referenced table.

Example 3.9. In order to check the semantics of foreign keys, our company database was extended by assuming the a employee's department must be a department of the company. The table definition of Example 3.7 was extended by:

```
ALTER TABLE Employee
ADD CONSTRAINT fk FOREIGN KEY(Dept)
                    REFERENCES Dept(Deptno)
```
□

In [11], the `fk` can be written as $\psi : \forall xyzw(\mathit{Employee}(x,y,z,w) \rightarrow \exists u \mathit{Dept}(z,u))$. Variable z is relevant to check the constraint, therefore the set of relevant attributes is $\mathcal{A}(\psi) = \{\mathit{Employee}[3], \mathit{Dept}[1]\}$. To check the `fk`, by Definition 8, we need to check $D \models \forall z(\mathit{Employee}^{\mathcal{A}}(z) \rightarrow z = null \vee \mathit{Dept}^{\mathcal{A}}(z))$, where $\mathit{Employee}^{\mathcal{A}}$ and $\mathit{Dept}^{\mathcal{A}}$ are the predicates projected onto the relevant attributes. In this example, the company database D satisfies the foreign key constraint `fk`.

A possible way to check satisfaction of an integrity constraint in DBMSs is by means of *violation view* [23, 38]. If the integrity constraint is satisfied, then violation view is empty, otherwise it contains the set of tuples that violates the constraint.

Example 3.10. (example 3.9 continued) For the foreign key in Example 3.9, the following query $\mathcal{Q}$ defines the violation view for databases with null values, and is used to check the satisfaction of `fk`:

```
SELECT Dept
FROM   Employee
WHERE  NOT EXISTS (SELECT * FROM
                   Dept
                   WHERE Deptno=Dept)
AND Dept IS NOT NULL
```

By running this query $\mathcal{Q}$ in three DBMSs, result of query $\mathcal{Q}$ is empty in every DBMS. Therefore, in all DBMSs, null values in referencing columns do not violate foreign key constraints. □

In this section, we focused on *unique*, *primary*, and *foreign key* constraints satisfaction in the presence of *null* supported by DB2, Oracle, SQL Server, and the semantics proposed [11]. We give the results of observations from the above examples:

- *Unique key*: This type of integrity constraint is supported by all testing systems and the semantics in [11]. Notice that Oracle and the semantics in [11] support multiple null values in the unique key column whereas SQL Server allows only one null value in the unique key column. In SQL Server, this is contradiction of null comparison of 3VL in SQL-2003, this is because, *null* = *null* returns *true* instead of *unknown*.

- *Primary key*: All testing systems and the semantics in [11] support the primary key constraint which coincides with semantics of the primary key proposed in

SQL-2003.

- *Foreign key*: All testing systems and the semantics in [11] implement only the *simple-match semantics* for foreign key constraints suggested in SQL-2003, which allows to contain null values in referencing columns. They do not support the *partial-match* and *full-match* semantics suggested in SQL-2003 [12, 40].

In [11], they present a repair semantics to restore consistency in the context of incomplete databases, which is based on the notion of IC satisfaction proposed in [11]. In our privacy work, we will make use of this semantics to obtain consistent and secret answers on the top of different DBMSs, therefore this semantics should be compatible with the way null values are treated in DBMSs. Above investigations give us some starting points on modifying this semantics in order to unify the semantics currently used in different DBMSs.

Chapter 4

A Semantics for Privacy with Uncertain Data

The privacy problem, as illustrated in Chapter 1, will be solved by performing a minimal set of virtual changes on the database instances, which captures the ideas of: (a) Having a class of virtual secrecy instances for a database that protects secret data as defined by the secrecy views; (b) Minimizing the distance between a secrecy instance and the given database instance, and this minimization takes into account the occurrence of null values. (c) Computing answers to a query that do not reveal any secret information.

In this chapter, we start in Section 4.1 with the semantics of secrecy in the presence of *null*, and we also characterize a class of secrecy instances. Then we introduce the precise definition of *secret answers* to first-order conjunctive queries in Section 4.2. In this research we restrict ourselves to conjunctive queries. An extension of our semantics for queries containing negation is left for future work.

4.1 Secrecy Instances of Incomplete Databases

In this work we will make use of *null* to protect secret information. As described in Section 3.2, the semantics of *null* will (essentially) correspond to the way nulls are handled by DBMSs that follow the SQL standard. The idea is that the extensions of the secrecy views will contain only tuples with *null* or will become empty. Our view evaluation problem corresponds to conjunctive query evaluation, which will be based on the notion of the null query answering introduced in Section 2.2.

Example 4.1. (example 2.9 continued) Consider the following instance D:

P	X	Y	Z
	1	1	1
	2	*null*	*null*
	null	3	3

Q	Y
	null
	1
	3

and the secrecy view $V_s(x) \leftarrow P(x,y,z), Q(y), y > 2$. This secrecy view can be expressed as the first-order query:

$$\mathcal{Q}^{V_s}(x) : \exists yz(P(x,y,z) \wedge Q(y) \wedge y > 2),$$

which is the same query as in Example 2.9.

Under the semantics of secrecy in the presence of *null*, we expect that the extensions of the secrecy views contain tuples only with *null* or become empty. This implies that the values in attribute X associated with variable x in $\mathcal{Q}^{V_s}$ are *null*, or the values in Y associated with variable y in $\mathcal{Q}^{V_s}$ are *null*, or the negation of the comparison is *true*. These three cases correspond to the three assignments in Example 2.9. Notice that the attributes X and Y play an important role for the semantics of secrecy.□

The above example shows that the occurrence of *null* in some attributes are crucial to gain the view extensions. In the following, we introduce the notion of null-special attributes (NSA) for a secrecy view to capture the position *null* in attributes, i.e. those that associate with variables involved in joins, built-ins, and the head of the view. This is because the position *null* in joins and built-ins will make the view extensions empty, while the position *null* in the head will make the view extensions contain tuples only with *null*.

Definition 23. For a Datalog rule of the form (2.12) and a term t (i.e. a variable or constant), let $pos^R(V_s, t)$ be the set of positions in predicate $R \in \mathcal{R}$ where t appears.□

Example 4.2. Consider a secrecy view: $V_s(x,z) \leftarrow P(x,y), Q(y,z)$. For a variable y, $pos^R(V_s, y) = \{2,1\}$, which means that y appears in the second position in P and in the first position in Q. The notion of pos^R in Definition 23 is essentially the same as the one presented in Chapter 2 Definition 6. □

Definition 24. The set of *combination attributes* for a secrecy view V_s in the form of (2.12) is: $\mathcal{C}(V_s) = \{R[i] \mid x$ is a variable appearing at least twice in the body of (2.12), and $i \in pos^R(V_s, x)\}$. □

Here, $R[i]$ denotes a position (or the correspondent attribute) in relation R. In fact, combination attributes of a secrecy view V_s are those involved in joins or built-in predicates, and they also correspond to the *restricted relevant variables* (cf. Definition 12) of the associated query $\mathcal{Q}^{V_s}$.

Example 4.3. (example 4.2 continued) The secrecy view in Example 4.2 can be expressed as the first-order query:

$$\exists y(P(x,y) \wedge Q(y,z)). \tag{4.1}$$

Since y appears twice in the body of the secrecy view, by Definition 24, $\mathcal{C}(V_s) = \{P[2], Q[1]\}$. According to Definition 12, the *restricted relevant variable* of query (4.1) is y. We can see that the *restricted relevant variable* y corresponds to *combination attributes* of the secrecy view. □

Definition 25. The set of *secrecy attributes* for a secrecy view V_s as in (2.12) is: $\mathcal{S}(V_s) = \{R[i] \mid x$ is a variable in the head of (2.12), and $i \in pos^R(V_s, x)\}$. □

Here, secrecy attributes of a secrecy view V_s are those appearing in the head of V_s's definition. They correspond to the *free variables* in the associated query $\mathcal{Q}^{V_s}$.

Example 4.4. (example 4.3 continued) Since x and z appear in the head of the secrecy view in Example 4.2, by Definition 25, $\mathcal{S}(V_s) = \{P[1], Q[2]\}$. The *free variables* of query (4.1) are x and z, which correspond to *secrecy attributes* of the associated secrecy view. □

Definition 26. The set of *null-special attributes* for a secrecy view V_s are those (associated to positions) in the set $\mathcal{N}(\mathcal{V}^s) = \mathcal{C}(V_s) \cup \mathcal{S}(V_s)$. □

Example 4.5. (example 4.1 continued) Consider the same secrecy view in Example 4.1: $V_s(x) \leftarrow P(x, y, z), Q(y), y > 2$. Since y appears three times in the body, by Definition 24, $\mathcal{C}(V_s) = \{R[2], S[1]\}$. Since x appears in the head, $\mathcal{S}(V_s) = \{R[1]\}$. Therefore, $\mathcal{N}(V_s) = \{R[1], R[2], S[1]\}$. The values in attribute Z are not crucial for gaining the view extensions, no matter whether they are *null* or not. This makes sense, because we only join tuples via attribute Y, and making sure that values in attribute Y are greater than 2. And then, we project values on attribute X. □

Definition 27. A database instance is *admissible* for a set $\mathcal{V}^s$ of secrecy views of the form (2.12), denoted $D \in Admiss(\mathcal{V}^s)$, if under the $\models_N^q$ semantics, each $V_s[D]$ is empty or in all its tuples only *null* appears. □

From the example above, we can see that, given a database instance D and a secrecy view V_s of the form (2.12), D is admissible, when one of the following three cases occurs: (a) There is a *null* in any of the combination attributes; (b) Values in the secrecy attributes are all *null*; (c) The negation of the built-ins is true.

Before making this claim formal, let us recall that, given a set of attributes $\mathcal{A}$, and a predicate $R \in \mathcal{R}$, we denote by $R^{\mathcal{A}}$ the predicate R projected onto the attributes in $\mathcal{A}$. $D^{\mathcal{A}}$ denotes the database D with all its database atoms projected onto the attributes in $\mathcal{A}$ (cf. Definition 7).

Proposition 1. Let $\mathcal{V}^s$ be a set of secrecy views V_s of the form (2.12). Let $\mathcal{Q}^{V_s}$ be their expressions as conjunctive queries of the form $\exists \bar{y}(\bigwedge_{i=1}^n R_i(\bar{x}_i) \land \varphi)$. For an instance D, $D \in \mathit{Admiss}(\mathcal{V}^s)$ iff $D^{\mathcal{N}(V_s)} \models \mathcal{Q}_N^{V_s}$, for each $V_s \in \mathcal{V}^s$, where $\mathcal{Q}_N^{V_s}$ is

$$\forall \bar{x}(\bigwedge_{i=1}^{n} R_i^{\mathcal{N}(V_s)}(\bar{x}_i) \ \rightarrow \bigvee_{v_j \in (\mathcal{C}(V_s) \cap \bar{x})} v_j = \mathit{null} \ \lor \bigwedge_{u_j \in (\mathcal{S}(V_s) \cap \bar{x})} u_j = \mathit{null} \ \lor \ \neg\varphi),$$

where $\bar{x} = \bigcup_{i=1}^n \bar{x}_i$. Here, $D^{\mathcal{N}(V_s)} \models \mathcal{Q}_N^{V_s}$ refers to classical first-order satisfaction and *null* is treated as any other constant. □

Example 4.6. (example 4.5 continued) According to the above definition, in order to check whether the database instance D is admissible, the following must hold:

$$D^{\mathcal{N}(V_s)} \models \forall xy(R^{\mathcal{N}(V_s)}(x,y) \land S^{\mathcal{N}(V_s)}(y) \rightarrow y = \mathit{null} \lor x = \mathit{null} \lor y <= 2),$$

with $D^{\mathcal{N}(V_s)}$ given below. When checking this, *null* is treated as any other constant in $\mathcal{U}$.

$R^{\mathcal{N}(V_s)}$	X	Y
	1	1
	2	*null*
	null	3

$S^{\mathcal{N}(V_s)}$	Y
	null
	1
	3

For $x = 1, y = 1$, the antecedent of the implication is satisfied since $R^{\mathcal{N}(V_s)}(1,1) \in D^{\mathcal{N}}$, $S^{\mathcal{N}(V_s)}(1) \in D^{\mathcal{N}}$. For these values, the consequent is also satisfied, because $y = 1 < 2$. For $x = 2, y = \mathit{null}$, the consequent is satisfied since y is *null*. For $x = \mathit{null}$, $y = 3$, the antecedent is satisfied since $R^{\mathcal{N}(V_s)}(\mathit{null},3) \in D^{\mathcal{N}}$, $S^{\mathcal{N}(V_s)}(3) \in D^{\mathcal{N}}$. For these values, the consequent is also satisfied, because $\mathit{null} = \mathit{null}$ is true. In this example, $D \models_N \mathcal{Q}^{V_s}$. Therefore, the given instance D is admissible. □

Proof of Proposition 1: First we proof that if $D \in Admiss(\mathcal{V}^s)$, then each $V_s[D]$ is empty or in all its tuples only *null* appears under the $\models^q_N$ semantics. Here, V_s is of the form (2.12), and $D \models_N \mathcal{Q}^{V_s}$ implies that $D^{\mathcal{N}(V_s)} \models \mathcal{Q}^{V_s}_N$, where $\mathcal{Q}^{V_s}_N$:

$$\forall \bar{x}(\bigwedge_{i=1}^{n} R_i^{\mathcal{N}(V_s)}(\bar{x}_i) \rightarrow \bigvee_{v_j \in (\mathcal{C}(V_s) \cap \bar{x})} v_j = null \vee \bigwedge_{u_j \in (\mathcal{S}(V_s) \cap \bar{x})} u_j = null \vee \neg\varphi). \qquad (4.2)$$

By contradiction, we assume that there exists an assignment $\bar{c} = (c_1, ..., c_n)$ for free variables $\bar{x} = (x_1, ..., x_n)$ in $\mathcal{Q}^{V_s}$[1], such that the following three statements hold:

1. If $x_i \in \mathcal{V}^R(\mathcal{Q}^{V_s})$, then $c_i \in (\mathcal{U} \setminus \{null\})$, otherwise, $c_i \in \mathcal{U}$.

2. There exists at least one $i \in (1, \ldots, n)$, such that $c_i \neq null$.

3. $D \models^q_N \mathcal{Q}^{V_s}[\bar{s}[\bar{x}|\bar{c}]]$.

$D \models^q_N \mathcal{Q}^{V_s}[\bar{s}[\bar{x}|\bar{c}]]$ implies that $D \models^q_N \exists \bar{z}\bar{w}(\bigwedge_{i=1}^{n} R_i(\bar{c}_i, \bar{z}_i, \bar{w}_i) \wedge \varphi[\bar{s}[\bar{x}|\bar{c}]])$, where $\bar{z} = \bigcup_{i=1}^n \bar{z}_i$, $\bar{w} = \bigcup_{i=1}^n \bar{w}_i$, and $\bar{z} \cup \bar{w} = \bar{y}$. Here, variables in $\bar{z}$ are *restricted relevant variables* in $\mathcal{Q}^{V_s}$, while no variable in $\bar{w}$ is relevant. By Definition 14, for all $\bar{a} \in (\mathcal{U} \setminus \{null\})$ and $\bar{d} \in \mathcal{U}$, it holds that $D \models^q_N \bigwedge_{i=1}^n R_i(\bar{c}_i, \bar{a}_i, \bar{d}_i) \wedge \varphi$. This indicates that $D \not\models \bigvee_{v_j \in \mathcal{V}^R(\mathcal{Q}^{V_s})} v_j = null$, and $D \models \varphi$. Therefore, $D^{\mathcal{N}(V_s)} \not\models \bigvee_{v_j \in (\mathcal{C}(V_s) \cap \bar{x})} v_j = null$. Since all *free variables* $\bar{x}$ in $\mathcal{V}^R(\mathcal{Q}^{V_s})$ are associated with *secrecy attributes* in V_s, and we have that $c_i \neq null$ for at least one $i \in (1, ..., n)$. Therefore, it holds that $D^{\mathcal{N}(V_s)} \not\models \bigwedge_{u_j \in (S(V_s) \cap \bar{x})} u_j = null$. Thus, we end up a contradiction with $D \models_N \mathcal{Q}^{V_s}$.

