Journal of Computer Security 1 (1992), 189–213
IOS Press

A UNIFIED MODEL FOR SECURITY AND INTEGRITY IN RELATIONAL DATABASES*

Amihai Motro
Department of Information and Software Systems Engineering
George Mason University
Fairfax, VA 22030-4444
USA

Abstract

The issues of database security and integrity are tightly related. Security mechanisms and integrity mechanisms are both concerned with protection of information, and both involve management of meta-information and procedures for screening transactions. Yet in database systems these tasks are entirely separate. In this paper we describe a unified model that accommodates both security and integrity, with all protective restrictions stated in terms of database *views*. These protective restrictions are treated as *knowledge*, from which transaction screening procedures *infer* the restrictions that apply to individual transactions. A transaction may be allowed or denied in its entirety, or specific non-violating subtransactions may be identified. This process is an application of the view inference problem, to which we offer two alternative solutions. We then show how users can exploit information made available by the integrity mechanism to bypass the security mechanism, and discuss how such security breaches can be avoided. Finally, we show how the model can accommodate a broader concept of integrity that was introduced recently.

1. Introduction

The issues of database *security* and *integrity* are tightly related. Security is concerned with the protection of data from unauthorized retrieval or modification. Integrity is concerned with the maintenance of the soundness and completeness of the data. Clearly, security violations may lead to integrity violations. In other words, to maintain integrity requires imposition of security measures. A closer look at how security and integrity are handled in database systems reveals further commonalities.

The prevailing approach towards the maintenance of security in relational databases is to use the mechanism of views. A *view* (or a *virtual* relation) is a relation that is defined in terms of the actual database relations. Most security models

* This work was supported in part by NSF Grant IRI-9007106.

for relational databases state access permissions by means of triplets: *(user, permission, view)*. Each such triplet grants *user* a specific *permission* (e.g., retrieval, modification) to access *view*. When a transaction is submitted to the database system, the permissions table is consulted to determine whether the transaction should be allowed.

The prevailing approach to the maintenance of integrity is to use the mechanism of constraints. In general, a *constraint* is an assertion that expresses a relationship that must be satisfied by the data in the database. Assuming that initially a database satisfies the constraints, modifying transactions are thereafter allowed only if they do not violate any of the constraints.

Thus, the function of an integrity mechanism is to manage integrity constraints, and reject transactions that violate these constraints, creating discord between the database and the real world; and the function of a security mechanism is to manage access permissions, and reject transactions that violate these permissions, possibly creating discord between the database and the real world. Hence, security and integrity mechanisms are both concerned with the *protection of information*, and both involve (1) management of meta-information (integrity constraints or access permissions), and (2) procedures for screening transactions.

Yet these tasks remain entirely separate. Because the meta-information involved is conceived as very different in both form and meaning, different theoretical models have been developed for these two tasks, and database systems maintain two separate subsystems for implementing these tasks.

In this paper we describe a unified model that accommodates both security and integrity. Our model draws from our earlier work on security [9] and integrity [10]. A brief discussion of related work in security and integrity will help explain the other novel aspects of our model.

1.1. *Background*

Our framework for database security is essentially *discretionary* access control [5, 4], whereby access privileges are established by the *users* of the system (this is in contrast with *mandatory* access controls, whereby access privileges are imposed by the *system*). In particular, our model is in the broad class of models that assume that access privileges are described in a *matrix*, with rows representing users, columns representing data objects, and each matrix position denoting the permissions granted to a specific user for a specific object. Two well-documented database security models are discussed below.

The authorization scheme of System R [6] allows permissions to be granted to both actual relations and views. It is a flexible scheme that incorporates useful features, such as allowing users to grant other users permissions that were granted to them. In our opinion, however, the mechanism has a serious limitation. Consider the situation of two actual relations A and B, and assume that we want to permit access to a particular view of A and B. We define this view V and grant access permission to V, but not to A or B. Authorization is now granted only to queries that access V. Queries that accesses A or B, will be rejected for lack of access permissions to these relations, even if the requests are *within the permissions* (as described in view V). Thus, V is not only a *statement* of the permissions, but the actual "access window", as well.

The authorization scheme of INGRES [14, 7] is quite different. Queries are handled by a "query modification" algorithm. Essentially, the algorithm searches for permitted views whose attributes *contain* the attributes addressed by the query, and the qualifications placed on these attributes in the views are then *conjoined* with the qualification specified in the query. The algorithm is attractive because when a query exceeds its permissions, it delivers the data that are within the permissions. However, there are several limitations. First, permissions are granted only for actual relations or views of single relations, and it is not possible to grant permissions to views of several relations. Second, the algorithm does not handle rows and columns symmetrically. Consider relation A with attributes A_1, A_2 and A_3, and assume permission is granted to the tuples of A_1 and A_2 that satisfy a condition P. A request to retrieve A_1 and A_2 would be reduced to the tuples of A_1 and A_2 that satisfy P. However, a request to retrieve A_1, A_2 and A_3 would be denied altogether, where one would expect that it would be reduced to tuples of A_1 and A_2. In addition, there are various cases where the algorithm actually delivers less than what the user is permitted to view.

As mentioned earlier, the prevailing approach to integrity is to guarantee that the database always satisfies a set of constraints. However, for reasons of efficiency, most database systems do not permit constraints that are arbitrarily complex assertions. Usually, only very specific types of assertions are permitted [17]; most notably, *domain* constraints, that restrict the values of an attribute to be within a prescribed range, *referential* constraints, that restrict the values of an attribute to values of another attribute, and *key* constraints, that require the values of an attribute to be different for every tuple. Thus, this integrity model remains largely unimplemented.

The integrity model based on constraints is quite limited, as it cannot provide the ultimate guarantee of integrity, which is correspondence between the database and the real world; i.e., a database has integrity if it includes all the information that it attempts to model, and nothing but it. In [10], a model is described that supports this more general concept of integrity.

1.2. *Outline of Approach*

The model described in this paper is notable for five aspects. First and foremost, it integrates the security and integrity functions. Essentially, this is done by phrasing integrity constraints as *views to which permissions are never granted.* Consequently, access permissions and integrity constraints (referred to collectively as *protective restrictions*) are both treated as views with specific permissions. This uniformity is then exploited both in the *static* component of the model, which is concerned with the statement of security and integrity restrictions, and in the *dynamic* component of the model, which is concerned with the enforcement of these restrictions, by means of appropriate transaction screening procedures.

Second, the protective restrictions are regarded as a form of *knowledge*, from which the restrictions that apply to individual access requests are *inferred.* In particular, users direct transactions at the actual database, not at views. This is in marked contrast with the System R approach, where access permissions force users to access the database through specific "windows".

By inferring the individual restrictions that apply to each transaction, the model can avoid total rejection of transactions that violate the restrictions. When a transaction violates the restrictions, the model discovers the sub-transactions that can be allowed. This aspect is reminiscent of the query modification scheme used in INGRES, but without most of the limitations of the latter, as described earlier.

Prevailing security and integrity mechanisms allow a database user to combine knowledge of system-wide integrity constraints with information that has been made available to this user through access permissions, and infer information that has not been made available to this user. Our model provides the analytical tools to pinpoint the combinations of integrity constraints and access permissions that might compromise security, and hence set policies that avoid such risks.

Finally, the model can accommodate the new integrity model proposed in [10]. In addition to access permissions and integrity constraints, the model would incorporate *completeness constraints* and *soundness constraints*. These constraints, which are also expressed as views, allow the database system and its administrators to monitor the correspondence between the database and the real world.

The remainder of this paper is organized as follows. Section 2 describes our unified model for security and integrity. An essential element of the model is the *view inference problem*. Two possible solutions to this problem are discussed in Sections 3 and 4. Section 5 discusses undesirable interaction between integrity constraints and access permissions. Section 6 introduces a broader concept of integrity, and shows how it is accommodated by the model. Section 7 concludes with a brief summary and discussion of issues requiring further research.

2. A Model for Security and Integrity

The model is described in two parts. A *static* part that is concerned with the statement of protective restrictions (access permissions and integrity constraints), and a *dynamic* part that is concerned with procedures for enforcement of these restrictions by means of appropriate transaction screening procedures. The screening procedures either *accept* or *deny* each transaction. A refinement is then described that allows *partial acceptance* of transactions. We begin by establishing several preliminary concepts.

2.1. *Preliminaries*

We assume the following definition of a relational database [8]. A *relation scheme* R is a finite set of *attributes* $A_1, \ldots, A_m$. With each attribute A_i a set of values D_i, called the *domain* of A_i, is associated (domains are non-empty, finite or countably infinite sets). A *relation* on the relation scheme R is a subset of the product of the domains associated with the attributes of R. A *database scheme* $\mathcal{R}$ is a set of relation schemes $R_1, \ldots, R_n$. A database instance $\boldsymbol{D}$ of the database scheme $\mathcal{R}$ is a set of relations $R_1(\boldsymbol{D}), \ldots, R_n(\boldsymbol{D})$, where each $R_i(\boldsymbol{D})$ is a relation on the relation scheme R_i. A *view* V is an expression in the relation schemes of $\mathcal{R}$ that defines a new relation scheme, and for each database instance $\boldsymbol{D}$ defines a unique relation on this scheme denoted $V(\boldsymbol{D})$[1].

[1] In particular, a view can be simply an actual relation.

Database views may possess properties. A *property* is a proposition which is either true or false for any given view. A property p is *inherited*, if all views derived from views with property p, also have property p. Note that inheritance is relative to a particular set of relation operators.

An example of an inherited property is permission to access. If user u has permission to access $V_1, \ldots, V_n$, then u also has permission to access any relation that can be derived from $V_1, \ldots, V_n$. Another example is being error-free. If the information included in $V_1, \ldots, V_n$ is known to be free of errors, then the information in any relation that can be derived from $V_1, \ldots, V_n$ is free of errors.

Given that specified parts of the database possess a particular property, it is sometimes important to determine the parts of an arbitrary view that inherit the property. Formally, this question is stated in the *view inference problem* defined as follows.

Assume views $V_1, \ldots, V_n$ have property p, and consider view V. Which views of V have property p? In particular, does V itself have property p? (i.e., can V be derived from $V_1, \ldots, V_n$?)

For now, assume there is a procedure that can determine these questions. Finally, we establish that integrity constraints can be stated as views with a particular property.

A general form for integrity constraints is $(\forall x_1)\ldots(\forall x_n)(\alpha(x_1, \ldots, x_n) \Longrightarrow \beta(x_1, \ldots, x_n))$, where x_i are domain variables and α and β are safe relational calculus expressions with these free variables. Such constraints may be rewritten as $\{x_1, \ldots, x_n \mid \alpha(x_1, \ldots, x_n) \wedge \neg\beta(x_1, \ldots, x_n)\} = \emptyset$. In other words, the constraint $(\forall x_1)\ldots(\forall x_n)(\alpha(x_1, \ldots, x_n) \Longrightarrow \beta(x_1, \ldots, x_n))$ is satisfied by a database, if and only if the view $\{x_1, \ldots, x_n \mid \alpha(x_1, \ldots, x_n) \wedge \neg\beta(x_1, \ldots, x_n)\}$ is empty.

2.2. *Stating Protective Restrictions*

Databases are accessed by means of *transactions* submitted by *users*. Each transaction specifies a *view* that is either to be *retrieved* or to be *modified*. Database security is enforced by granting users permission to retrieve or to modify specific views, and allowing only transactions that are within the permissions granted to the users who submit them. Database integrity is enforced by stating constraints (expressed as views), and allowing only transactions that do not cause the database to violate the constraints (i.e., cause these views to be non-empty).

Thus, there are views to which users are granted retrieval or modification permissions, and there are views that must be empty, and to which no permissions are ever granted[2]. Altogether, protective restrictions are stated by associating three properties with views:

- *Retrieve permission granted to user* u, denoted R/u. View V has property R/u, if user u was granted permission to retrieve view V.
- *Modify permission granted to user* u, denoted M/u. View V has property M/u, if user u was granted permission to modify view V.

[2] To maintain emptiness, it is enough to disallow *insertions* into such views; however, as deletions, modifications, and retrievals will always fail, we simply withhold all permissions to these views.

- *Empty (permissions never granted)*, denoted $\perp$. View V has property $\perp$, if V is empty, and permissions are never granted to V.

This information is stated in a table *VP = (View,Property)*, where *view* identifies a view definition, and *property* is either R/u, M/u or $\perp$, where u identifies a user.

Clearly, R/u, M/u and $\perp$ are inherited properties. Thus, a user who was granted permission to retrieve (modify) a set of views, also has permission to retrieve (modify) any view of these views. Similarly, any view of a given set of empty views (satisfied constraints) is also an empty view (satisfied constraint). Note that $\perp$ *overrides* R/u and M/u: u should not be allowed to access a view with property $\perp$, even if the view has properties R/u or M/u.