Now, Let us prove that under the $\models^q_N$ semantics, $V_s[D]$ is empty or in all its tuples only *null* appears, then $D \models_N \mathcal{Q}^{V_s}$.

1. First, we consider that $V_s[D]$ are tuples containing only null values. By Definition 14, there exists an assignment $\bar{c} = (c_1, ..., c_i)$, where $\bigwedge_{i=1}^n c_i = null$ for free

[1]As indicated in Section 2.4, V_s can be can be written as the conjunctive query: $\exists \bar{y}\ (R_1(\bar{x}_1) \wedge \ldots \wedge R_n(\bar{x}_n) \wedge \varphi)$, with $\bar{x} = (\bigcup_i \bar{x}_i) - \bar{y}$.

variables $\bar{x} = (x_1, ..., x_n)$ in $\mathcal{Q}^{V_s}$. Since all *free variables* $\bar{x}$ are associated with *secrecy attributes* in V_s, it holds that $D^{\mathcal{N}(V_s)} \models \bigwedge_{u_j \in (S(V_s) \cap \bar{x})} u_j = null$ which makes $D \models_N \mathcal{Q}^{V_s}$ true.

2. Now, $V_s[D]$ is the empty set. By definition 15, there is no null-valid assignment with respect to $\mathcal{Q}^{V_s}$, such that $v_j \neq null$ for every $v_j \in \mathcal{Q}^{V_s}$. This means that there is at least one *restricted relevant attribute* v_j, such that $v_j = null$. Therefore, $D^{\mathcal{N}(V_s)} \models \bigvee_{v_j \in (C(V_s) \cap \bar{x})} v_j = null$ which makes $D \models_N \mathcal{Q}^{V_s}$ true. □

Virtual updates based on *null* will be used to obtain admissible instances starting from a given instance D. A *secrecy instance* for D will be admissible and will also minimally differ from D. The latter condition requires a suitable comparison of database instances. As expected, this comparison will consider the presence of *null* in the tuples of those instances.

Definition 28. [32] (less or equal informative constants) We define a partial order in database domain $\mathcal{U}$, denoted by $\sqsubseteq$, as follows: $c \sqsubseteq d$ iff $c = d$ or $c = null$, where $c, d \in \mathcal{U}$.□

This partial order implies that constant c provide less or equal information than a constant d in database. Now, we extend $\sqsubseteq$ as a partial order on tuples.

Definition 29. [32] (less or equal informative tuples) Let $t_1 = (c_1, \ldots, c_n)$ and $t_2 = (d_1, \ldots, d_n)$ be finite sequences of constants. t_1 is less or equal informative than t_2, denoted by $t_1 \sqsubseteq t_2$ iff: $c_i \sqsubseteq d_i$ for $\forall i \in \{1, \ldots, n\}$. Also, t_1 is less informative than t_2, denoted by $t_1 \sqsubset t_2$ iff $t_1 \sqsubseteq t_2$ and $t_1 \neq t_2$. □

For example, tuple $(a, null)$ provides less information than tuple (a, b). Then $(a, null) \sqsubsetneq (a, b)$ holds.

Definition 30. Let D, D', D'' be database instances over the same schema Σ and domain $\mathcal{U}$. It holds that $D' \leq_D D''$ iff for every tuple $t = R(\bar{a}) \in D'$, there is a unique tuple t' such that $t' \in D''$ and $t' \sqsubseteq t$. $D' <_D D''$ means $D' \leq_D D''$, but not $D'' \leq_D D'$ □

Here, D' and D'' are usually instances obtained by possible value updates on D using *null*. If $D' <_D D''$, we say that D' is closer to D than D''. We may assume that tuples have identifiers, so the unique tuple t' above will be denoted by $\mu(t)$. Therefore, it is possible to correlate the original tuples with those in the *null*-based updated instance. The tuples t, t' in Definition 30 will be then correlated tuples.

Definition 31. Given a database instance D and a set $\mathcal{V}^s$ of views V_s of the form (2.12), a *secrecy instance* D_s for D wrt $\mathcal{V}^s$ is an instance over the same schema, such that: (a) $D_s \in \mathit{Admiss}(\mathcal{V}^s)$, and (b) D_s is $\leq_D$-*minimal* in the class of database instances that satisfy (a) and share the schema with D, i.e. there is no instance D'_s in that class with $D'_s <_D D_s$. $\mathit{Sec}(D, \mathcal{V}^s)$ denotes the set of all the secrecy instances for D wrt $\mathcal{V}^s$. □

Example 4.7. Given the instance $D = \{P(1,2), R(2,1)\}$, and the *secrecy view*: $V_s(x,z) \leftarrow P(x,y), R(y,z), y < 3$, consider the alternative updated instances D_i.

Notice that, all updated instances D_1, D_2, D_3 and D_4 satisfy the first condition, of admissibility, of *secrecy instance* in Definition 31, that is:

i	D_i
1	$\{P(\mathit{null},2), R(2,\mathit{null})\}$
2	$\{P(1,\mathit{null}), R(2,1)\}$
3	$\{P(1,2), R(\mathit{null},1)\}$
4	$\{P(1,\mathit{null}), R(\mathit{null},1)\}$

$D_i^{\mathcal{N}(V_s)} \models \forall xy(P^{\mathcal{N}(V_s)}(x,y) \wedge R^{\mathcal{N}(V_s)}(y,z) \rightarrow y = \mathit{null} \vee x = \mathit{null} \wedge z = \mathit{null} \vee y \geq 3)$, with $D_i^{\mathcal{N}(V_s)} = \{P^{\mathcal{N}(V_s)}(1,2),\ R^{\mathcal{N}(V_s)}(2,1)\}$. However, D_4 is not a *secrecy instance*, because $P(1,\mathit{null}) \sqsubset P(1,2)$; and then, $D_3 <_D D4$. Thus, D_1, D_2, and D_3 are the only three secrecy instances of original instance D. $\square$

Secrecy instances for a given instance D return uninformative answers (i.e. empty or nulls) when asked about the secrecy view contents. Furthermore, we do not want to trivialize too much of the original information in D. For this reason we make the secrecy instances minimally depart from D wrt a notion of distance that privileges taking into account the null values. In particular, these requirements enforce updates on D that are based only changes of constants by *null*.

4.2 Privacy Preserving Query Answering

Our task is to compute secret answers to queries from a given database D that contains secret information. The answers to be returned will be obtained by querying secrecy instances of D instead of directly querying D. There may be several secrecy instances and each of them will return trivial answers when asked about the secrecy views. We may consider this collection of secrecy instances as representing some sort of *logical database* (given through its models), and in consequence, the expected answers are those that are true of all the chosen instances. These are usually called *certain answers* [24].

Definition 32. Let $\mathcal{Q}(\bar{x}) \in L(\Sigma)$ be a conjunctive query. A tuple $\bar{a}$ 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 $D_s \models_N^q \mathcal{Q}[\bar{a}]$ for each $D_s \in Sec(D, \mathcal{V}^s)$. $SA(\mathcal{Q}, D, \mathcal{V}^s)$ denotes the set of all secret answers. $\square$

Example 4.8. (example 4.7 continued) Consider $\mathcal{Q}(x,z) : \exists y(P(x,y) \wedge R(y,z) \wedge y < 3)$. For the alternative secrecy instances, we obtain: $\mathcal{Q}(D_1) = \{(null, null)\}$, $\mathcal{Q}(D_2) = \varnothing$, and $\mathcal{Q}(D_3) = \varnothing$. These answers can be obtained by transforming $\mathcal{Q}$ into a new query $\mathcal{Q}_N$ using a methodology proposed in Definition 17. In this case, we obtain

$\mathcal{Q}_N(x,z) : \exists y(P(x,y) \land R(y,z) \land y < 3 \land y \neq \mathit{null})$. This query can be evaluated on each of the secrecy instances treating *null* as any other constant. Finally, we obtain $SA(\mathcal{Q}, D, \{V_s\}) = \varnothing$.

Notice that, as expected, $SA(\mathcal{Q}^{V_s}, D) = \varnothing$, where $\mathcal{Q}^{V_s}$ is the query associated to the secrecy view definition. $\square$

Secret answers (SAs) are based on a *skeptical* or *cautious* semantics that considers what is true of all the secrecy instances. A more relaxed alternative is a *possible* or *brave* semantics, which considers an answer as valid if it is an answer from at least one of the secrecy instances. For instance, in Example 4.8, the tuple $P(1,2)$ is a *secret answer* to the query $P(x,y)$ under the brave semantics, since $P(1,2)$ is an answer to $P(x,y)$ on the secrecy instance D_3. Similarly, $R(2,1)$ is a *secret answer* to $R(x,y)$ under the brave semantics.

Notice that the possible answers, that obtained from some secrecy instance, give us more information from the original database than the cautious answers. However, they do not help to solve the privacy problem, because from the secret answers $P(1,2)$ and $R(2,1)$, the user will immediately obtain the contents of the secrecy view.

Even under the skeptical semantics, the user may try to pose queries to obtain sensitive information, as the following example shows.

Example 4.9. Consider the new database instance $D = \{P(1,2), P(3,4), R(2,1), R(3,3)\}$, and the secrecy view $V_s(x,z) \leftarrow P(x,y), R(y,z)$. This database D has three secrecy instances, indicated in the following table:

i	D_i
1	$\{P(\mathit{null},2), P(3,4), R(2,\mathit{null}), R(3,3)\}$
2	$\{P(1,\mathit{null}), P(3,4), R(2,1), R(3,3)\}$
3	$\{P(1,2), P(3,4), R(\mathit{null},1), R(3,3)\}$

The user may pose the queries $\mathcal{Q}_2(x,y) : P(x,y)$ and $\mathcal{Q}_3(x,y) : R(x,y)$ to try to reconstruct the original database D. For the query $\mathcal{Q}_2(x,y)$, it holds $\mathcal{Q}_2(D_1) = \{(null,2),(3,4)\}$, $\mathcal{Q}_2(D_2) = \{(1,null),(3,4)\}$, and $\mathcal{Q}_2(D_3) = \{(1,2),\ (3,4)\}$. In consequence, $SA(\mathcal{Q}_2, D, \{V_s\}) = \{(3,4)\}$.

For the query $\mathcal{Q}_3(x,y)$, it holds $\mathcal{Q}_3(D_1) = \{(2,null),(3,3)\}$, $\mathcal{Q}_3(D_2) = \{(2,1),(3,3)\}$, and $\mathcal{Q}_3(D_3) = \{(null,1),\ (3,3)\}$. In consequence, $SA(\mathcal{Q}_3, D, \{V_s\}) = \{(3,3)\}$.

By combining the secret answers to the subqueries $\mathcal{Q}_2$ and $\mathcal{Q}_3$ of $\mathcal{Q}^{V_s}$, where $\mathcal{Q}^{V_s}$ is the query that defines the secrecy view, it is not possible for the user to obtain the extension $\{(1,1)\}$ of $V_s[D]$. Actually, if the user poses the queries $\mathcal{Q}_2$ and $\mathcal{Q}_3$, for him the relations would look like the following:

P	X	Y
	3	4

R	Y	Z
	3	3

In this case, any other conjunctive query posed to detect the presence of initial nulls, like $\exists y(P(x,y) \wedge x = null)$, will get an empty set of secret answers, and the user will not know anything more about the contents of the original instance. □

Definition 33. Let $\mathcal{V}^s$ be a set of secrecy views V_s of the form (2.12). The *secrecy answer instance* for $\mathcal{V}^s$ from D is defined by $D_{\mathcal{V}^s} = \{R_i(\bar{t}) \mid R_i \in \mathcal{R}$ and $\bar{t} \in SA(R_i(\bar{x}_i), D, \mathcal{V}^s)\}$ □

In Definition 33 we are building a database instance by collecting the SAs to all the atomic queries of the form $R(\bar{x})$, with $R \in \mathcal{R}$. This instance has the same schema as the original instance D.

Example 4.10. (example 4.9 continued) Consider the secrecy view $V_s(x,z) \leftarrow P(x,y),\ R(y,z)$. It holds: $D_{\mathcal{V}^s} = \{P(3,4)\} \cup \{R(3,3)\} = \{P(3,4), R(3,3)\}$. □

Suppose there is a set of secrecy views defined on the database D that contains sensitive information. We expect that it is not possible for the user to obtain the extensions of the specific secrecy views by combination of secret answers to conjunctive queries. The user will try to reconstruct the original instance that builds from the SAs to the queries the instance D and at the end checks if there are answers in the secrecy view evaluated in the constructed instance.

The following proposition states that by combination of SAs to queries, there is no way for the user to find out the extensions of the secrecy views evaluated in the original instance D.

Proposition 2. For every V_s of the form (2.12) in $\mathcal{V}^s$, $SA(\mathcal{Q}^{V_s}, D, \mathcal{V}^s) = V_s[D_{\mathcal{V}^s}]$. $\square$

Proof: We know that, $SA(\mathcal{Q}^{V_s}, D, \mathcal{V}^s)$ is empty or in all its tuples only null appears.

1. First, we consider $SA(\mathcal{Q}^{V_s}, D, \mathcal{V}^s) = \varnothing$. This means that there exists at least one secrecy instance D_s, such that $D_s \not\models_N^q \mathcal{Q}^{V_s}[\bar{s}[\bar{x}|\bar{a}]]$[2]. By Definition 15, there exists at least one *restricted relevant variable* $v_j \in \mathcal{V}^R(\mathcal{Q}^{V_s})$, and $\bar{s}(v_j) = null$ in D_s. We can always find one subquery $R_i(\bar{x}_i)$ of $\mathcal{Q}^{V_s}$, such that $v_j \in \bar{x}_i$. Let $\{\bar{t}_1, .., \bar{t}_m\} = SA(R_i(\bar{x}_i), D, \mathcal{V}^s)$. The values in t_i violated V_s and associated with v_j are *null*. So, by Definition 15, we can conclude that, $D_{\mathcal{V}^s} \not\models_N^q \mathcal{Q}^{V_s}[\bar{s}[\bar{x}|\bar{a}]]$. Thus, $V_s[D_{\mathcal{V}^s}]$ is empty.