While we assume that users are granted permission to modify *views* of the database, it is well-known that modifications to views cannot always be propagated to the actual relations [3]. For example, given the actual relations *Supplier = (Sname,Scity)* and *Customer = (Cname,Ccity)*, insertions of new tuples into the view *Associates* $= \{(x_1, x_2) \mid (x_1, x_2) \in Supplier \vee (x_1, x_2) \in Customer\}$ cannot be propagated unambiguously to the actual relations. On the other hand, insertions of new tuples into the view *Fairfax_suppliers* $= \{(x_1, x_2) \mid (x_1, x_2) \in Supplier \wedge (x_2 = \text{"}Fairfax\text{"})\}$ pose no problems. The issue whether a modification to a view *can be performed* is separate from the issue whether a modification to a view *should be allowed*. In the following discussion we shall assume that the views to which users are granted modification permission, and the views which users attempt to modify, are all views that can indeed be modified.

Employee:

Name	*Rank*	*Salary*	*Department*
Andy	*senior*	*43,000*	*strip*
Calvin	*junior*	*35,000*	*strip*
Cathy	*junior*	*48,000*	*strip*
Dennis	*junior*	*38,000*	*panel*
Herman	*senior*	*55,000*	*panel*
Ziggy	*senior*	*67,000*	*panel*

Department:

Dname	*Manager*
panel	*Herman*
strip	*Cathy*

Figure 1: Database Instance

Assume a database with relations *Employee* $= (Name, Rank, Salary, Department)$ and *Department (Dname,Manager)*. An instance of this relation is shown in Figure 1. Consider these four views:

1. Names and salaries of all employees:
$$V_1 = \{(x_1, x_2) \mid (\exists y_1, y_2)(x_1, y_1, x_2, y_2) \in Employee\}.$$
2. Names and ranks of employees earning less than 50,000:
$$V_2 = \{(x_1, x_2) \mid (\exists y_1, y_2)(x_1, x_2, y_1, y_2) \in Employee \wedge (y_1 \leq 50,000)\}.$$
3. Junior employees earning more than 50,000:
$$V_3 = \{(x_1, x_2, x_3, x_4) \mid (x_1, x_2, x_3, x_4) \in Employee \wedge (x_2 = \text{"}junior\text{"}) \wedge (x_3 > 50,000)\}.$$

4. Department managers who are not employees:

$$V_4 = \{(x) \mid (\exists y_1)(y_1, x) \in \mathit{Department} \\ \wedge\ ((\forall y_2, y_3, y_4)(x, y_2, y_3, y_4) \notin \mathit{Employee})\}.$$

Figure 2 shows an example of the VP table. Jones has permission to retrieve view V_1, Smith has permission to retrieve and modify view V_2, and views V_3 and V_4 are empty (all junior employees earn less than 50,000, and all department managers are employees).

View	*Property*
V_1	*R/Jones*
V_2	*R/Smith*
V_2	*M/Smith*
V_3	$\perp$
V_4	$\perp$

Figure 2: Example of the VP table

2.3. *Screening Transactions*

Assuming the above database, the following four examples demonstrate the permissions needed for various transactions. A transaction that lists the names and salaries of the senior employees requires a permission to retrieve the view

$$T_1 = \{(x_1, x_2) \mid (\exists y_1, y_2)(x_1, y_1, x_2, y_2) \in \mathit{Employee} \wedge (y_1 = \text{“}\mathit{senior}\text{”})\}.$$

A transaction that updates the ranks of the employees earning less than 20,000 requires a permission to modify the view

$$T_2 = \{(x) \mid (\exists y_1, y_2, y_3)(y_1, x, y_2, y_3) \in \mathit{Employee} \wedge (y_2 \leq 20,000)\}.$$

A transaction that deletes the employees earning more than 100,000 requires a permission to modify the view

$$T_3 = \{(x_1, x_2, x_3, x_4) \mid (x_1, x_2, x_3, x_4) \in \mathit{Employee} \wedge (x_3 > 100,000)\}.$$

A transaction that inserts an employee Marvin with rank junior, salary 40,000 and department strip requires a permission to modify the view

$$T_4 = \{(x_1, x_2, x_3, x_4) \mid (x_1, x_2, x_3, x_4) \in \mathit{Employee} \wedge (x_1 = \text{“}\mathit{Marvin}\text{”}) \\ \wedge (x_2 = \text{“}\mathit{junior}\text{”}) \wedge (x_3 = 40,000) \wedge (x_4 = \text{“}\mathit{strip}\text{”})\}.$$

Note that views for deletions and insertions show *all* the attributes of the tuples that are deleted or inserted.

These transactions should be allowed if their views are either *stated* in VP with the appropriate properties, or can be *derived* from stated views with the appropriate properties. For example, permission to retrieve the names and salaries

of all employees implies permission to retrieve the names and salaries of senior employees; i.e., T_1 is a view of V_1. Similarly, permission to modify the names and ranks of employees earning less than 50,000 implies permission to modify the ranks of employees earning less than 20,000; i.e., T_2 is a view of V_2. The procedures for screening transactions must therefore determine whether the transaction view is a view of permitted views.

Assuming the same database, consider a transaction that lists the names of the junior employees earning more than 60,000:

$$T_5 = \{(x) \mid (\exists y_1, y_2, y_3)(x, y_1, y_2, y_3) \in \textit{Employee} \\ \wedge (y_1 = \text{"}\textit{junior}\text{"}) \wedge (y_2 > 60,000)\}.$$

If T_5 *appears* in *VP* with the property $\perp$, or can be *derived* from views in *VP* with property $\perp$, this transaction is *unsatisfiable*. Since emptiness of the set of junior employees earning more than 50,000 implies emptiness of the set of junior employees earning more than 60,000 (i.e., T_5 is a view of V_3), this transaction is indeed unsatisfiable. The procedures for screening transactions must therefore determine whether the transaction view is a view of empty views.

Thus, screening transactions for both security and satisfiability is essentially determining whether the transaction view has a specific property (R/u, M/u, or $\perp$). Unfortunately, screening transactions for integrity is not as simple.

While screening for security and satisfiability involves testing whether the transaction view has a particular property, screening for integrity requires testing whether a transaction will invalidate properties stated in *VP*. Detecting such occurrences is simple, when the transaction itself incorporates a statement that contradicts the properties stated in *VP*. For example, a transaction that attempts to insert a junior employee with salary 60,000, postulates the existence of a junior employee with salary 60,000, a statement which contradicts the statement in *VP* that junior employees may not earn more than 50,000. In general, however, a transaction may invalidate properties stated in *VP*, without incorporating a statement which, added to *VP*, causes a contradiction. For example, consider a transaction that inserts a new department, but specifies a manager who is not an employee. The statement by itself does not contradict the statements in *VP*. The contradiction is detected only after the entire set of employees is considered!

Note that an integrity constraint is violated if a transaction inserts a tuple into its view, but the transaction need not be an insertion, as deletions and updates can also effect insertions into views. For example, the constraint that all department managers are employees could be violated (i.e., V_4 could become non-empty), when an employee is deleted.

Hence, screening for integrity may require accessing the database itself. A simple procedure is to accept the transaction provisionally, and then evaluate the views in *VP* with property $\perp$ that involve attributes modified (by insertion, deletion or update) by the transaction. If any of these view is not empty, the transaction is rejected and undone[3].

The above discussions are summarized in the following two procedures.

[3] It helps if views in *VP* with property $\perp$ are indexed by their attributes.

Procedure 1: Retrievals. User u submits transaction t, requesting to retrieve view R.

1. Retrieve from VP the views with property $\perp$. Let these views be $C_1, \ldots, C_n$.
2. Apply the view inference algorithm to R and $C_1, \ldots, C_n$ to determine whether R has property $\perp$.
3. If R has property $\perp$, then deny transaction t on grounds of unsatisfiability.
4. Otherwise, retrieve from VP the views with property R/u. Let these views be $V_1, \ldots, V_m$.
5. Apply the view inference algorithm to R and $V_1, \ldots, V_m$, to determine whether R has property R/u.
6. If R has property R/u, then allow transaction t; otherwise, deny t on grounds of security violation.

Procedure 2: Modifications. User u submits a transaction t, requesting to modify view M.

1. Perform the transaction provisionally.
2. Retrieve from VP the views with property $\perp$ that reference the attributes modified by M. Let these views be $C_1, \ldots, C_n$.
3. Materialize $C_1, \ldots, C_n$.
4. If any C_i is non-empty, then deny transaction t on grounds of integrity violation, and undo t.
5. Otherwise, retrieve from VP the views with property M/u. Let these views be $V_1, \ldots, V_m$.
6. Apply the view inference algorithm to M and $V_1, \ldots, V_m$, to determine whether M has property M/u.
7. If M has property M/u, then allow transaction t; otherwise, deny t on grounds of security violation.

Note that these procedures maintain the precedence of $\perp$ over R/u and M/u; i.e., users are prevented from accessing views that have the property $\perp$, even if these views have the properties R/u or M/u.

2.4. *Allowing Parts of Transactions*

The verdict of the screening procedures is to allow or deny entire transactions. When a transaction is denied on grounds of a security violation, it may be useful to determine whether some *parts* of the transaction are within the permissions.

Consider again a retrieval transaction t submitted by user u, that is denied because its view R does not have the property R/u. Recall that the view inference problem is to find the *views* of a given view V that have the property p. The previous procedures applied the problem to determine only whether V *itself* has the property p. Assume now that the algorithm finds the views of R that have the property R/u. These views are *partial requests* which u is entitled to submit.

They can be *imposed* on the complete answer R, to determine the parts that can be delivered and the parts that must be "masked".

Procedure 1 can now be improved, as follows. Steps 1–3 screen t for unsatisfiability, and remain unchanged. Steps 4–6 are replaced by

4. Perform transaction t, retrieving view R.
5. Retrieve from VP the views with property R/u. Let these views be $V_1, \ldots, V_m$.
6. Apply the view inference algorithm to R and $V_1, \ldots, V_m$, to determine the views of R that have property R/u. Let these views be $U_1, \ldots, U_k$
7. Retrieve views $U_1, \ldots, U_k$ from view R.

The effect of the last step is to delete from R entire tuples or columns that exceed the permissions. This result is accompanied by the definitions of $U_1, \ldots, U_k$, to define precisely the part actually delivered.

As an example, consider the following transaction, submitted by Smith, to list the names, ranks and salaries of employees earning more than 40,000:

$$T_6 = \{(x_1, x_2, x_3) \mid (\exists y_1)(x_1, x_2, x_3, y_1) \in \mathit{Employee} \wedge (x_3 > 40,000)\}.$$

Smith is allowed to retrieve only V_2, which is the names and ranks of employees earning less than 50,000. The view of T_6 which is within the permission (i.e., a view of T_6 which is a view of V_2) is the names and ranks of employees earning less than 50,000:

$$P_1 = \{(x_1, x_2) \mid (\exists y_1, y_2)(x_1, x_2, y_1, y_2) \in \mathit{Employee} \wedge (y_1 \leq 50,000)\}.$$

The original request T_6 and the actual answer P_1 are shown below.

T_6:

Name	*Rank*	*Salary*
Andy	*senior*	*43,000*
Cathy	*junior*	*48,000*
Herman	*senior*	*55,000*
Ziggy	*senior*	*67,000*

P_1:

Name	*Rank*
Andy	*senior*
Cathy	*junior*

Thus, the original request was reduced by removing all salaries and the names of employees earning more than 50,000. The actual answer is accompanied by the definition of P_1.

Note that, in general, several (possibly overlapping) partial views may be permissible. Since an attribute may be allowed in some views and not in other views, blanks may be scattered throughout the answer.

A similar refinement can be developed for transactions that modify the database. Let t be a modification transaction submitted by user u, that is denied because its view M does not have the property M/u. Again, the view inference

algorithm finds views of M that have property M/u. These views are *partial requests* which u is entitled to submit.

Finally, the view inference problem was also applied to determine whether a retrieval transaction is satisfiable (Procedure 1, steps 1–3). When the view R of a retrieval transaction t is *not* a view of empty views (i.e., the query is satisfiable), it is possible to find the views of R that are empty. These views are *constraints on the answer*, and can enrich the meaning of answers [12, 11].

View inferencing has been used repeatedly in the enforcement mechanism described in this section. The next two sections describe two alternative view inferencing methods.

3. An Algebraic Approach to View Inferencing

An algebraic solution to the view inference problem is to represent the definitions of the given views in special relations, called meta-relations, whose structure mirrors the actual relation. Standard algebraic operations (product, selection and projection) are extended to these meta-relations. When a query is presented to the database system, it is performed *both* on the actual relations, resulting in an answer, and on the meta-relations, resulting in definitions of views of the answer that inherit particular properties of the given views.

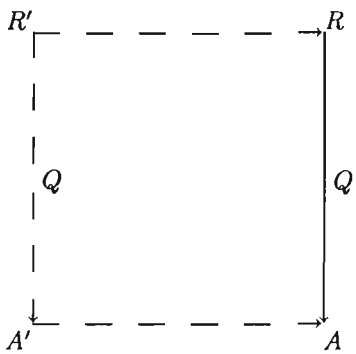

Figure 3: Meta-processing

This approach is illustrated by the commutative diagram shown in Figure 3. The horizontal lines describe the relationships between meta-relations and relations, and the vertical lines describe query processing and meta-processing. The solid line describes the standard relational model: the relation A is derived from the database R to answer a query Q. The dashed lines describe the extended model: the meta-relations R' define views of the database relations R that have a given property, and query processing is extended to manipulate also R', to yield the meta-relation A', that defines views of the answer A with the given property.