2. Second, we consider $SA(\mathcal{Q}^{V_s}, D, \mathcal{V}^s)$ contain only null tuples. This means that for each secrecy instance $D_{s_i} \in Sec(D, \mathcal{V}^s)$, it holds $D_{s_i} \models_N^q [\bar{s}[\bar{x}|\overline{null}]]$. For each subqueries $R_i(\bar{x}_i)$ of $\mathcal{Q}^{V_s}$, let $\{\bar{t}_1, .., \bar{t}_m\} = SA(R_i(\bar{x}_i), D, \mathcal{V}^s)$. Since $D_{s_i} \models_N^q [\bar{s}[\bar{x}|\overline{null}]]$, the values in t_i violated V_s and associated with free variables are

[2] $\bar{s}$ is a function from the set of variables to the underlying database domain $\mathcal{U}$, such that for the free variables $(x_1, ..., x_n)$ of $\mathcal{Q}$ it holds $\bar{s}(x_i) = a_i$, with $a_i \in \mathcal{U}$ (cf. Definition 13).

null. Then, we can conclude that $D_{\mathcal{V}^s} \models^q_N Q^{V_s}[\bar{s}[\bar{x}|\overline{null}]]$. Thus, $V_s[D_{\mathcal{V}^s}]$ in all its tuples only null appears. □

Notice that we assume that the original database may be incomplete, in the sense that some information is represented using null values. A null is used to represent *unknown* or *inapplicable* information. In consequence, a user who obtains nulls from a query will not know if the nulls were already there or were (virtually) introduced for privacy purposes. However, if the user knows the definition of the secrecy views, it is possible for him to combine SAs with the definition of the secrecy view to determine the original contents of the view, as shown in the following example.

Example 4.11. Consider the secrecy view $V_s(x) \leftarrow P(x,y), x = 1$, and the database instance $D = \{P(1,1)\}$. The D has the following secrecy instance D_s :

P	X	Y
	null	1

For the query $\mathcal{Q}(x) : \exists y(P(x,y) \wedge x = 1)$, the secrecy answer to $\mathcal{Q}(x)$ on D is $\varnothing$. If the user knows there exists one tuple in the original database, by combining the secrecy view definition, he could imply that $(1) \in V_s[D]$. □

In summary, for our solution to work, we are relying on the following assumptions about an external user:

(a) He interacts with a possibly incomplete database.

(b) The interaction is via query answering.

(c) The user does not know the secrecy view definitions.

Based on these assumptions, by Proposition 2, the user can not obtain information about the specific secrecy views, by combination of SAs to any conjunctive queries. Therefore, there is not leakage of sensitive information to the user.

4.3 Summary

We have introduced a semantics of secrecy that considers the possible occurrence of null values in databases. Under this semantics, we capture a class of secrecy instances that makes the extensions of the secrecy views either become empty or contain tuples showing only null values. Furthermore, the secrecy instances do not depart from the original instance by more than what is needed to protect the secret data. So, they minimally differ in a precise sense from the original instance.

The queries can be posed against all of secrecy instances simultaneously to obtain secret answers. They do not reveal any secret data as defined by the secrecy views. The semantics of secret answers (cf. Definition 32) is based on the notion of the null query answering introduced in [11].

Chapter 5

Secrecy Logic Programs

As indicated in Chapter 1, we expect the updates leading to the secrecy instances to be virtual. Actually, the secrecy instances are more of an auxiliary notion to define the right secrecy semantics. In general, we expect not to have to compute all the secrecy instances, materialize them, and then cautiously query them. 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 $\mathcal{Q}'$. The new query would be posed to D and the usual answers returned by D should be the secret answers to $\mathcal{Q}$. We would like $\mathcal{Q}'$ to be still a simple query, that can be easily evaluated. However, the possibility of being able to do this is restricted by the intrinsic complexity of the problem of computing secret answers, which is likely to be higher than polynomial time in data (cf. Section 7.1). In consequence, $\mathcal{Q}'$ may not be a conjunctive query, actually not even a FO query.

An alternative is to specify the secrecy instances in a compact manner, by means of a logical theory, and do reasoning from that theory. This will not decrease a high intrinsic complexity, but can be much more efficient than computing all the secrecy instances and querying them in turns. Actually, the secrecy instances for a original database can be specified as the stable models of a disjunctive logic program.

The idea is that, given a database instance D, and a set $\mathcal{V}^s$ of secrecy views of the form (2.12), the secrecy program $\Pi(D, \mathcal{V}^s)$ is constructed in such a way that there is a one-to-one correspondence between the stable models of $\Pi(D, \mathcal{V}^s)$ and the secrecy instances of D.

The secrecy program uses annotation constants with the intended, informal semantics shown in the table below. The annotations are used to keep track of virtual updates, i.e. of old and new data:

Annotation	Atom	The tuple $R(\bar{a})$
$\mathbf{b_u}$	$R\_(\bar{a}, \mathbf{b_u})$	has already been updated (old data)
$\mathbf{a_u}$	$R\_(\bar{a}, \mathbf{a_u})$	is updated tuple in database (new data)
$\mathbf{t}$	$R\_(\bar{a}, \mathbf{t})$	is new or old tuple
$\mathbf{s}$	$R\_(\bar{a}, \mathbf{s})$	stays in the final database

Table 5.1: Annotation Constants

In a program we will find each database predicate $R \in \mathcal{R}$ together with its new version $R\_$ of it, that contains an extra argument that is used to place an annotation constant. The idea is to use logic rules to specify and capture how a database violates secrecy, and how the database can become secret with respect to a set $\mathcal{V}^s$ of secrecy views via null-based updates.

Actually, each atom of the form $R(\bar{a})$ will receive one of the constants in Table 5.1. In $R\_(\bar{a}, \mathbf{b_u})$, annotation $\mathbf{b_u}$ means that the atom has already been updated, and $\mathbf{a_u}$ should appear in the new, updated atom. For example, consider a tuple $R(a, b) \in D$. A new tuple $R(a, null)$ is obtained via the update of b to $null$. Therefore, $R\_(a, b, \mathbf{b_u})$ denotes the old atom before updating, while $R\_(a, null, \mathbf{a_u})$ denotes the new atom after the update.

Annotations are performed according to the following sequential steps: First, each ground atom $R(\bar{a})$ from the database becomes a fact in $\Pi(D, \mathcal{V}^s)$. Next, for each secrecy view definition, a disjunctive rule is constructed in such a way that the body of the rule captures the violation condition for the secrecy; and the head describes how to restore secrecy by updating the corresponding tuples. The update is achieved by using annotation $\mathbf{a_u}$.

Example 5.1. As an illustration, consider a schema with relations $P(X,Y)$, $R(Y,Z)$, and the following secrecy view

$$V_s(x,z) \leftarrow P(x,y), R(y,z), y < 3. \tag{5.1}$$

The disjunctive program rule:

$$\begin{aligned}(P\_(null, y, \mathbf{a_u}) \wedge R\_(y, null, \mathbf{a_u})) \vee P\_(x, null, \mathbf{a_u}) \vee R\_(null, z, \mathbf{a_u})\\ \leftarrow P(x,y), R(y,z), y < 3, y \neq null, aux(x,z),\end{aligned} \tag{5.2}$$

states that if the given database is not already a secrecy instance, then secrecy is restored by either updating values in the combination attribute Y with nulls, or values in the secrecy attributes X and Z with nulls, simultaneously.

In general, disjunctive logic programs do not allow for conjunctions in the head. Therefore, the intended head rule of (5.2) generates two rules:

$$P\_(null, y, \mathbf{a_u}) \vee P\_(x, null, \mathbf{a_u}) \vee R\_(null, z, \mathbf{a_u}) \leftarrow Body.^1 \tag{5.3}$$

$$R\_(y, null, \mathbf{a_u}) \vee P\_(x, null, \mathbf{a_u}) \vee R\_(null, z, \mathbf{a_u}) \leftarrow Body. \tag{5.4}$$

Furthermore, we need to restore secrecy only if the given database is not already a secrecy instance, which happens when all the combination attributes are not *null*, the secrecy attributes are not simultaneously *null*, and formula φ is true. For the secrecy view (5.1), the secrecy attributes are X and Z, and the combinability attribute is Y. We use the following auxiliary rules to capture the idea of the secrecy attributes are not simultaneously *null*.

$$aux(x,z) \leftarrow P(x,y), R(y,z), y < 3, x \neq null. \tag{5.5}$$

$$aux(x,z) \leftarrow P(x,y), R(y,z), y < 3, z \neq null. \tag{5.6}$$

[1]The body of rule, denoted by *Body*, remains the same as the one of (5.2).

The annotation constant $\mathbf{t}$ is introduced in order to solve the interaction problem among secrecy views. Thus, it becomes highly significant in cases when the updated instance may still not be secrecy according another secrecy view. For example, $P(x, null, \mathbf{a_u})$ is generated by using Rule (5.3), which is not secrecy with respect to the different secrecy view $V_s(x) \leftarrow P(x, y)$. So, we need to keep updating the instance wrt $V_s(x) \leftarrow P(x, y)$. The aftermath is that the bodies of rules (5.3) and (5.4) have to be modified to:

$$P_{\_}(x, y, \mathbf{t}), R_{\_}(y, z, \mathbf{t}), y < 3, y \neq null, aux(x, z), \tag{5.7}$$

and the program rules (5.5) and (5.6) have to be changed to:

$$aux(x, z) \leftarrow P_{\_}(x, y, \mathbf{t}), R_{\_}(y, z, \mathbf{t}), y < 3, x \neq null, \tag{5.8}$$

$$aux(x, z) \leftarrow P_{\_}(x, y, \mathbf{t}), R_{\_}(y, z, \mathbf{t}), y < 3, z \neq null, \tag{5.9}$$

where the atom $P_{\_}(x, y, \mathbf{t})$ becomes true if either $P(x, y)$ or $P_{\_}(x, y, \mathbf{a_u})$ are true. Similarly for the atom $R_{\_}(y, z, \mathbf{t})$.

Moreover, we need to collect the tuples in the database that have already been updated and (virtually) no longer exist in the database. This is achieved by using the following rules:

$$P_{\_}(x, y, \mathbf{b_u}) \leftarrow P_{\_}(x, y, \mathbf{t}), R_{\_}(y, z, \mathbf{t}), y < 3, y \neq null, aux(x, z), P_{\_}(null, y, \mathbf{a_u}), x \neq null. \tag{5.10}$$

$$R_{\_}(y, z, \mathbf{b_u}) \leftarrow P_{\_}(x, y, \mathbf{t}), R_{\_}(y, z, \mathbf{t}), y < 3, y \neq null, aux(x, z), R_{\_}(y, null, \mathbf{a_u}), z \neq null. \tag{5.11}$$

$$P_{\_}(x, y, \mathbf{b_u}) \leftarrow P_{\_}(x, y, \mathbf{t}), R_{\_}(y, z, \mathbf{t}), y < 3, y \neq null, aux(x, z), P_{\_}(x, null, \mathbf{a_u}). \tag{5.12}$$

$$R_{\_}(y, z, \mathbf{b_u}) \leftarrow P_{\_}(x, y, \mathbf{t}), R_{\_}(y, z, \mathbf{t}), y < 3, y \neq null, aux(x, z), R_{\_}(null, z, \mathbf{a_u}). \tag{5.13}$$

Finally, atoms with annotation constant $\mathbf{s}$ are the ones that become true in the secrecy instances. This constant is used to read off the database atoms in the secrecy instances. So, the secrecy instances are obtained by restrictions of the models of the program to those atoms with the **s** annotation by using following rules:

$$P\_(x, y, \mathbf{s}) \leftarrow P\_(x, y, \mathbf{t}),\ \mathit{not}\ P\_(x, y, \mathbf{b_u}). \tag{5.14}$$

$$R\_(x, y, \mathbf{s}) \leftarrow R\_(x, y, \mathbf{t}),\ \mathit{not}\ R\_(x, y, \mathbf{b_u}). \tag{5.15}$$

□

Before presenting the formal and general definition of logic program, we need some concepts. For an atom of the form $R(\bar{x})$, $\mathit{change}(R(\bar{x}), \bar{y}, t)$, with $\bar{y} = \{y_1, ..., y_n\}$ and $\bar{y} \subseteq \bar{x}$, represents changing each variable $y_i \in \bar{x}$ to t, with t is a variable or domain constant. Let us recall that a secrecy view V_s is of the form:

$$V_s(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n),\ \varphi, \tag{5.16}$$

For each $R_i(\bar{x}_i)$ in the body of the secrecy view, if $(\bar{x}_i \cap \mathcal{C}(V_s)) \neq \varnothing$, let $\mathcal{CP}(V_s) = \{\mathit{change}(R_i(\bar{x}_i), y, \mathit{null})$, for each $y \in (\bar{x}_i \cap \mathcal{C}(V_s))\}$, and if $(\bar{x}_i \cap \mathcal{S}(V_s)) \neq \varnothing$, let $\mathcal{SP}(V_s) = \{\mathit{change}(R_i(\bar{x}_i), \bar{y}, \mathit{null})$, for $\bar{y} = (\bar{x}_i \cap \mathcal{C}(V_s))\}$. The idea behind these two notions is to characterize the LHS of the disjunctive program rule, which is used to restore secrecy.

Example 5.2. Consider the following secrecy view: $V_s(x, z, w) \leftarrow P(x, y), Q(y, z, w)$. By Definition 24, $C(V_s) = \{P[2], Q[1]\}$, and $S(V_s) = \{P[1], Q[2], Q[3]\}$ according to Definition 25. $\mathcal{CP}(V_s) = \{P(x, \mathit{null}), Q(\mathit{null}, z, w)\}$, and $\mathcal{SP}(V_s) = \{P(\mathit{null}, y), Q(y, \mathit{null}, \mathit{null})\}$. □

Definition 34. Given a database instance D, a set $\mathcal{V}^s$ of secrecy views V_ss of the form (2.12), the secrecy program $\Pi(D, \mathcal{V}^s)$ contains the following rules:

1. Facts: $R(\bar{a})$ for each atom $R(\bar{a}) \in D$.

2. For every V_s of the form (2.12), let $\mathcal{SP}(V_s) = \{R^1(\bar{x}_1), \dots, R^a(\bar{x}_a)\}$, and $\mathcal{C}P(V_s) = \{R^1(\bar{x}_1), ..., R^b(\bar{x}_b)\}$. The rules:

   (a) If $\mathcal{S}(V_s) \cap \mathcal{C}(V_s) \neq \varnothing$, the rule:

   $$\bigvee_{R_c \in \mathcal{CP}(V_s)} (R_{c\_}(\bar{x}_c, \mathbf{a_u}) \leftarrow \bigwedge_{i=1}^{n} R_{i\_}(\bar{x}_i, \mathbf{t}) \wedge \varphi \wedge \bigwedge_{c_l \in \mathcal{C}(V_s)} c_l \neq null.$$

   (b) If $\mathcal{S}(V_s) \cap \mathcal{C}(V_s) = \varnothing$, the rule:

   $$R_{sj\_}(\bar{x}_{sj}, \mathbf{a_u}) \vee \bigvee_{R_c \in \mathcal{CP}(V_s)} (R_{c\_}(\bar{x}_c, \mathbf{a_u})$$
   $$\leftarrow \bigwedge_{i=1}^{n} R_{i\_}(\bar{x}_i, \mathbf{t}) \wedge \varphi \wedge \bigwedge_{c_l \in \mathcal{C}(V_s)} c_l \neq null \wedge aux_{V_s}(\bar{s}),$$

   with $R_{sj} \in \mathcal{SP}(V_s)$, and $1 \leq j \leq a$.

   Plus the auxiliary rules:

   If $\mathcal{S}(V_s)$ is $s^1, ..., s^k$, then for $1 \leqslant i \leqslant k$,

   $aux_{V_s}(\bar{\mathbf{s}}) \leftarrow \bigwedge_{i=1}^{n} R_{i\_}(\bar{x}_i, \mathbf{t}) \wedge \varphi \wedge s_i \neq null$, where $\bar{s} = \bigcup s_i$.

3. The collection rules:

   $$R_{sj\_}(\bar{x}_{sj}, \mathbf{b_u}) \leftarrow \bigwedge_{i=1}^{n} R_{i\_}(\bar{x}_i, \mathbf{t}) \wedge \varphi \wedge aux_{V_s}(\bar{s}) \wedge \bigwedge_{c_l \in \mathcal{C}(V_s)} c_l \neq null \wedge R_{sj\_}(\bar{x}_{sj}, \mathbf{a_u}) \wedge$$
   $$\bigwedge_{s_l \in (\mathcal{S}(V_s) \cap \bar{x}_{sj})} s_l \neq null, \text{ for } R_{sj} \in \mathcal{SP}(V_s), \text{ and } 1 \leq j \leq a.$$

   $$R_{ck\_}(\bar{x}_{ck}, \mathbf{b_u}) \leftarrow \bigwedge_{i=1}^{n} R_{i\_}(\bar{x}_i, \mathbf{t}) \wedge \varphi \wedge aux_{V_s}(\bar{s}) \wedge \bigwedge_{c_l \in \mathcal{C}(V_s)} c_l \neq null \wedge$$
   $$R_{ck\_}(\bar{x}_{ck}, \mathbf{a_u}), \text{ for } R_{ck} \in \mathcal{CP}(V_s), \text{ and } 1 \leq k \leq b.$$

4. For each predicate $R \in \mathcal{R}$, the annotation rules:

   $R\_(\bar{x}, \mathbf{t}) \leftarrow R(\bar{x})$. $R\_(\bar{x}, \mathbf{t}) \leftarrow R\_(\bar{x}, \mathbf{a_u})$.

5. For each predicate $R \in \mathcal{R}$, the interpretation rule:

   $R\_(\bar{x}, \mathbf{s}) \leftarrow R\_(\bar{x}, \mathbf{t})$, *not* $R\_(\bar{x}, \mathbf{b_u})$. □

The rules in 1. establish the program facts which are the elements of the database. The rules in 2. are the most important and express how to restore secrecy. The body of the rule (right-hand side) will be true if the database instance is not a secrecy instance, and with the left-hand side captures the intended way of restoring secrecy. 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. capture the atoms that are part of the database or updated atoms in the process of restoring secrecy. Rules in 5. collect the tuples that stay in the final state of the updated database.

Example 5.3. (example 4.7 continued) Consider $D = \{P(1,2),\ R(2,1)\}$ 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. $P(1,2),\ R(2,1).$ (the initial database contents)

2. $P\_(null, y, \mathbf{a_u}) \vee P\_(x, null, \mathbf{a_u}) \vee R\_(null, z, \mathbf{a_u})$

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

$R\_(y, null, \mathbf{a_u}) \vee P\_(x, null, \mathbf{a_u}) \vee R\_(null, z, \mathbf{a_u})$

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

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

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

3. $P\_(x,y,\mathbf{b_u}) \leftarrow P\_(x,y,\mathbf{t}), R\_(y,z,\mathbf{t}),$

$$y < 3, y \neq null, aux(x,z), P\_(null, y, \mathbf{a_u}), x \neq null.$$

$R\_(y,z,\mathbf{b_u}) \leftarrow P\_(x,y,\mathbf{t}), R\_(y,z,\mathbf{t}),$

$$y < 3, y \neq null, aux(x,z), R\_(y, null, \mathbf{a_u}), z \neq null.$$

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

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

4. $P\_(x,y,\mathbf{t}) \leftarrow P(x,y).\ \ P\_(x,y,\mathbf{t}) \leftarrow P\_(x,y,\mathbf{a_u}).$

$R\_(x,y,\mathbf{t}) \leftarrow R(x,y).\ \ R\_(x,y,\mathbf{t}) \leftarrow R\_(x,y,\mathbf{a_u}).$

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

$R\_(x,y,\mathbf{s}) \leftarrow R\_(x,y,\mathbf{t}),\ not\ R\_(x,y,\mathbf{b_u}).$ □

The secrecy instances are in one-to-one correspondence with the stable models of program $\Pi(D, \mathcal{V}^s)$. Given a model, the associated secrecy instance is obtained by collecting the atoms that are annotated with $\mathbf{s}$.

The program can be evaluated, for example, using the *DLV* system that computes the disjunctive stable models semantics. It offers a nice and effective interface to commercial DBMSs [30].

Example 5.4. (example 5.3 continued) The program has three stable models (the facts of the program are omitted for simplicity):

$M_1 = \{P(1,2,\mathbf{t}),\ R(2,1,\mathbf{t}),\ aux(1,1),\ \underline{P(1,2,\mathbf{s})},\ R(2,1,\mathbf{b_u}),\ R(null,1,\mathbf{a_u}),$
$\quad R(null,1,\mathbf{t}),\ \underline{R(null,1,\mathbf{s})}\}.$

$M_2 = \{P(1,2,\mathbf{t}),\ R(2,1,\mathbf{t}),\ aux(1,1),\ P(1,2,\mathbf{b_u}),\ \underline{R(2,1,\mathbf{s})},\ P(1,null,\mathbf{a_u}),$
$\quad P(1,null,\mathbf{t}), \underline{P(1,null,\mathbf{s})}\}.$

$M_3 = \{P(1,2,\mathbf{t}),\ R(2,1,\mathbf{t}),\ aux(1,1),\ P(1,2,\mathbf{b_u}),\ R(2,1,\mathbf{b_u}),\ P(null,2,\mathbf{a_u}),$
$\quad R(2,null,\mathbf{a_u}),\ P(null,2,\mathbf{t}),\ R(2,null,\mathbf{t}),\ aux(1,null),\ aux(null,1),$
$\quad \underline{P(null,2,\mathbf{s})},\ \underline{R(2,null,\mathbf{s})}\}.$

The secrecy instances select the underlined atoms: $D_1 = \{P(1,2),\ R(null,1)\}$, $D_2 = \{P(1,null), R(2,1)\}$, and $D_3 = \{P\ (null,2), R(2,null)\}$. As expected, these are the secrecy instances obtained in Example 4.7. □

If we want to obtain the secret answers to a FO conjunctive query $\mathcal{Q}$ written in the language of $L(\Sigma)$, it is not necessary to explicitly compute all the stable models. Instead, the query can be posed directly to the models of the repair program, i.e. to the secrecy instances, and answered according to the skeptical semantics. This will return the secret answers to the query. Of course, the query has to be transformed as a top-layer program. This transformation is done in the following way:

1. Conjunctive query $\mathcal{Q}$ of the form (2.1) first has to be rewritten as the query $\mathcal{Q}_N$ using the methodology introduced in Definition 17.

2. The new query $\mathcal{Q}_N$ is transformed into a query written as a logic program $\Pi(\mathcal{Q})$, which is a stratified Datalog program [33, 4]. $\Pi(\mathcal{Q})$ contains a predicate *Ans*, to collect the final query answers.

3. Each database predicate $R(\bar{x})$ in $\Pi(\mathcal{Q})$ is replaced by $R(\bar{x}, \mathbf{s})$.

4. The query program $\Pi(\mathcal{Q})$ is combined with the secrecy program $\Pi(D, \mathcal{V}^s)$ into a new program Π.

5. The extension of the answer predicate *Ans* in the intersection of all stable models of Π contains exactly the secret answers. If $\mathcal{Q}$ is a boolean query, *yes* is a secret answer if *Ans* is in the intersection of all the models. Otherwise, the secret answer is *no*.

Example 5.5. (example 5.4 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 the query $\mathcal{Q}_N$ obtained in Example 4.7. This new query can be evaluated against instances with *null* treated as any other constant. Now, query $\mathcal{Q}_N$ can be transformed into a query program $\Pi(\mathcal{Q})$: $Ans(x, z) \leftarrow P(x, y, \mathbf{s}), R(y, z, \mathbf{s}), y < 3, y \neq null$, which has to be evaluated in combination with the program in Example 5.3, under the skeptical semantics. In this evaluation, *null* is treated as an ordinary constant. The stable models of program Π are the stable models of $\Pi(D, \mathcal{V}^s)$ expanded by the answers to the query:

$M_1 = M_1 \cup \varnothing$,

$M_2 = M_2 \cup \varnothing$,

$M_3 = M_3 \cup \{Ans(null, null)\}$.

Since secret answers are those we get from all the possible secrecy instances, the answer to this query $\mathcal{Q}$ is $\emptyset$.

On the other hand, for the boolean query $\mathcal{Q}_2$: $\exists xyz(P(x,y) \wedge R(y,z))$, the query program $\Pi(\mathcal{Q}_2)$ is: $Ans \leftarrow P(x,y,\mathbf{s}), R(y,z,\mathbf{s}), y \neq null$. The stable models of program Π are the stable models of $\Pi(D, \mathcal{V}^s)$ expanded by the answers to the query:

$M_1 = M_1 \cup \{\}$,

$M_2 = M_2 \cup \{\}$,

$M_3 = M_3 \cup \{Ans\}$.

The secret answer to this boolean query $\mathcal{Q}_2$ is *no*. □

We have successfully experimented with the DLV system for computing secret answers with respect to above examples.

5.1 Optimizing Query Evaluation from Secrecy Programs

As indicated above, we compute secret answers by evaluating the combination of the secrecy and query programs. These programs are constructed considering all the database predicates and facts, which contain more information than necessary to obtain secret answers. In fact, in most of cases, only a subset of database predicates and facts are needed to compute answers to a specific query. So bringing the whole database into DLV is inefficient. Therefore, we need to optimize the secrecy programs by generating only those parts that are relevant for answering the query, and importing only the relevant data facts for computing answers.

Our optimization approach starts by capturing the relevant database predicates for a specific query. This is done by analyzing the relationship between predicates in the secrecy view definitions and queries. Intuitively, rules in the secrecy programs that do not involve any relevant predicate are eliminated. So the relevant predicates are used for pruning secrecy programs. As a result, programs are smaller than the original ones. Next, our optimization is to apply the built-ins in the query to the relevant database predicates to retrieve the appropriate data. In this manner, we import only a subset of data rather than retrieving all the data. Our optimization to the evaluation of programs make query processing more efficient. In Chapter 6 we perform experiments to show the effectiveness.

5.1.1 Relevant Predicates

In the area of consistent query answering (CQA), an optimization technique is presented that captures the relevant predicates [13] to compute consistent answers from the repair programs.

Given a query, there might be a set of integrity constraints that are not relevant to the query, i.e. their satisfaction or not does not affect the answers to the query. In order to capture the relevant ICs, the relevant predicates are defined and detected by analyzing the relationship between predicates in queries and ICs. This is done by means of a dependency graph. Basically, a dependency graph for ICs represents the dependency relationship among predicates in ICs, i.e. transitive dependencies between predicates. From the graph, it can easily find the relevant predicates that are needed to answer a query.

In this research we adopted the idea of the dependency graph approach introduced in [13, 14, 15] to capture the relevant predicates for our secrecy programs. The following is definition of the dependency graph for a set of secrecy views.

Definition 35. The *dependency graph*, denoted by $\mathcal{G}(\mathcal{V}^s)$ for a set $\mathcal{V}^s$ of secrecy views V_ss of the form (2.12) is defined as follows: for each database predicate $R \in \mathcal{R}$ appearing in $\mathcal{V}^s$ is a vertex, and there is an edge (R_i, R_j) between R_i and R_j iff there exists a secrecy view $V_s \in \mathcal{V}^s$ such that both R_i and R_j appeals in the body of V_s. $\square$

Example 5.6. Figure 5.1 illustrates the dependency graph $\mathcal{G}(\mathcal{V}^s)$ for the set $\mathcal{V}^s$ of secrecy views: $\{V_s(x) \leftarrow P(x,y), Q(x,y),\ V_s(x) \leftarrow S(x,y),\ V_s(y) \leftarrow S(x,y), R(y)\}$. $\square$

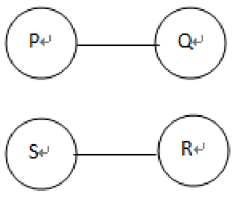

Figure 5.1: Dependency Graph

Definition 36. The set of *relevant predicates* for secret answers to a query $\mathcal{Q}$ with respect to a set $\mathcal{V}^s$ of secrecy views V_ss of the form (2.12), denoted by $Rel(\mathcal{Q}, \mathcal{V}^s)$, contains the following:

(a) For each predicate $R \in \mathcal{Q}$, R is in $Rel(\mathcal{Q}, \mathcal{V}^s)$,

(b) For a predicate R' such that there is a path between (R, R') in the dependency graph $\mathcal{G}(\mathcal{V}^s)$, $R' \in Rel(\mathcal{Q}, \mathcal{V}^s)$. $\square$

Proposition 3. Let $\Pi(D, \mathcal{V}^s, \mathcal{Q})$ be the combination of the secrecy and query programs, and $\Pi(D, \mathcal{V}^s, \mathcal{Q}) \downarrow \mathcal{Q}$ denotes the same as program $\Pi(D, \mathcal{V}^s, \mathcal{Q})$ except that the former contains only rules for each predicate $R \in Rel(\mathcal{Q}, \mathcal{V}^s)$. It holds that $\Pi(D, \mathcal{V}^s, \mathcal{Q}) \downarrow \mathcal{Q}$ and $\Pi(D, \mathcal{V}^s, \mathcal{Q})$ retrieve the same secret answers to query $\mathcal{Q}$. $\square$

We now sketch the proof, which is based on the same basic idea as the proof in [13, Proposition 5.6].