3.1. *Preliminaries*

This approach considers only views that are defined by *conjunctive relational calculus expressions* [16]. Using domain relational calculus, expressions from this family have the form:

$$\{x_1, \ldots, x_n \mid (\exists y_1, \ldots, y_m)\, \psi_1 \wedge \ldots \wedge \psi_k\},$$

where the ψs may be of two kinds:

1. **membership:** $(z_1, \ldots, z_p) \in R$, where R is a database relation (of arity p), and the zs are either xs or ys or constants.
2. **comparative:** $w_1 \, \theta \, w_2$, where w_1 is either an x or a y, w_2 is either an x or a y or a constant, and θ is a comparator (e.g., $<$, $\leq$, $>$, $\geq$, $=$, $\neq$).

In particular, each x and each y must appear at least once among the zs.

We refer to such views as *conjunctive views*. While this family is a strict subset of the relational calculus, it is a powerful subset. The family of conjunctive relational calculus expressions is precisely the family of relational algebra expressions with the operations *product*, *selection* and *projection* (where the selection expressions are conjunctive).

3.2. *Representation: Meta-Relations*

The representation of conjunctive views in meta-relations recalls the representation of QBE queries in skeleton tables [18].

For each relation R, a *meta-relation* R' is defined. The scheme of R' is identical to the scheme of R, except for an additional attribute called *Property*. Also, an auxiliary relation is defined: $Comparison = (X, Compare, Y)$. The meta-relations will be used to store membership formulas of views. Their tuples will be referred to as *meta-tuples*. Comparative formulas will be stored in the relation *Comparison*.

Consider a view

$$V = \{(x_1, \ldots, x_n) \mid (\exists y_1, \ldots, y_m) \, \psi_1 \wedge \ldots \wedge \psi_k\}.$$

A formula ψ of the kind $(z_1, \ldots, z_p) \in R$ is first modified so that the zs that are xs are suffixed with $*$, and the zs that are variables (i.e., xs or ys) that appear only once in the whole expression are replaced with $\sqcup$ (blank). Hence, each component of the modified formula is either a constant, a variable, or a blank, and each may be suffixed by $*$. This tuple is prefixed with the property of the view V, and is stored in R'. A formula ψ of the kind $w_1 \, \theta \, w_2$, where θ is not $=$, is transformed to the tuple (w_1, θ, w_2) and stored in the auxiliary relation *Comparison*. If θ is $=$, then all occurrences of w_1 in the other formulas are substituted with w_2. Finally, we assume that variable names are not shared among views.

As an example, recall the database and views from Section 2.2. The meta-relation representations of V_1, V_2 and V_3 are shown below (p denotes the property of the view).

1. Names and salaries of all employees:

$$V_1 = \{(x_1, x_2) \mid (\exists y_1, y_2)(x_1, y_1, x_2, y_2) \in Employee\}.$$

V_1 is represented with one meta-tuple:

$$(p, *, \sqcup, *, \sqcup) \in Employee'.$$

2. Names and ranks of employees earning less than 50,000:

$$V_2 = \{(x_1, x_2) \mid (\exists y_1, y_2)(x_1, x_2, y_1, y_2) \in \mathit{Employee} \wedge (y_1 \leq 50,000)\}.$$

V_2 is represented with two meta-tuples:

$$(p, *, *, w_1, \sqcup) \in \mathit{Employee}',$$
$$(w_1, \leq, 50,000) \in \mathit{Comparison}.$$

3. Junior employees earning more than 50,000:

$$V_3 = \{(x_1, x_2, x_3, x_4) \mid (x_1, x_2, x_3, x_4) \in \mathit{Employee} \wedge (x_2 = \text{“}\mathit{junior}\text{”}) \wedge (x_3 > 50,000)\}.$$

V_3 is represented with two meta-tuples:

$$(p, *, \mathit{junior}*, w_2*, *) \in \mathit{Employee}',$$
$$(w_2, >, 50,000) \in \mathit{Comparison}.$$

View V_4 (department managers who are not employees) is not a conjunctive view, and therefore cannot be expressed in this approach. As an example of a view that references more than one relation, consider

4. Names and ranks of employees working for Herman:

$$V_4 = \{(x_1, x_2) \mid (\exists y_1, y_2, y_3)(x_1, x_2, y_1, y_2) \in \mathit{Employee} \wedge (y_2, y_3) \in \mathit{Department} \wedge (y_3 = \text{“}\mathit{Herman}\text{”})\}.$$

V_4 is represented with two meta-tuples:

$$(p, *, *, \sqcup, w_3) \in \mathit{Employee}',$$
$$(p, w_3, \mathit{Herman}) \in \mathit{Department}'$$

3.3. *Inference: Meta-Processing*

Meta-relations are processed with three meta-operations: meta-product, meta-selection and meta-projection. The precise definition of these operations is given in [10]. These definitions were shown to be correct, in the sense that the meta-product defines views that would be obtained by applying the product to the views defined in the given meta-relations; the meta-selection defines views that would be obtained by applying the selection to the views defined in the given meta-relation; and the meta-projection defines views that would be obtained by applying the projection to the views defined in the given meta-relation.

Formally, let $\boldsymbol{D}$ be an instance of this database. Assume that R' and S' are meta-relations that define views of R and S, and let r and s be meta-tuples in R' and S', respectively. Let Q' be the result of a meta-operation, and let q be

a meta-tuple in Q'. Let $r(\boldsymbol{D})$, $s(\boldsymbol{D})$, and $q(\boldsymbol{D})$ denote, respectively, the relations defined by the meta-tuples r, s and q. Then:

meta-product $Q' = R' \times S'$:	$q(\boldsymbol{D}) = r(\boldsymbol{D}) \times s(\boldsymbol{D})$.
meta-selection $Q' = \sigma_\lambda R'$:	$q(\boldsymbol{D}) = \sigma_\lambda r(\boldsymbol{D})$.
meta-projection $Q' = \pi_{R-R_i} R'$:	$q(\boldsymbol{D}) = \pi_{R-R_i} r(\boldsymbol{D})$.

Second, it is simple to establish that the view properties discussed in Section 2 (i.e, R/u, M/u and $\perp$) are inherited by these meta-operations. Formally,

meta-product:	if $r(\boldsymbol{D})$ and $s(\boldsymbol{D})$ have a property, then $q(\boldsymbol{D})$ has this property.
meta-selection:	if $r(\boldsymbol{D})$ has a property, then $q(\boldsymbol{D})$ has this property.
meta-projection:	if $r(\boldsymbol{D})$ has a property, then $q(\boldsymbol{D})$ has this property.

These two results may be summarized as follows. Assume a database scheme $\mathcal{R}$ and views $\mathcal{V}$ with a property p. Let T be a transaction against this database. Let S be the relational algebra expression that implements T. Let S' be the relational algebra expression obtained from S by substituting every reference to R with a reference to R'. S operates on the actual database relations to yield the answer A. S' operates on the meta-relations to yield the meta-relation A' of views of A. Then, the views in A' possess the property p.

The result guarantees that the method for generating integrity constraints is *sound*, but it does not guarantee that it is *complete*. That is, this method generates views of the result that indeed have the property, but does not necessarily generate all such views. Yet, several simple refinements were developed, that generate additional desirable views [10].

3.4. *Example*

With the database instance of Figure 1, the four views of Section 3.2, and the *VP* table of Figure 2, consider again the transaction T_6 of Section 2.4 (Smith is requesting to list the names, ranks and salaries of employees earning more than 40,000). Processing this query in the database yielded the answer

Name	*Rank*	*Salary*
Andy	*senior*	*43,000*
Cathy	*junior*	*48,000*
Herman	*senior*	*55,000*
Ziggy	*senior*	*67,000*

Processing this query in the meta-database yields the meta-answer

Name	*Rank*	*Salary*
*	*	x

X	*Compare*	Y
x	$\leq$	50,000

This meta-answer describes a view of the answer, which is the names and ranks of employees earning less than 50,000.

4. A Logic Approach to View Inferencing

The view inference problem can also be treated in the framework of logic-based (deductive) databases. Within this framework relations and views are modeled, respectively, with *extensional* predicates (predicates defined by stored facts) and *intensional* predicates (predicates defined by rules) [15].

The solution we describe here is adapted from [2, 1], where the subject is the optimization of query processing by considering integrity constraints. Chakravarthy, et al., show how integrity constraints, which are assertions on the database, can be *compiled* into equivalent assertions, called residues, that are attached to individual extensional predicates. When a query is submitted, these residues are *reduced* to assertions that apply to the query. The query optimizer then uses these query residues to transform the original query into equivalent queries that can be processed faster in the database.

We observe that views with properties are also assertions on the database, and apply the same method to these assertions. The residues attached to each extensional predicate correspond to views of the extensional predicate that have the property, and the residues subsequently derived for each query correspond to views of the answer that have the property.

4.1. *Preliminaries*

An *atom* is a predicate (a Boolean-valued function) with a list of arguments (variables or constants). A *rule* has the form $Q \leftarrow P_1 \wedge \ldots \wedge P_n$ where Q and each P_i are atoms. Q is the *head* of the rule, and $P_1 \wedge \ldots \wedge P_n$ is the *body* of the rule. Thus, rules define new predicates in terms of existing predicates. Note that a rule may be without a body (i.e., $n = 0$). A rule without a body and without any variables is a *fact*. Variables appearing only in the body of a rule may be regarded as quantified existentially within the body, while other variables are quantified universally over the entire rule. Thus, a rule without a body is quantified universally.

The relational model defined in Section 2.1 can now be stated in terms of logic. A *relation* is a predicate, with an associated set of stored facts. The predicate is defined to be *true* over the associated facts, and *false* otherwise. A *Comparator* is a predicate, whose associated set of true facts is assumed to be known (and treated as if it is stored). A *view* is a predicate defined by a rule. We shall assume that rules are non-recursive, and that only relation predicates and comparators are used in the bodies of rules (i.e., view predicates are not used to define further view predicates)[4].

The previous database is implemented with two predicates: $Employee = (Name, Rank, Salary, Department)$, and $Department = (Dname, Manager)$. The views V_1–V_4 from Section 3.2 are then defined with the following rules:

1. Names and salaries of all employees:

$$V_1(x_1, x_2) \leftarrow Employee(x_1, y_1, x_2, y_2).$$

[4] The latter assumption is not restrictive, as every non-recursive rule can be transformed so that its body includes only relation predicates and comparators.

2. Names and ranks of employees earning less than 50,000:

$$V_2(x_1, x_2) \leftarrow Employee(x_1, x_2, y_1, y_2) \wedge (y_1 \leq 50,000).$$

3. Junior employees earning more than 50,000:

$$V_3(x_1, x_2, x_3, x_4) \leftarrow Employee(x_1, x_2, x_3, x_4) \wedge (x_2 = \text{"}junior\text{"}) \wedge (x_3 > 50,000).$$

4. Names and ranks of employees working for Herman:

$$V_4(x_1, x_2) \leftarrow Employee(x_1, x_2, y_1, y_2) \wedge Department(y_2, y_3) \wedge (y_3 = \text{"}Herman\text{"}).$$

The family of views expressible with rules of this type is identical to the family of conjunctive views defined in Section 3.1.

4.2. *Representation: Relation Residues*

A *substitution* is a consistent mapping of variables in a given formula to other variables or constants. A substitution that maps a_i to b_i $(i = 1, \ldots, n)$ is denoted $\{b_1/a_1, \ldots, b_n/a_n\}$. The result of applying a substitution θ to a formula F is denoted F_θ.

A formula F *subsumes* a formula G, if there is a substitution θ such that F_θ is a subformula of G. For example, consider the formulas

$$F : V(x_1) \leftarrow Employee(x_1, y_1, y_2, y_3),$$
$$G : V(x_1) \leftarrow Employee(x_1, y_1, y_2, y_3) \wedge (y_1 = \text{"}junior\text{"}) \wedge Department(y_3, y_4) \wedge (y_4 = \text{"}Herman\text{"}).$$

F (the names of employees) subsumes G (the names of junior employees working for Herman), with the substitution: $\{junior/y_1\}$.

A formula F *partially subsumes* a formula G, if there is a substitution θ such that a subformula of F_θ is a subformula of G. The part of F_θ which is not a subformula of G is called the *residue* of F over G. For example, consider the formula

$$H : V(x_1) \leftarrow Employee(x_1, y_1, y_2, y_3) \wedge (y_2 > 40,000).$$

H (the names of employees earning more than 40,000) partially subsumes G, with the substitution $\{junior/y_1\}$. The residue of H over G is

$$K : (y_2 > 40,000).$$

The residue of F over G expresses F relative to G. Thus, if V is a view with a property, and R is a relation, then the residue of V over R is the view of R that has the property. For example, assume that user u has permission to access the names of junior employees. The residue of this view over the entire employee relation would be "junior only".

An algorithm for the computation of residues is given in [2]. This algorithm is applied here to transform the property views to "property residues" attached to individual relations. Formally, let $R_1, \ldots, R_n$ be the database relations, and let $V_1, \ldots, V_m$ be the property views. The property views are transformed to the residues $W_{i,1}, \ldots, W_{i,k_i}$ that are attached to R_i $(i = 1, \ldots, n)$. Note that this transformation is performed only once, and can be regarded as a "compilation" of the property views.