Proof of Proposition 3: Obviously, the program $\Pi(D, \mathcal{V}^s, \mathcal{Q})$ can be split into two logic programs $\Pi_R(D, \mathcal{V}^s, \mathcal{Q})$, and $\Pi_{NR}(D, \mathcal{V}^s, \mathcal{Q})$, respectively. Let $\Pi_R(D, \mathcal{V}^s, \mathcal{Q}) = \Pi(D, \mathcal{V}^s, \mathcal{Q}) \downarrow \mathcal{Q}$. The models of program $\Pi(D, \mathcal{V}^s, \mathcal{Q})$, denoted by $\mathcal{SM}(\Pi(D, \mathcal{V}^s, \mathcal{Q}))$, can be obtained by $\bigcup_M(M \cup \mathcal{SM}(\Pi_{NR}(D, \mathcal{V}^s, \mathcal{Q}))$, for each stable model M of $\Pi_R(D, \mathcal{V}^s, \mathcal{Q})$. Since the program $\Pi_{NR}(D, \mathcal{V}^s, \mathcal{Q})$ does not have rules whose predicates are related with the query predicates, we can not find *Ans* predicate in any of stable models of $\Pi_{NR}(D, \mathcal{V}^s, \mathcal{Q})$. The *Ans* predicate can be obtained only by the program $\Pi_R(D, \mathcal{V}^s, \mathcal{Q})$. Therefore, all the stable models of $\Pi(D, \mathcal{V}^s, \mathcal{Q})$ contain the same *Ans* predicate as $\Pi_R(D, \mathcal{V}^s, \mathcal{Q})$. Thus, $\Pi(D, \mathcal{V}^s, \mathcal{Q}) \downarrow \mathcal{Q}$ and $\Pi(D, \mathcal{V}^s, \mathcal{Q})$ retrieve the same secret answers to query $\mathcal{Q}$. □

Example 5.7. (example 5.6 continued) Consider the query $\mathcal{Q}: Ans(x) \leftarrow S(x, y, \mathbf{s})$, and the database $D = \{S(1,2),\ R(2), P(1,1), Q(1,1)\}$. The set of relevant predicates for the query $\mathcal{Q}$ is $Rel(\mathcal{Q}, \mathcal{V}^s) = \{S, R\}$. Therefore, program $\Pi(D, \mathcal{V}^s, \mathcal{Q}) \downarrow \mathcal{Q}$ contains the following rules:

$S(1,2).\ R(2).$

$S\_(x, y, \mathbf{t}) \leftarrow S\_(x, y, \mathbf{a_u}).\ S\_(x, y, \mathbf{t}) \leftarrow S(x, y).$

$R\_(x, \mathbf{t}) \leftarrow R\_(x, \mathbf{a_u}).\ R\_(x, \mathbf{t}) \leftarrow R(x).$

$S\_(x, null, \mathbf{a_u}) \vee R\_(null, \mathbf{a_u}) \leftarrow S\_(x, y, \mathbf{t}), R\_(y, \mathbf{t}), y \neq null.$

$S\_(x, y, \mathbf{b_u}) \leftarrow S\_(x, y, \mathbf{t}), R\_(y, \mathbf{t}), S\_(x, null, \mathbf{a_u}), y \neq null.$

$R\_(y, \mathbf{b_u}) \leftarrow S\_(x, y, \mathbf{t}), R\_(y, \mathbf{t}), R\_(null, \mathbf{a_u}), y \neq null.$

$S\_(null, y, \mathbf{a_u}) \leftarrow S\_(x, y, \mathbf{t}), auxS(x).$

$auxS(x) \leftarrow S\_(x, y, \mathbf{t}), x \neq null.$

$S\_(x, y, \mathbf{b_u}) \leftarrow S\_(x, y, \mathbf{t}), S\_(null, y, \mathbf{a_u}), auxS(x).$

$S\_(x, y, \mathbf{s}) \leftarrow S\_(x, y, \mathbf{t}), not\ S\_(x, y, \mathbf{b_u}).$

$R\_(x, \mathbf{s}) \leftarrow R\_(x, \mathbf{t}), not\ R\_(x, \mathbf{b_u}).$

$Ans(x) \leftarrow S(x, y, \mathbf{s}).$

The program $\Pi(D, \mathcal{V}^s, \mathcal{Q}) \downarrow \mathcal{Q}$ has two stable models:

$\mathcal{M}_1 = \{S(null, null, \mathbf{s}), R(2, \mathbf{s}), Ans(null)\}.$

$\mathcal{M}_2 = \{S(null, 2, \mathbf{s}), R(1, \mathbf{s}), Ans(null)\}$

Since there is Ans-atoms in common, the secret answer to this query $\mathcal{Q}$ is $(null)$, as expected. Notice that, rules and database facts that do not involve relevant predicates are not generated. Thus, secret answers are obtained by running the smaller secrecy program together with the query program in DLV system. $\square$

5.1.2 Relevant Facts

In the former approach, all the database elements are imported into the logic program as facts for computing secret answers (cf. Rule 1 in Definition 34). This is inefficient, since we only require a subset of database facts. In this section, we show how to obtain the appropriate data, by taking advantage of all the built-in predicates such as $=, >, <$ in the query. The following example will illustrate the idea.

Example 5.8. Consider the database instance $D = \{P(1,5), P(1,3), S(1,2)\}$, a set $\mathcal{V}^s$ of secrecy views: $\{V_s(x) \leftarrow P(x,y), y < 6, V_s(x) \leftarrow S(x,y)\}$, and the conjunctive query $\mathcal{Q}$: $Ans(x) \leftarrow P(x,y), y = 3$ with a built-in predicate $y = 3$. According to Definition 36, the revelent predicate for this query $\mathcal{Q}$ is P. Therefore, we do not need to import any data for the database predicate (or relation) S. Furthermore, only data need to be imported into the secrecy program is $P(1,3)$ instead of $P(1,5)$ and $P(1,3)$. This makes sense, since $P(1,5)$ can never be an answer to the query $\mathcal{Q}$, no matter whether it is defined in secrecy views or not. $\square$

Algorithm 1 brings built-in conditions in the query to the relevant database predicates to retrieve appropriate data facts. This makes sense, since the built-ins in the query would be used to restrict the tuples involved in the computation of the query.

Example 5.9. (example 5.8 continued) The query $\mathcal{Q}$ contains a built-ins $y = 3$. The

Algorithm 1 Relevant Facts Algorithm

```
Require: $\mathcal{Q}$, $Rel(\mathcal{Q},\mathcal{V}^s)$
1: procedure RelFact($\mathcal{Q}$,$Rel(\mathcal{Q},\mathcal{V}^s)$)
2:     Let $\mathcal{BV}(\mathcal{Q})$ be a set of built-in variables in the body of query $\mathcal{Q}$
3:     for each predicate $P_i$ in $Rel(\mathcal{Q},\mathcal{V}^s)$ do
4:         for each variable $v_i$ in $P_i$ do
5:             if $v_i \in \mathcal{BV}(\mathcal{Q})$ then
6:                 Retrieve data for predicate $P_i$ with built-ins
7:             end if
8:         end for
9:     end for
10: end procedure
```

relevant predicate P contains the variable y. We get the following query statement for the relevant predicate P to retrieve relevant data facts:

```
SELECT * DISTINCT
FROM P
WHERE Y=3
```

When evaluating secrecy and query programs in DLV, this query statement can be used to import data facts, residing in database systems into DLV (cf. Section 6.1 for details). □

5.2 Summary

In this chapter we have introduced disjunctive logic programs with stable model semantics to specify the secrecy instances. These programs can be used to compute secret answers.

We also presented a methodology to secrecy programs to optimize the evaluation of programs. Optimization is achieved in two steps. In the first step, the relevant database predicates are captured to remove the rules (and database facts) from the secrecy programs, as shown by Example 5.7. This is done by analyzing the relationship between the database predicates in queries and secrecy views, by meanings of dependency graph in Definition 35. As a result, we obtain reduced secrecy programs, i.e. programs that consider the relevant predicates to compute a query. The second step in our optimization is to apply the built-in conditions from the query to the relevant database predicates to retrieve the appropriate data. In this manner, only the set of data facts for the query is imported into the reasoning system.

The optimization reduces the amount of data involved in the computation of queries. In Chapter 6, we report experimental results that show that optimized programs are more efficient to compute secret answers for queries than the straightforward evaluation of programs.

Chapter 6

Experiments

In this chapter, we present experimental results about the computation of *secret answers* to queries. These queries are representative of the type of queries in our research. The experiments also show query answering with and without the optimizations steps introduced in Section 5.1.

6.1 Preliminaries

Secret answers to queries can be computed by evaluating queries against the secrecy programs. For this purpose we use the DLV system, that implements the stable model semantics of disjunctive logic programs [20, 30]. In this section, we will briefly the use of the DLV^{DB} system to compute *secret answers*. DLV^{DB} [39, 31], an extension of the DLV system, that provides flexible and easy interfaces with external commercial database management systems via ODBC (Open Database Connectivity).

Figure 6.1 illustrates the general architecture used in query evaluation. The data is stored in a relational DBMS, in our case, IBM DB2. The input for query evaluation includes the following: a first-order conjunctive query $\mathcal{Q}$, a database instance D, and a set $\mathcal{V}^s$ of secrecy views. The only output of this architecture is the set of secret answers to the input query $\mathcal{Q}$ with respect to $\mathcal{V}^s$. We will use the example below to illustrate the application of the DLV system to the computation of secret query answers.

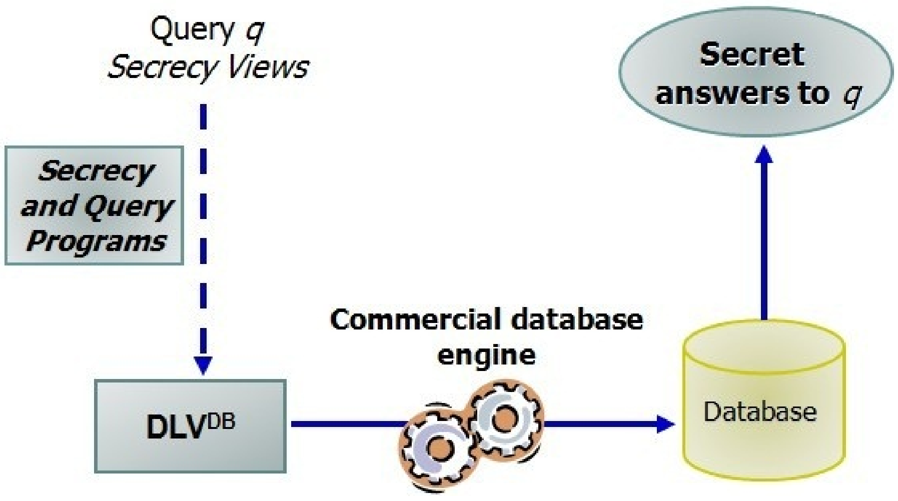

Figure 6.1: Architecture of Query Evaluation

Example 6.1. Consider the database instance $D = \{R(a,b), S(b), R(b,c)\}$, and a secrecy view V_s: $V_s(x) \leftarrow R(x,y), S(y)$. D contains secret information $V_s(a)$ wrt V_s. There are three secrecy instances:

$D_1 = \{R(a, null), R(b,c), S(b)\}$.

$D_2 = \{R(a,b), R(b,c), S(null)\}$.

$D_3 = \{R(null, b), R(b,c), S(b)\}$.

For query $\mathcal{Q} : Ans(x) \leftarrow R(x,y)$, the secret answer is (b). □

First, the secrecy program $\Pi(D, \mathcal{V}^s)$ needs to be constructed wrt the given database instance D and the set $\mathcal{V}^s$ of secrecy views V_s. Here, the facts of the program are not imported from the database directly. Instead, some suitable sentences are included into the secrecy program which can be interpreted by DLV^{DB}.

The secrecy program contains, for each extensional predicate P, the following import sentence:

$$\#import(dbName, dbUser, dbPassword, \text{“}\texttt{SELECT * FROM P}\text{”}, \texttt{P}, typeConv),$$

where $dbName$ is the database name, $dbPasswrod$ is the user password, and $dbUser$

is the user of the database. P defines the name of the predicate that will be used in the program.

Since DLV^{DB} supports only unsigned integer and constant data types, *typeConv* specifies the conversion for mapping DBMS data types to DLV^{DB} data types for each column. The *typeConv* parameter is a string with the following syntax: *type*: *Conv* [, *Conv*], where *type*: is a string constant and *Conv* is one of several conversion types:

- *U_Int*: the column is converted to an unsigned integer;
- *Const*: the column is converted to a string without quotes;
- *Q_Const*: the column is converted to a string with quotes.

The number of the entries in the conversion list has to match the number of columns in the selected table.

Example 6.2. Assume the following table definition:

CREATE TABLE p (X varchar(10))

We insert two tuples in the table p using the following SQL statement:

INSERT INTO p

VALUES ('a'), ('b')

Let us now use import the table p into the DLV^{DB} syetem. Let the file *importonly.dlv* contains just a single *#import* statement:

$$\#import(test, \text{``}ab2admin\text{''}, \text{``}test\text{''}, \text{``}\texttt{SELECT * FROM p}\text{''}, \texttt{P}, type : Q\_Const).$$

Invoking DLV^{DB} with this file yields:

$$dl\ -silent\ importonly.dlv$$

$$\{P(\text{``}a\text{''}), P(\text{``}b\text{''})\}$$

The varchar values converted as strings with quotes, which are constants in the DLV^{DB} system. $\square$

As a result, the database facts stored in the database *dbName* will be imported into the reasoning system DLV^{DB}, where the secrecy program is run. Now we describe how DLV^{DB} interacts with DBMS.

Program 1 is the secrecy program for the database instance D and the secrecy view in Example 6.1. We assume the following database parameters:

- Database name: test
- Database user: db2admin
- Database password: test

Program 1.

$\#import(test, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{“}\texttt{SELECT * FROM R}\text{”},$

$\qquad\qquad R1,\ type : Q\_Const, Q\_Const).$

$\#import(test, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{“}\texttt{SELECT * FROM S}\text{”}, \texttt{S1},\ type : Q\_Const).$

$R\_(x, y, \mathbf{t_d}) :- R1(x, y). \quad S\_(y, \mathbf{t_d}) :- S1(y).$

$R\_(\text{“}null\text{”}, y, \mathbf{a_u}) \vee R\_(x, \text{“}null\text{”}, \mathbf{a_u}) \vee S\_(\text{“}null\text{”}, \mathbf{a_u})$

$\qquad\qquad :- R\_(x, y, \mathbf{t}), S\_(y, \mathbf{t}), y \neq \text{“}null\text{”}, aux(x).$

$aux(x) :- R\_(x, y, \mathbf{t}), S\_(y, \mathbf{t}), y \neq \text{“}null\text{”}, x \neq \text{“}null\text{”}.$

$R\_(x, y, \mathbf{b_u}) :- R\_(x, y, \mathbf{t}), S\_(y, \mathbf{t}), R\_(\text{“}null\text{”}, y, \mathbf{a_u}), y \neq \text{“}null\text{”}, aux(x), x \neq \text{“}null\text{”}.$

$R\_(x, y, \mathbf{b_u}) :- R\_(x, y, \mathbf{t}), S\_(y, \mathbf{t}), R\_(x, \text{“}null\text{”}, \mathbf{a_u}), y \neq \text{“}null\text{”}, aux(x).$

$S\_(y, \mathbf{b_u}) :- R\_(x, y, \mathbf{t}), S\_(y, \mathbf{t}), S\_(\text{“}null\text{”}, \mathbf{a_u}), y \neq \text{“}null\text{”}, aux(x).$

$R\_(x, y, \mathbf{t}) :- R\_(x, y, \mathbf{t_d}). \quad R\_(x, y, \mathbf{t}) :- R\_(x, y, \mathbf{a_u}).$

$S\_(x, \mathbf{t}) :- S\_(y, \mathbf{t_d}). \quad S\_(y, \mathbf{t}) :- S\_(y, \mathbf{a_u}).$

$R\_(x, y, \mathbf{s}) :- R\_(x, y, \mathbf{t}),\ not\ R\_(x, y, \mathbf{b_u}).$

$S\_(y, \mathbf{s}) :- S\_(y, \mathbf{t}),\ not\ S\_(y, \mathbf{b_u}).$ □

Secrecy programs should be stored for other queries. This is because the secrecy programs just depend on the given set of secrecy views and database relations, which are independent from the particular query. Therefore, they can be used for any other conjunctive query to compute secret answers. In fact, secrecy programs need to be rewritten only when the database schema or secrecy views are changed.

After having the secrecy program for the given database D, the secret answers to the query $\mathcal{Q}$ posed to D can be computed by running the query program in combination with the secrecy program. The query program for the query $\mathcal{Q}$ in Example 6.1 is the following:

Program 2. The query program for the query $\mathcal{Q}$.

$Ans(X) :- R\_(X, Y, \mathbf{s}).$
$Ans(X)?$ □

As a reminder, we need to emphasize here that the query to be run has to be transformed into one that treats the nulls as any other constant.

Finally, the combined program, stored in a text file named ex.dlv, is run in DLV^{DB} under skeptical (cautious) reasoning to compute secret answers as follows:

$$dl.exe - silent - cautious \quad ex.dlv$$

$$\text{``}b\text{''}$$

For Program 1, DLV^{DB} returns (b) as the secret answer to user, as we expected. Of course, we could use the optimization steps presented in this research to improve efficiency. The next section, we will look at the execution time on computation of secret answers to queries with and without optimization.

6.2 Experimental Setup

Serval experiments on computation of secret answers to queries were run on DLV system, performed by a HP PC with Inter(R) Core(TM)2 CPU and 2GB RAM using Windows 7 operating system. The database instance was stored on the IBM DB2 Professional Database Server Edition V9.7. All the programs were run in the version of DLV^{DB} for Windows released on September 20th, 2007.

For the experiments we consider the database instance D, with 300 tuples for an company database with following relations and the set of secrecy views:

- $\mathit{Employee}(\mathit{ENO}, \mathit{ENAME}, \mathit{SALARY}, \mathit{DNO})$. It stores the employee number, name, salary, and department number of employees.
- $\mathit{Department}(\mathit{DNO}, \mathit{DNAME})$. It stores the department number, and name of departments.
- $\mathit{Project}(\mathit{PNO}, \mathit{PNAME}, \mathit{AID})$. It stores the project's numbers, names, and addresses.
- $\mathit{Address}(\mathit{AID}, \mathit{CITY}, \mathit{STATE}, \mathit{COUNTRY})$. It stores address information wrt city, state and country.

The set $\mathcal{V}^s$ of secrecy views is given by:

$$V_s(dno, dname) \leftarrow Department(dno, dname), dname = \text{``}operations\text{''}. \tag{6.1}$$

$$V_s(eno, ename) \leftarrow Employee(eno, ename, salary, dno), salary < 30,000. \tag{6.2}$$

$$\begin{aligned} V_s(pno, pname) \leftarrow & Project(pno, pname, aid), \\ & Address(aid, city, state, country), city = \text{``}ott\text{''}. \end{aligned} \tag{6.3}$$

Assuming the database name *company*, the database user *ad2admin*, and the password *test*, the secrecy program for the database instance D contains the rules in Program 3.

Program 3.

$$
\begin{aligned}
&\#import(company, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{“}\texttt{SELECT * FROM Employee}\text{”}, Emp,\\
&\qquad type : Q\_Const, Q\_Const,\\
&\qquad Q\_Const, Q\_Const).\\
&\#import(company, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{“}\texttt{SELECT * FROM Department}\text{”}, Dept,\\
&\qquad type : Q\_Const, Q\_Const).\\
&\#import(company, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{“}\texttt{SELECT * FROM Project}\text{”}, Proj,\\
&\qquad type : Q\_Const, Q\_Const, Q\_Const).\\
&\#import(company, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{“}\texttt{SELECT * FROM Address}\text{”}, Addr,\\
&\qquad type : Q\_Const, Q\_Const,\\
&\qquad Q\_Const, Q\_Const).
\end{aligned}
$$

$$
\begin{aligned}
Department\_(dno, dame, \mathbf{t_d}) &:- Dept(dno, dname).\\
Department\_(dno, \text{“}null\text{”}, \mathbf{a_u}) &:- Department\_(dno, \text{“}Operations\text{”}, \mathbf{t}).\\
Department\_(dno, \text{“}Operations\text{”}, \mathbf{b_u}) &:- Department\_(dno, \text{“}Operations\text{”}, \mathbf{t}),\\
&\quad Department\_(dno, \text{“}null\text{”}, \mathbf{a_u}).\\
Department\_(dno, dname, \mathbf{t}) &:- Department\_(dno, dname, \mathbf{t_d}).\\
Department\_(dno, dname, \mathbf{t}) &:- Department\_(dno, dname, \mathbf{a_u}).\\
Department\_(dno, dname, \mathbf{s}) &:- Department\_(dno, dname, \mathbf{t}),\\
not\ &\ Department\_(dno, dname, \mathbf{b_u}).\\
Employee\_(eno, ename, salary, dno, \mathbf{t_d}) &:- Emp(eno, ename, salary, dno).\\
Employee\_(\text{“}null\text{”}, \text{“}null\text{”}, salary, dno, \mathbf{a_u}) &\vee Employee\_(eno, ename, \text{“}null\text{”}, dno, \mathbf{a_u})\\
&:- Employee\_(eno, ename, salary, dno, \mathbf{t}),\\
&\quad salary < \text{“}30000\text{”}, salary \neq \text{“}null\text{”},\\
&\quad auxEmp(eno, ename).\\
auxEmp(eno, ename) &:- Employee\_(eno, ename, salary, dno, \mathbf{t}),\\
&\quad salary < \text{“}30000\text{”}, salary \neq \text{“}null\text{”},\\
&\quad eno \neq \text{“}null\text{”}.
\end{aligned}
$$

$$\begin{aligned}
auxEmp(eno, ename) :- & Employee\_(eno, ename, salary, dno, \mathbf{t}),\\
& salary < \text{“30000”}, salary \neq \text{“}null\text{”},\\
& ename \neq \text{“}null\text{”}.\\
Employee\_(eno, ename, salary, dno, \mathbf{b_u}) :- & Employee\_(eno, ename, salary, dno, \mathbf{t}),\\
& salary < \text{“30000”}, salary \neq \text{“}null\text{”},\\
& auxEmp(eno, ename),\\
& Employee\_(eno, ename, \text{“}null\text{”}, dno, \mathbf{a_u}).\\
Employee\_(eno, ename, salary, dno, \mathbf{b_u}) :- & Employee\_(eno, ename, salary, dno, \mathbf{t}),\\
& salary < \text{“30000”}, salary \neq \text{“}null\text{”},\\
& auxEmp(eno, ename),\\
& Employee\_(\text{“}null\text{”}, \text{“}null\text{”}, salary, dno, \mathbf{a_u}),\\
& eno \neq \text{“}null\text{”}, ename \neq \text{“}null\text{”}.\\
Employee\_(eno, ename, salary, dno, \mathbf{t}) :- & Employee\_(eno, ename, salary, dno, \mathbf{t_d}).\\
Employee\_(eno, ename, salary, dno, \mathbf{t}) :- & Employee\_(eno, ename, salary, dno, \mathbf{a_u}).\\
Employee\_(eno, ename, salary, dno, \mathbf{s}) :- & Employee\_(eno, ename, salary, dno, \mathbf{t}),\\
not\ & Employee\_(eno, ename, salary, dno, \mathbf{b_u}).\\
Project\_(pno, pname, aid, \mathbf{t_d}) :- & Proj(pno, pname, aid).\\
Address\_(aid, city, state, country, \mathbf{t_d}) :- & Addr(aid, city, state, country).\\
Project\_(\text{“}null\text{”}, \text{“}null\text{”}, aid, \mathbf{a_u}) \lor & Project\_(pno, pname, \text{“}null\text{”}, \mathbf{a_u}) \lor\\
Address\_(\text{“}null\text{”}, \text{“}ott\text{”}, state, country, \mathbf{a_u}) \lor & Address\_(aid, \text{“}null\text{”}, state, country, \mathbf{a_u})\\
:- & Project\_(pno, pname, aid, \mathbf{t}),\\
& Address\_(aid, \text{“}ott\text{”}, state, country, \mathbf{t}),\\
& aid \neq \text{“}null\text{”}, auxProj(pno, pname).\\
auxProj(pno, pname) :- & Project\_(pno, pname, aid, \mathbf{t}),\\
& Address\_(aid, \text{“}ott\text{”}, state, country, \mathbf{t}),\\
& aid \neq \text{“}null\text{”}, pno \neq \text{“}null\text{”}.\\
auxProj(pno, pname) :- & Project\_(pno, pname, aid, \mathbf{t}),
\end{aligned}$$

$$\begin{aligned}
& \quad Address\_(aid, \text{"}ott\text{"}, state, country, \mathbf{t}),\\
& \quad aid \neq \text{"}null\text{"}, pname \neq \text{"}null\text{"}.\\
Project(pno, pname, aid, \mathbf{b_u}) & :- Project\_(pno, pname, aid, \mathbf{t}),\\
& \quad Address\_(aid, \text{"}ott\text{"}, state, country, \mathbf{t}),\\
& \quad aid \neq \text{"}null\text{"}, auxProj(pno, pname).\\
& \quad Project\_(pno, pname, \text{"}null\text{"}, \mathbf{a_u})\\
Project\_(pno, pname, aid, \mathbf{b_u}) & :- Project\_(pno, pname, aid, \mathbf{t}),\\
& \quad Address\_(aid, \text{"}ott\text{"}, state, country, \mathbf{t}),\\
& \quad aid \neq \text{"}null\text{"}, auxProj(pno, pname),\\
& \quad Project\_(\text{"}null\text{"}, \text{"}null\text{"}, aid, \mathbf{a_u}),\\
& \quad pno \neq \text{"}null\text{"}, pname \neq \text{"}null\text{"}.\\
Address\_(aid, \text{"}ott\text{"}, state, country, \mathbf{b_u}) & :- Project(pno, pname, aid, \mathbf{t}),\\
& \quad Address(aid, \text{"}ott\text{"}, state, country, \mathbf{t}),\\
& \quad aid \neq \text{"}null\text{"}, auxProj(pno, pname),\\
& \quad Address(\text{"}null\text{"}, \text{"}ott\text{"}, state, country, \mathbf{a_u}).\\
Address\_(aid, \text{"}ott\text{"}, state, country, \mathbf{b_u}) & :- Project\_(pno, pname, aid, \mathbf{t}),\\
& \quad Address\_(aid, \text{"}ott\text{"}, state, country, \mathbf{t})\\
& \quad aid \neq \text{"}null\text{"}, auxProj(pno, pname),\\
& \quad Address\_(aid, \text{"}null\text{"}, state, country, \mathbf{a_u}).\\
Project\_(pno, pname, aid, \mathbf{t}) & :- Project\_(pno, pname, aid, \mathbf{t_d}).\\
Project\_(pno, pname, aid, \mathbf{t}) & :- Project\_(pno, pname, aid, \mathbf{a_u}).\\
Address\_(aid, city, state, country, \mathbf{t}) & :- Addrress\_(aid, city, state, country, \mathbf{t_d}).\\
Address\_(aid, city, state, country, \mathbf{t}) & :- Addrress\_(aid, city, state, country, \mathbf{a_u}).\\
Project\_(pno, pname, aid, \mathbf{s}) & :- Project\_(pno, pname, aid, \mathbf{t}),\\
& \quad not\ Project\_(pno, pname, aid, \mathbf{b_u}).\\
Address\_(aid, city, state, country, \mathbf{s}) & :- Addrress\_(aid, city, state, country, \mathbf{t}),\\
& \quad not\ Addrress\_(aid, city, state, country, \mathbf{b_u}).
\end{aligned}$$

Notice that when Program 3 is evaluated in DLV, the whole company database stored in the DBMS will be imported into DLV, which is inefficient. Therefore, we need to reduce as much as possible the interaction between the DBMS, where the data resides, and the DLV system, where the secrecy program is evaluated. As indicated above, our optimized methodology can generate programs for queries that will import only a subset of the database. We illustrate effectiveness by running three conjunctive queries that are representative of the type of queries in this research.

6.2.1 Test Case1

A conjunctive query $\mathcal{Q}_1$ that asks for employee's number and name if his or her salary is less than 40000:

$$Ans(eno, name) \leftarrow Employee(eno, name, salary, deptno), salay < 40000. \tag{6.4}$$

This query is open, i.e. with free variables and built-ins.

Program 4 is the query program for query $\mathcal{Q}_1$. The secrecy program listed in Program 3 together with this query program can be evaluated using the DLV system to obtain secret answers. Of course, this is a naive method for computing answers, which imports all the data from the database and using the rules for all secrecy view definitions.

Program 4.

$$\begin{aligned} Ans(eno, ename) :- & Employee(eno, ename, salary, dno, \mathbf{s}), \\ & salary \neq \text{“}null\text{”}, salary < 40000. \end{aligned} \qquad \square$$

For the query $\mathcal{Q}_1$, the programs can be optimized by using the relevant predicates and facts (cf. Section 5.1). Here, the relevant predicate for query $\mathcal{Q}_1$ wrt the set $\mathcal{V}^s$ of secrecy views is *Employee*. According to Algorithm 1, the built-in predicate $salary < 40000$ in query $\mathcal{Q}_1$ can be applied to the relevant predicate *Employee*. The following is the import sentence in the optimized program which brings only relevant data facts and predicates into DLV.

$$\#import(company, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{“}\mathtt{SELECT\ *\ FROM\ Employee}\\ \mathtt{WHERE\ Salary < 40000}\text{”}, Emp,\\ type: Q\_Const, Q\_Const, Q\_Const, Q\_Const). \tag{6.5}$$

Program 7 in Appendix A lists all the rules of the optimized program for the query $\mathcal{Q}_1$.

6.2.2 Test Case2

The second query $\mathcal{Q}_2$ asks for employee's name, who works in “sales” department.

$$Ans(name) \leftarrow Employee(eno, name, salary, dno),\\ Department(dno, dname), dname = \text{“}sales\text{”}. \tag{6.6}$$

This conjunctive query is open with joins and built-ins. This type of queries is widely used in practice. The query program for $\mathcal{Q}_2$ shows as follows:

Program 5.

$$\begin{aligned} Ans(ename) :- & Employee(eno, ename, salary, dno, \mathbf{s}),\\ & Department(dno, dname, \mathbf{s}), dname = \text{“}sales\text{”},\\ & dname \neq \text{“}null\text{”}, dno \neq \text{“}null\text{”}. \end{aligned} \quad \Box$$

This query program plus Program 3 can compute secret answers for the conjunctive query $\mathcal{Q}_2$. Program 8 in Appendix A is the optimized program for the secrecy program in 3, and the query program in 5. Notice that in Program 8 only the relevant portion of the database is imported into DLV. This is, only tuples for relations *Employee*, and *Department* are imported into DLV. The following are two import sentences will be used in the optimized program:

$$\#import(company, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{“}\texttt{SELECT * FROM Employee}\text{”}, \\ type: Q\_Const, Q\_Const, Q\_Const, Q\_Const). \tag{6.7}$$

$$\#import(company, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{“}\texttt{SELECT * FROM Department} \\ \texttt{WHERE dname} = \text{‘}\texttt{sales}\text{'”}, Emp, type: Q\_Const, Q\_Const). \tag{6.8}$$

6.2.3 Test Case3

Consider the following boolean query $\mathcal{Q}_3$:

$$Ans \leftarrow Department(dno, dname), dno = \text{“}101\text{”}, dname = \text{“}sales\text{”}. \tag{6.9}$$

The query program to compute secret answers for $\mathcal{Q}_3$ shows as follows:

Program 6.

$Ans : -Department(dno, dname, \mathtt{s}), dno = \text{“}101\text{”}, dname = \text{“}sales\text{”},$
$dno \neq \text{“}null\text{”}, dname \neq \text{“}null\text{”}.$ □

The optimized program in 9 only imports one tuple $Department(101, sales)$ into DLV for this boolean query. The import sentences in the Program 9 is:

$$\#import(company, \text{“}db2admin\text{”}, \text{“}test\text{”}, \text{”}\texttt{SELECT * FROM Department} \\ \texttt{WHERE Name} = \text{‘}\texttt{sales}\text{'}\ \texttt{AND Depo} = \text{‘}101\text{'”}, \\ Dept, type: Q\_Const, Q\_Const). \tag{6.10}$$

6.3 Results

Figure 6.2, 6.3 and 6.4 illustrate the execution times of the three queries on the database instance D.