As an example, the views V_1–V_4 would be compiled into the following relation residues:

- $Employee(u_1, u_2, u_3, u_4)$:
  1. $V_1(u_1, u_3) \leftarrow$.
  2. $V_2(u_1, u_2) \leftarrow (u_3 \leq 50,000)$.
  3. $V_3(u_1, u_2, u_3, u_4) \leftarrow (u_2 = \text{"}junior\text{"}) \wedge (u_3 > 50,000)$.
  4. $V_4(u_1, u_2) \leftarrow Department(u_4, u_5) \wedge (u_5 = \text{"}Herman\text{"})$.
- $Department(u_5, u_6)$:
  1. $V_4(u_1, u_2) \leftarrow Employee(u_1, u_2, u_3, u_5) \wedge (u_6 = \text{"}Herman\text{"})$.

4.3. *Inference: Query Residues*

Assume now a query $Q \leftarrow P_1 \wedge \ldots \wedge P_t$. Recall that each atom P_i is either a relation predicate or a comparator. Let P be an atom from the body of the query which is a relation predicate, and let $(x_1, \ldots, x_r)$ be its argument list (the xs are either variables or constants).

Consider now the same predicate P in the compiled schema. Let $(y_1, \ldots, y_r)$ be the argument list of P in the compiled schema (the ys are variables), and let $W_1, \ldots, W_k$ be the residues of P in the compiled schema. The substitution $\{x_1/y_1, \ldots, x_r/y_r\}$ in this set of residues generates a set of residues for the query atom P. Each of these residues expresses a view of P with a particular property. The *disjunction* of residues with a specific property, gives a single view of P with this property. This process is repeated for every such query atom P, and the *conjunction* of the views thus obtained provides a single view of the query with this property.

4.4. *Example*

With the compiled schema shown in Section 3.4, and the *VP* table of Figure 2, consider again the transaction T_6 of Section 2.4 (Smith is requesting to list the names, ranks and salaries of employees earning more than 40,000):

$$T_6(x_1, x_2, x_3) \leftarrow Employee(x_1, x_2, x_3, y_1) \wedge (x_3 > 40,000).$$

This query has a single atom which is a query predicate, *Employee*, and in the compiled schema, only the second residue of *Employee* has the property $R/Smith$. This residue reduces to the following query residue:

$$V_2(x_1, x_2) \leftarrow (x_3 \leq 50,000).$$

This rule describes the view of T_6 which may be delivered to Smith. Assuming that T_6 had been computed, the actual answer would be:

$$V_2(x_1, x_2) \leftarrow T_6(x_1, x_2, x_3) \wedge (x_3 \leq 50,000).$$

5. Interaction between Constraints and Permissions

Integrity constraints reveal information. As an extreme example, assume constraints of the kind "if the employee is Andy, then the salary is 43,000". Even without any access permissions to the relation *Employee*, users can apply the definitions of such constraints, to infer information that has not been made available to them. When combined with information that *has* been made available, even less explicit constraints can be used to breach security.

Consider the relation *Employee* and the constraint view "junior employees earning more than 50,000" (i.e., junior employees earn less than 50,000). Assume that u has permission to retrieve the names and ranks of the employees in the strip department, but not their salaries. Given this constraint, u can infer that Calvin and Cathy earn less than 50,000. Thus, u obtains information about an attribute that has not been made available to u.

Consider again the relation *Employee* and the same constraint. Assume that u has the same retrieval permissions, and also permission to modify the ranks. Given this constraint, u can modify the rank of every senior employee in the strip department to junior, and infer from the acceptance or rejection of each update whether the particular employee earns more than or less than 50,000. In this case, that Andy earns less than 50,000 and Herman and Ziggy earn more than 50,000. Again, u obtains information about an attribute that has not been made available to u.

In general, when a constraint is satisfied (i.e., its view is empty), one can infer that a tuple that satisfies a subformula of the constraint, would not satisfy the remainder of the constraint. This was exploited by u in both examples. In the first example, u knew that Calvin and Cathy satisfy the position requirement, and therefore concluded that they do not satisfy the salary requirement. In the second example, u updated the database to force Andy, Herman and Ziggy to satisfy the position requirement, and then discovered whether or not they satisfy the salary requirement.

The interaction between integrity constraints and access permissions can be formalized with the logic-based approach described in Section 4. Briefly, the *residue* of a constraint over a permitted view describes the constraint applicable to this view. We shall show how to further reduce the residue; the *reduced residue* will describe the *information provided by the constraint, which has not been made available to the user.*

Let U be a view permitted to user u (u has either retrieve or modify permissions to U), and let view V be an integrity constraint. The *residue* of V over U describes the integrity constraint applicable to the accessible information. A subformula of this residue is *redundant* if either

- U can be restricted (by selecting values of retrievable attributes), so that it satisfies the subformula.

- U can be updated (by changing values of modifiable attributes), so that it satisfies the subformula.

The redundant formulas are the part of the residue that is ***subsumed*** by the information made available to the user. The remaining part, the *reduced residue*, expresses information that is ***additional*** to the information made available to the user.

We demonstrate the process with the two previous examples.

In the first example we have

$$V(x_1, x_2, x_3, x_4) \leftarrow \mathit{Employee}(x_1, x_2, x_3, x_4) \wedge (x_2 = \text{"}\mathit{junior}\text{"}) \wedge (x_3 > 50,000),$$
$$U(x_1, x_2) \leftarrow \mathit{Employee}(x_1, x_2, y_1, y_2) \wedge (y_2 = \text{"}\mathit{strip}\text{"}).$$

Using the substitution $\{y_1/x_3, \text{"}\mathit{strip}\text{"}/x_4\}$, the residue of V over U is

$$R : (x_2 = \text{"}\mathit{junior}\text{"}) \wedge (y_1 > 50,000).$$

R expresses a constraint that applies to the view U: among the employees in the strip department, there are no junior employees earning over 50,000. We now attempt to reduce this residue, by satisfying some of its subformulas. Because U retrieves the attribute *Position* (variable x_2), it is possible to satisfy the first subformula by restricting U to employees who are juniors. Define

$$U'(x_1) \leftarrow U(x_1, x_2) \wedge (x_2 = \text{"}\mathit{junior}\text{"}).$$

The reduced residue of V over U' is only

$$R' : (y_1 > 50,000).$$

R' expresses a constraint on view U': among the junior employees in the strip department, there are no employees earning over 50,000. Clearly, R' betrays information on salaries.