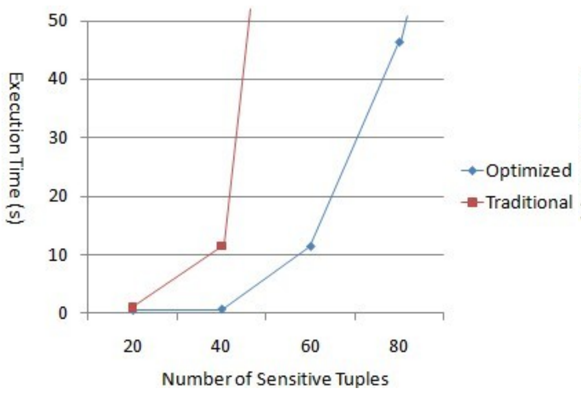

Figure 6.2: Execution Time for $\mathcal{Q}_1$

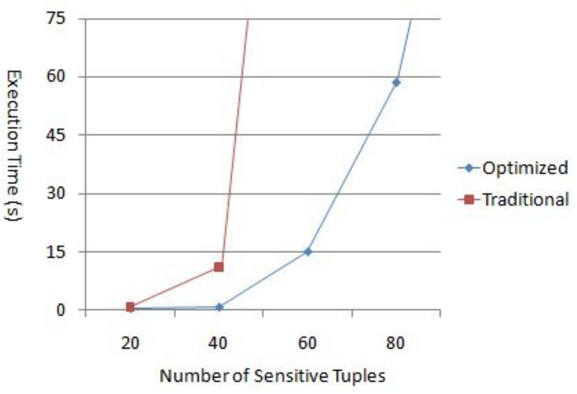

Figure 6.3: Execution Time for $\mathcal{Q}_2$

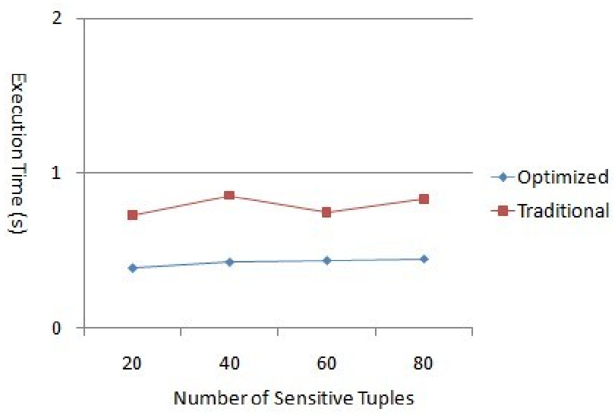

Figure 6.4: Execution Time for $\mathcal{Q}_3$

We can see that, the optimized programs are faster to compute secret answers to queries than the straightforward evaluation of program. This is because they capture the relevant predicates to compute query. As a result, only the rules, and database facts, that are relevant to compute a query are kept in them. Therefore, the data flow between the DLV system and the database is reduced.

Based on the experiments performed, we conclude that the computation of secret answers based on logic programs is viable, especially for the a ground (or partially-ground) query. This makes sense, built-ins in queries would be used to restrict the tuples involved in the computation of the answers.

Optimizations on the process of retrieving answers have been widely studied in the literature. The magic sets (MS) method [21, 6, 34] is one of the best known techniques for optimizing query answering over logic programs. The idea behind magic sets is to determine sets of facts that are relevant to the query, and then use these facts to reduce the amount of data used in query evaluation. In [13, 14], the MS technique for repair programs for consistent query answering in DLV is applied. In [13], experiments on the computation of consistent answers to queries using MS technique are presented. From those experiments, it is possible to conclude that MS technique has an excellent performance on query evaluation. It is a future work to apply MS technique to our secrecy program to further improve efficiency.

Chapter 7

Discussion

7.1 Connection with CQA

Consider a database instance D that fails to satisfy a given set of integrity constraints IC. It still contains useful and correct information. The area of *consistent query answering* (CQA) [8] 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 (2.12) can be seen as an integrity constraint expressed in the FO language $L(\Sigma \cup \{V_s\})$:

$$\forall \bar{x}(V_s(\bar{x}) \longleftrightarrow \exists \bar{y}(R_1(\bar{x}_1) \wedge \cdots \wedge R_n(\bar{x}_n) \wedge \varphi)), \tag{7.1}$$

with $\bar{y} = (\cup \bar{x}_i) \smallsetminus \bar{x}$. From this perspective, maintaining the view defined by (7.1) (i.e. synchronized with the base relations) [23] becomes a problem of *database maintenance*, i.e. maintenance of the consistency of the database wrt (7.1) 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 to maintain the view defined in (7.1). 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 (7.1): If $\bar{x}$ is $x^1, \ldots, x^k$, then for $1 \leq i \leq k$,

$$\forall \bar{x}\bar{y} \neg (R_1(\bar{x}_1) \wedge \cdots \wedge R_n(\bar{x}_n) \wedge \varphi \wedge x^i \neq \mathit{null}). \tag{7.2}$$

That is, from each view definition (7.1) we obtain k *denial constraints*, i.e. prohibited conjunctions of (positive) database atoms and built-ins, which have been investigated in CQA [16, 8]. In consequence, the secrecy instances correspond to the repairs of D wrt the set IC of all the ICs of the form (7.2), obtained from each of the secrecy view definitions. These repairs are defined according to the null-based semantics introduced in Section 4.1, i.e. $\leq_D$-minimality, which was investigated in [12, 11]. This reduction allows us to profit from existing results for CQA.

Example 7.1. The secrecy view defined by $V_s(x, z) \leftarrow P(x, y), R(y, z)$ gives rise to the following denial constraints:

$$\neg \exists xyz (P(x, y) \wedge R(y, z) \wedge x \neq \mathit{null}),$$
$$\neg \exists xyz (P(x, y) \wedge R(y, z) \wedge z \neq \mathit{null}),$$

The initial instance D has to be minimally repaired in order to satisfy them. □

7.2 Connection with Data Cleaning

A database instance may contain several tuples in it that refer to the same external entity that is being modeled by the database. This problem could be caused by errors in data, by data coming from different sources that use different formats, etc.

In this sense, the database is considered to contain duplicate data. These duplicates could compromise secrecy, as illustrated in the following example.

Example 7.2. Consider a database D that includes the following relation *Student*, with attributes *Name, ID, DOB, Gender,* and *AvgMark* (we use an extra column to denote the tuple):

Table 7.1: Student Table

Name	*ID*	*DOB*	*Gender*	*AvgMark*	
J. Smith	001	880801	*M*	60	*t*1
John Smith	001	880801	*M*	60	*t*2

Assume the following secrecy view defined on the relation *Student* to protect *John Smith*'s average mark:

$$V_s(v) \leftarrow Studnet(x, y, z, u, v), x = \text{`}John\ Smith\text{'}. \tag{7.3}$$

According to the secrecy semantics proposed in Chapter 4, we obtain the following two secrecy instances:

Name	*ID*	*DOB*	*Gender*	*AvgMark*
J. Smith	001	880801	*M*	60
John Smith	001	880801	*M*	*null*

Name	*ID*	*DOB*	*Gender*	*AvgMark*
J. Smith	001	880801	*M*	60
null	001	880801	*M*	60

Suppose the user issues the following query:

$$Ans(x, y, z, u, v) \leftarrow Student(x, y, z, u, v). \tag{7.4}$$

Query (7.4) is evaluated against the above set of secrecy instances, and it will get tuples $t_1 = Student(J.\ Smith, 001, 880801, M, null)$, and $t_3 = Student(null$, 001, 880801,

$M, null)$ as secret answers. Actually, two tuples t_1, t_3 refer to the same student. Even though the name of t_1 is not identical to the name of t_3, the same student id in the two tuples is sufficient to establish that the tuples refer to the same student. Thus, t_1 and t_3 are considered as duplicates. The user can take advantage of duplicates to infer sensitive information. In our case, he can obtain *John Smith's* average mark 60 from t_1, which is sensitive information. □

The above example shows that even if the data privacy implementation ensures that the query result contains only authorized information, it is possible for users to gain access to sensitive information based on duplicates, which brings the issue of duplicate detection, and data cleaning [36].

The *Suspect Duplicate Processing* (SDP) feature in IBM InfoSphere Master Data Management Server (MDM) provides a practical method of duplicate detection. We could make use of this practical method to remove duplicates. Before presenting the ideas, we give a brief introduction to MDM.

Master data [18] is the most valuable data that an organization owns, which represents core information about the business, such as customers, products, and accounts etc. Since it is highly valuable, all parts of an organization must agree on it. In an ideal world, we expect a single place where all master data in an organization is stored and managed. All updates should take place against this single copy of master data, and all the users of master data should also interact with this single copy. In this way, the organization can use the information in a consistent way.

In fact, the goal of MDM is to enable this ideal state. It provides a consistent way of using master data entities. In particular, it has the SDP feature which provides a mechanism to keep a single copy of customer data. SDP involves two steps to examining data to find duplicates in the relation. The first step is to select the match criteria and weights, and the second one is matching.

In the rest of this section, we will illustrate how to SDP find duplicate tuples in *Student* relation defined in Table 7.1. Although our illustration uses a relation about students, in general this approach is applicable to any relational database.

Not every data element is important from the suspect duplicate processing point of view. Therefore, data administrators of MDM need to identify the set of date elements that best help to uniquely identify the record in the relation according business requirements. This set of data elements is known as *critical data elements*, which will be used to determine a match.

Example 7.3. (example 7.2 continued) Consider the relation *Student*. The same student id in the two tuples is sufficient to establish that the tuples refer to the same student. In addition, if for two tuples the values of name, date of birth, and gender are the same, then two tuples may be duplicates. So, a set of *critical data elements* for the *Student* includes: *ID*, *Name*, *DOB*, and *Gender*.

After the *critical data elements* have been selected, weights need to be assigned to each critical element. This makes sense, since *ID* should have a higher weight for matching than *DOB*. Assignment of weights is based on expert domain knowledge of the MDM data administrators. An example of a weight assignment for each critical data element in the *Student* relation is shown in the following table:

Critical Data Element	Weight
Gender	1
Data of Birth	2
Name	4
ID	8

Based on the weight assignment, MDM DBA needs to define the *Match Relevancy* table shown in Table 7.2, in which defines a combination of critical data elements that match, and shows what the score is for such a combination.

Basically, match relevancy describes the critical data elements between two tuples

Table 7.2: Match Relevancy

Match_Relev	Name	Score
1	All Elements Matched	15
2	ID,Name,DOB	14
3	ID,Name,Gender	13
4	ID,DOB,Gender	11
...	...	...

that match, and the corresponding score is used to measure the quality of the match. The score is calculated by sum of weights of critical data elements that match. For the two tuples in the *Student* table, values of *ID, DOB*, and *Gender* match. The corresponding match relevancy score is 11 calculated by adding up the weights of *ID, DOB*, and *Gender*, as indicated in the fourth row in Table 7.2.

Similarity, MDM DBA also needs to define the *Non-Match Relevancy* table shown in Table 7.3, which follows the same pattern as the *Match Relevancy* table.

Table 7.3: Non-Match Relevancy

None_Match_Relev	Name	Score
1	All Elements Matched	0
2	ID,Name,DOB	1
3	ID,Name,Gender	2
4	ID,DOB,Gender	4
...	...	...

Non-Match relevancy describes the critical elements between two tuples that do not match, and the corresponding score is calculated simply by adding up the weights for the critical data elements that don't match. For the tuples in the *Student* table, since the values of *Name* do not match (values of *ID, DOB* and *Gender* match), the non-match relevancy score is 4, as indicated in the fourth row of Table 7.3.

Finally, MDM DBA should define the *Match Matrix* table shown in Table 7.4, which brings match/non-match relevancy scores for two tuples to determine duplicate.

Table 7.4: Match Matrix

Match_Relev	None_Match_Relev	Category
1	1	Duplicate
2	2	Duplicate
3	3	Duplicate
4	4	Duplicate
...	...	...

For example, the fourth row in Table 7.4 indicates that match relevancy 4 (*ID, DOB*, and *Gender* match), and non-match relevancy 4 (*Name* dose not match) lead to duplicate. Our tuples in relation *Student* satisfy the fourth row of the *Match Matrix* table. Therefore, we can conclude that those two tuples are duplicates, and we need to delete one tuple to eliminate duplicate. □

7.3 Related Work

Several models for data privacy and access control have been proposed in the recent past [3, 29, 37, 44, 9, 10], which aims to grant a particular user or group access to individual data items in a relation. In contrast, although the current SQL supports access control at the level of tables or columns, it does not provide any way to specify authorization to control which tuples can be accessed by which users.

Virtual Private Database (VPD) feature of Oracle's 9*i* [3] supports fine-grained access control by transparent query modification. The authorization policy is encoded into PL/SQL functions defined for each relation, which are used to return *where* clause predicates to be appended to the user query before it is executed. The added predicates ensure that the user gets to see only those tuples in each table that he or she is authorized to see. As noted in [19, 35], because of the lack of a formal mathematical basis for VPD, it is very difficult to define policy functions corresponding to business requirements, especially authorization policies involving joins of tables. Our framework provides a simple language that enables us to specify authorization policy.

Cell-level access control is described by LeFevre *et al.* [29]. Their work focuses on ensuring limited data disclosure, based on the premise that data owners have control over who is allowed to see their personal data and for what purpose. In their work, they introduce two models of cell-level limited disclosure enforcement: *table semantics* and *query semantics*. The implementation of such models is based on query modification technique.

Rizvi *et al.* [37] present two basic approaches to enforce fine-grained access control. The first approach is the *Truman model* and the second one is the *Non-Truman model*. In the Truman model, the access control is defined by a collection of authorization views for each relation in the database. Each user has associated a set of authorization views. When a user issues a query, the query is modified transparently by replacing each relation in the query by the corresponding authorization view associated with the user. As Rizvi *et al.* [37] point out, using a Truman model, the answers to queries may be misleading, or worse, incorrect.

Non-Truman models, by contrast, prevent misleading answers. Under the Non-Truman model, a query that violates access control polices is simply rejected, rather than modified. Only valid queries, i.e., queries could be rewritten using only the authorization views, are answered.

Both approaches in [37] specify access control by means of positive authorizations, i.e., they represent privileges granted to a user. However, the lack of a positive authorization for a given user does not prevent this user from receiving this authorization [7]. For example, the user can derive such authorization from its ancestor users. In contrast, our work makes sure that certain users are never allowed access to sensitive data.

In [9, 10, 42], they take a different approach to address privacy issues in incomplete propositional databases. The approach, *Control Query Evaluation* (CQE), 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. For each query posed to the database by that user, it is checked whether the answer to that query would allow the user to infer any sensitive information. If this is the case, the answer is distorted by either *lying or* refusal or *combined lying and refusal.* In [25], they extend CQE to restricted incomplete first-order logic databases. In that case, CQE is used through a transformations from the specification language to the corresponding propositional language.

This approach does not seem to be comparable to ours. This approach has a very high complexity even with propositional logic. In addition, their work does not consider the possible presence of *null* in the database. In contrast, we assume that the database may contains *null* and *null* is treated as `NULL` in the SQL standard. Furthermore, it is very natural to expect to obtain as many “useful” answers as possible while still protecting sensitive information. However, there is no formal definition for “maximum” useful answers in their work. Hence, it is impossible to prove the approach provide the best answers. In our work, the idea of “maximum” useful answers was captured by secrecy instances which minimizes the difference to the original database.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this work, we have developed a logical framework and a methodology to answer conjunctive queries that do not reveal secret information as specified by secrecy views. We have concentrated on the case of conjunctive secrecy views and conjunctive queries, but it is possible to relax these restrictions. 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 tuples showing only null values. The queries can be posed against any of these virtual instances or cautiously against all of them, simultaneously.

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

We introduced disjunctive logic programs with stable model semantics to specify the secrecy instances. These programs can be used to compute secret answers. We also presented a optimization methodology, which captures the relevant database predicates and facts to compute a specific query.

8.2 Future Work

Our work leaves several open problems that 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. 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.

- Another dimension of the problem consists in adding ICs to the schema. If they are known to the user and also 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. In this case, the definition of secrecy instance should have the extra condition of satisfying the ICs, because ICs could be violated through the process of updating databases.

Bibliography

[1] SQL-2003 standard part 1: Framework (sql/framework). Available at: http://www.wiscorp.com/SQLStandards.html.

[2] SQL-2003 standard part 2: Foundation (sql/foundation). Available at: http://www.wiscorp.com/SQLStandards.html.

[3] The virtual private database in oracle9ir2. Available at: http://otn.oracle.com/deploy/security/oracle9ir2/pdf/vpd9ir2twp.pdf.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[5] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 68–79, New York, NY, USA, 1999. ACM.

[6] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[7] E. Bertino, P. Samarati, and S. Jajodia. An extended authorization model for relational databases. *IEEE Trans. on Knowl. and Data Eng.*, 9:85–101, January 1997.

[8] L. Bertossi. Consistent query answering in databases. *SIGMOD Rec.*, 35:68–76, June 2006.

[9] J. Biskup and T. Weibert. Confidentiality policies for controlled query evaluation. In *Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 1–13, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] J. Biskup and T. Weibert. Keeping secrets in incomplete databases. *Int. J. Inf. Secur.*, 7:199–217, May 2008.

[11] L. Bravo. Handling inconsistency in databases and data integration systems. In *PhD. Thesis.* Carleton University, Department of Computer Science, 2007.

[12] L. Bravo and L. Bertossi. Semantically correct query answers in the presence of null values. In *Proceedings of EDBT WS on Inconsistency and Incompleteness in Databases (IIDB 06)*, volume 5956 of *LNCS*, pages 230–247. Springer, 2006.

[13] M. Caniupan. Optimizing and implementing repair programs for consistent query answering in databases. In *PhD. Thesis.* Carleton University, Department of Computer Science, 2007.

[14] M. Caniupan and L. Bertossi. Optimizing repair programs for consistent query answering. In *Proceedings of the 25th International Conference on The Chilean Computer Science Society*, pages 3–12, Washington, DC, USA, 2005. IEEE Computer Society.

[15] M. Caniupan, L. Bertossi, and L. Bravo. Optimizing repair programs and their evaluation for consistent query answering in databases (extended version). In *Proceedings of the 25th International Conference on The Chilean Computer Science Society*, pages 3–12, Washington, DC, USA, 2005. IEEE Computer Society.

[16] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197:90–121, February 2005.

[17] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4:397–434, December 1979.

[18] A. Dreibelbis, E. Hechler, I. Milman, M. Oberhofer, P. Run, and D. Wolfson. *Enterprise Master Data Management: An SOA Approach to Managing Core Information.* IBM Press, 2008.

[19] S. Dwivedi and B. Menezes. Database access control for e-business - a case study. In *Proceedings of the 11th International Conference on Management of Data*, COMAD '05, pages 168–175, 2005.

[20] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Commun.*, 12:99–111, January 1999.

[21] W. Faber, G. Greco, and N. Leone. Magic sets and their application to data integration. *J. Comput. Syst. Sci.*, 73:584–609, June 2007.

[22] G. H. Gessert. Four valued logic for relational database systems. *SIGMOD Rec.*, 19:29–35, March 1990.

[23] A. Gupta and I. S. Mumick. Maintenance of materialized views: problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 12(2):3–18, June 1995.

[24] T. Imieliński and W. Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31:761–791, September 1984.

[25] C. T. Joachim Biskup and L. Wiese. Towards controlled query evaluation for incomplete first-order databases. In *Proceedings of the 6th International Symposium on Foundations of Information and Knowledge Systems*, volume 5956 of *LNCS*, pages 230–247. Springer, 2010.

[26] R. Kocharekar. Nulls in relational databases: revised. *SIGMOD Rec.*, 18:68–73, March 1989.

[27] J. Lechtenbörger. The impact of the constant complement approach towards view updating. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '03, pages 49–55, New York, NY, USA, 2003. ACM.

[28] J. Lechtenbörger and G. Vossen. On the computation of relational view complements. *ACM Trans. Database Syst.*, 28:175–208, June 2003.

[29] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the 30th International Conference on Very Large Data Bases*, VLDB '04, pages 108–119. VLDB Endowment, 2004.

[30] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7:499–562, July 2006.

[31] N. Leone, L. Vicenzino, and G. Terracina. Dlvdb: Bridging the gap between asp systems and dbmss. In *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning*, volume 2923 of *LNAI*, pages 341–345. Springer, 2004.

[32] M. Levene and G. Loizou. Null inclusion dependencies in relational databases. *Information and Computation*, 136(2):67–108, 1994.

[33] J. W. Lloyd. *Foundations of logic programming; (2nd extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[34] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. *SIGMOD Rec.*, 19:247–258, May 1990.

[35] L. E. Olson, C. A. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 289–298, New York, NY, USA, 2008. ACM.

[36] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23:3–13, 2000.

[37] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 551–562, New York, NY, USA, 2004. ACM.

[38] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: trading space for time. *SIGMOD Rec.*, 25:447–458, June 1996.

[39] G. Terracina, N. Leone, V. Lio, and C. Panetta. Adding efficient data management to logic programming systems. In *Proceedings of the 16th International Symposium on Methodologies for Intelligent Systems (ISMIS 2006)*, volume 4203 of *LNCS*, pages 524–533. Springer, 2006.

[40] C. Türker and M. Gertz. Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. *The VLDB Journal*, 10:241–269, December 2001.

[41] Y. Vassiliou. Null values in data base management a denotational semantics approach. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 162–169, New York, NY, USA, 1979. ACM.

[42] T. Weibert. A framework for inference control in incomplete logic databases. In *PhD. Thesis*. Technische Universität Dortmund, 2008.

[43] C. Zaniolo. Database relations with null values. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '82, pages 27–33, New York, NY, USA, 1982. ACM.

[44] Z. Zhang and A. O. Mendelzon. Authorization views and conditional query containment. In *Proceedings of the 10th International Conference on Database Theory (ICDT 2005)*, volume 3363 of *INCS*, pages 259–273. Springer, 2005.

Appendix A

Optimized Secrecy Programs

Program 7. Optimized Logic Program for Query $\mathcal{Q}_1$.

$$
\begin{array}{rl}
\#import(company, \text{“}db2admin\text{”}, \text{“}test\text{”}, & \text{“}\ \texttt{SELECT * FROM Employee} \\
 & \texttt{WHERE Salary} < \texttt{40000}\text{”}, Emp, \\
 & type: Q\_Const, Q\_Const, \\
 & Q\_Const, Q\_Const). \\
Employee\_(eno, ename, salary, dno, \mathbf{t_d}) & :-\ Emp(eno, ename, salary, dno). \\
Employee\_(\text{“}null\text{”}, \text{“}null\text{”}, salary, dno, \mathbf{a_u}) & \vee\ Employee\_(eno, ename, \text{“}null\text{”}, dno, \mathbf{a_u}) \\
 & :-\ Employee\_(eno, ename, salary, dno, \mathbf{t}), \\
 & salary < \text{“}30000\text{”}, salary \neq \text{“}null\text{”}, \\
 & auxEmp(eno, ename). \\
auxEmp(eno, ename) & :-\ Employee\_(eno, ename, salary, dno, \mathbf{t}), \\
 & salary < \text{“}30000\text{”}, salary \neq \text{“}null\text{”}, \\
 & eno \neq \text{“}null\text{”}. \\
auxEmp(eno, ename) & :-\ Employee\_(eno, ename, salary, dno, \mathbf{t}), \\
 & salary < \text{“}30000\text{”}, salary \neq \text{“}null\text{”}, \\
 & ename \neq \text{“}null\text{”}. \\
Employee\_(eno, ename, salary, dno, \mathbf{b_u}) & :-\ Employee\_(eno, ename, salary, dno, \mathbf{t}), \\
 & salary < \text{“}30000\text{”}, salary \neq \text{“}null\text{”}, \\
 & auxEmp(eno, ename), \\
 & Employee\_(eno, ename, \text{“}null\text{”}, dno, \mathbf{a_u}). \\
Employee\_(eno, ename, salary, dno, \mathbf{b_u}) & :-\ Employee\_(eno, ename, salary, dno, \mathbf{t}), \\
 & salary < \text{“}30000\text{”}, salary \neq \text{“}null\text{”}, \\
 & auxEmp(eno, ename), \\
 & Employee\_(\text{“}null\text{”}, \text{“}null\text{”}, salary, dno, \mathbf{a_u}), \\
 & eno \neq \text{“}null\text{”}, ename \neq \text{“}null\text{”}. \\
Employee\_(eno, ename, salary, dno, \mathbf{t}) & :-\ Employee\_(eno, ename, salary, dno, \mathbf{t_d}). \\
Employee\_(eno, ename, salary, dno, \mathbf{t}) & :-\ Employee\_(eno, ename, salary, dno, \mathbf{a_u}). \\
Employee\_(eno, ename, salary, dno, \mathbf{s}) & :-\ Employee\_(eno, ename, salary, dno, \mathbf{t}), \\
not & Employee\_(eno, ename, salary, dno, \mathbf{b_u}). \\
Ans(eno, ename) & :-\ Employee(eno, ename, salary, dno, \mathbf{s}), \\
 & salary \neq \text{“}null\text{”}, salary < 40000. \\
Ans(eno, ename) & ?
\end{array}
$$

Program 8. Optimized Logic Program for Query $\mathcal{Q}_2$.

```
#import(company, "db2admin," "test",  " SELECT * FROM Employee", Emp).
#import(company, "db2admin", "test",  " SELECT * FROM Department
                                        WHERE Dname = 'sales'", Dept).
           Department_(dno, dame, t_d) :- Dept(dno, dname).
        Department_(dno, "null", a_u) :- Department_(dno, "Operations", t).
  Department_(dno, "Operations", b_u) :- Department_(dno, "Operations", t),
                                         Department_(dno, "null", a_u).
           Department_(dno, dname, t) :- Department_(dno, dname, t_d).
           Department_(dno, dname, t) :- Department_(dno, dname, a_u).
           Department_(dno, dname, s) :- Department_(dno, dname, t),
                                     not Department_(dno, dname, b_u).
 Employee_(eno, ename, salary, dno, t_d) :- Emp(eno, ename, salary, dno).
Employee_("null", "null", salary, dno, a_u) ∨ Employee_(eno, ename, "null", dno, a_u)
                                       :- Employee_(eno, ename, salary, dno, t),
                                          salary < "30000", salary ≠ "null",
                                          auxEmp(eno, ename).
                    auxEmp(eno, ename) :- Employee_(eno, ename, salary, dno, t),
                                          salary < "30000", salary ≠ "null",
                                          eno ≠ "null".
                    auxEmp(eno, ename) :- Employee_(eno, ename, salary, dno, t),
                                          salary < "30000", salary ≠ "null",
                                          ename ≠ "null".
 Employee_(eno, ename, salary, dno, b_u) :- Employee_(eno, ename, salary, dno, t),
                                          salary < "30000", salary ≠ "null",
                                          auxEmp(eno, ename),
                                          Employee_(eno, ename, "null", dno, a_u).
 Employee_(eno, ename, salary, dno, b_u) :- Employee_(eno, ename, salary, dno, t),
                                          salary < "30000", salary ≠ "null",
                                          auxEmp(eno, ename),
                                          Employee_("null", "null", salary, dno, a_u),
                                          eno ≠ "null", ename ≠ "null".
   Employee_(eno, ename, salary, dno, t) :- Employee_(eno, ename, salary, dno, t_d).
   Employee_(eno, ename, salary, dno, t) :- Employee_(eno, ename, salary, dno, a_u).
   Employee_(eno, ename, salary, dno, s) :- Employee_(eno, ename, salary, dno, t),
                                      not Employee_(eno, ename, salary, dno, b_u).
                          Ans(ename) :- Employee(eno, ename, salary, dno, s),
                                          Department(dno, dname, s),
                                          dname = "sales",
                                          dname ≠ "null", dno ≠ "null".
                          Ans(ename)  ?
```

Program 9. Optimized Logic Program for Query $\mathcal{Q}_3$.

$$\begin{aligned}
\#\mathit{import}(\mathit{company}, \text{“}\mathit{db2admin}\text{”}, \text{“}\mathit{test}\text{”},\ & \text{“}\ \texttt{SELECT * FROM Department}\\
& \texttt{WHERE Dname = `sales'}\\
& \texttt{AND Dno =' 101'}\text{”}, \mathit{Dept},\\
& \mathit{type}: \mathit{Q\_Const}, \mathit{Q\_Const}).\\
\mathit{Department\_}(\mathit{dno}, \mathit{dame}, \mathbf{t_d}) :-\ & \mathit{Dept}(\mathit{dno}, \mathit{dname}).\\
\mathit{Department\_}(\mathit{dno}, \text{“}\mathit{null}\text{”}, \mathbf{a_u}) :-\ & \mathit{Department\_}(\mathit{dno}, \text{“}\mathit{Operations}\text{”}, \mathbf{t}).\\
\mathit{Department\_}(\mathit{dno}, \text{“}\mathit{Operations}\text{”}, \mathbf{b_u}) :-\ & \mathit{Department\_}(\mathit{dno}, \text{“}\mathit{Operations}\text{”}, \mathbf{t}),\\
& \mathit{Department\_}(\mathit{dno}, \text{“}\mathit{null}\text{”}, \mathbf{a_u}).\\
\mathit{Department\_}(\mathit{dno}, \mathit{dname}, \mathbf{t}) :-\ & \mathit{Department\_}(\mathit{dno}, \mathit{dname}, \mathbf{t_d}).\\
\mathit{Department\_}(\mathit{dno}, \mathit{dname}, \mathbf{t}) :-\ & \mathit{Department\_}(\mathit{dno}, \mathit{dname}, \mathbf{a_u}).\\
\mathit{Department\_}(\mathit{dno}, \mathit{dname}, \mathbf{s}) :-\ & \mathit{Department\_}(\mathit{dno}, \mathit{dname}, \mathbf{t}),\\
& \mathit{not}\ \mathit{Department\_}(\mathit{dno}, \mathit{dname}, \mathbf{b_u}).\\
\mathit{Ans} :-\ & \mathit{Department}(\mathit{dno}, \mathit{dname}, \mathbf{s}),\\
& \mathit{dno} = \text{“101”}, \mathit{dname} = \text{“}\mathit{sales}\text{”},\\
& \mathit{dno} \neq \text{“}\mathit{null}\text{”}, \mathit{dname} \neq \text{“}\mathit{null}\text{”}.\\
\mathit{Ans}\ \ & ?
\end{aligned}$$