In the second example we also have

$$V(x_1, x_2, x_3, x_4) \leftarrow \mathit{Employee}(x_1, x_2, x_3, x_4) \wedge (x_2 = \text{"}\mathit{junior}\text{"}) \wedge (x_3 > 50,000),$$
$$U(x_1, x_2) \leftarrow \mathit{Employee}(x_1, x_2, y_1, y_2) \wedge (y_2 = \text{"}\mathit{strip}\text{"}).$$

Again, using the same substitution, the residue of V over U is:

$$R : (x_2 = \text{"}\mathit{junior}\text{"}) \wedge (y_1 > 50,000).$$

Again, we try to reduce this residue by satisfying some of its subformulas. Because U permits modification of the attribute *Position* (variable x_2), it is possible to satisfy the first subformula of this residue by changing all senior positions to junior (i.e., we *update* U, rather than *restrict* it). Define

$$U''(x_1) \leftarrow U(x_1, x_2 \leftarrow \text{"}\mathit{junior}\text{"}).$$

The reduced residue of V over U'' is only

$$R' : (y_1 > 50,000).$$

R' no longer expresses a constraint (it may not be satisfied by some employees). It is a *decidable* condition for view U'': among the employees in the strip department, it is possible to decide who earns more than 50,000. Clearly, R' betrays information on salaries. Since U'' is not as restricted as U', the breach of security is even greater than in the first example.

The amount of information revealed in this manner depends on the complexity of the reduced residue. For example, residues of the kind (*variable* $\neq$ *constant*) are more informative than residues of the kind (*variable* $>$ *constant*). In the first example, the residue (*salary* $\neq 50,000$) would have provided the exact salary of Calvin and Cathy. Also, short residues are more informative than long residues. Assume that u has only access to employees names and departments. The residue (*position* $=$ "*junior*") $\wedge$ (*salary* $> 50,000$) reveals only that the given employees cannot be junior and earn more than 50,000. The residue (*salary* $> 50,000$) reveals that the given employees earn less than 50,000. This explains our attempts at reducing the residues as much as possible.

Two extreme cases are when the reduced residue is *null*, or when it is *invariable* (identical to the original constraint). In the first case, there is complete subsumption: the information made available to the user subsumes the constraint entirely. This is a case where absolutely no additional information is revealed by the constraint. In the second case, there is not even partial subsumption: the constraint is entirely "external" to the information made available to the user. In this case also no additional information is revealed by the constraint. More accurately, the information revealed by the constraint itself does not interact in any exploitable way with the information made available to the user.

To prevent any negative effect of such residues two policies may be adopted. One policy guarantees null residues, the other guarantees null or invariable residues.

Since integrity constraints provide information, they should be classified. A constraint V should be available to user u, only if the residue of V over the views permitted to u is null. A simple policy is to allow u to know about V only if u has unqualified access to all the attributes of V. This policy is sufficient to guarantee null residues, but is not necessary; i.e., more limited access permissions may guarantee null residues as well (though at a price of a more complicated policy). While knowledge of integrity constraints is useful to the users of a database, it is hard to argue that users need to know about constraints that are not internal to their views!

If it is impossible or undesirable to control access to constraints, then views and constraints must be carefully defined so that a constraint is either subsumed in a view, or is entirely external to it. A simple policy is to consider the attribute sets of views and constraints, and to insist that the attributes of each constraint are either entirely contained in the attributes of the views available to each user, or entirely external to them. Again, this policy is sufficient to guarantee that residues are either null or invariable, but is not necessary.

Of course, in-between solutions may be designed, that permit residues that are neither null nor invariable, avoiding only residues that are considered too "short".

6. A Broader Concept of Integrity

A database is a model of the real world, and its designers and administrators invest continuous efforts to ensure that this model would approximate the real world as accurately as possible. Still, if the database if large enough, it is almost certain to be an imperfect model. Consequently, the answers issued by the database system in response to queries are imperfect as well.

The question of the *integrity* of the information that is received from a database (or, for that matter, from any other source) is usually the primary concern of the user. Usually, this question has two parts: (1) Is the information sound? (2) Is the information complete? For example, when one requests the personnel database to list all the employees who work for Herman, one is concerned with the *soundness* of the information received (does each person named indeed work for Herman?), as well as with its *completeness* (are there any other employees who work for Herman that were not listed?).

In other words, answers have integrity, if they contain *the whole truth* (completeness) and *nothing but the truth* (soundness).

The issue of soundness is addressed only in part in database systems, through the mechanism of integrity constraints (e.g., as defined in our own model in Section 2). However, such constraints *enhance* the soundness of a database (and hence the soundness of its answers), but they cannot *ensure* it. For example, it is not possible to express constraints that ensure that the information about each junior employee is sound.

The issue of completeness is usually not addressed in database systems. The concept of information completeness is related to the Closed World Assumption (CWA) [13]. Under this assumption, a database contains all occurrences of the data it attempts to model. More accurately, the CWA states that if a fact is not included in the database (and cannot be inferred from it), then it is false. In practice, however, this assumption is not realistic, as most databases include at least some information that is possibly incomplete. In other words, in most cases this assumption can only be made on some *subsets* of the database.

Recently, a new model for integrity was developed [10], that addresses these problems. New kinds of integrity constraints, called *soundness constraints* and *completeness constraints* were introduced[5]. A soundness constraint asserts that a particular subset of the database is guaranteed to be sound, and a completeness constraint asserts that a particular subset of the database is guaranteed to be complete. Together, soundness and completeness constraints are assertions on the integrity of the database.

In addition, the new integrity model is designed to *certify* the integrity of answers. For each answer issued in response to a query, it determines (1) whether the answer is *sound* (the information is ensured to be correct), or only *partially sound* (specified portions are ensured to be correct); and (2) whether the answer is *complete* (the information is ensured to include all the real world occurrences), or only *partially complete* (specified portions are ensured to include all the real world

[5] In [10] the term *validity* is used for soundness.

occurrences). In effect, the database is *qualifying* its answers, by the scope of its knowledge (as defined by the soundness and completeness constraints).

Assume the example database satisfies two integrity constraints that guarantee sound information about junior employees, and complete information about employees who work for Herman. The answer to a query to retrieve the employees who earn over 35,000 is certified to be sound with regard to the junior employees, and complete with regard to the employees who work for Herman. Clearly, answers that are accompanied by such "certification of quality" are more meaningful.

Formally, soundness and completeness constraints are defined as views. Let $\mathcal{R}$ be the schema of the database, and let $\boldsymbol{D}$ be the instance of the database. The portion of the real world modeled by this database can be viewed as a hypothetical instance $\boldsymbol{W}$ of $\mathcal{R}$. Let S and C be two views of $\mathcal{R}$. The view S is a ***soundness constraint* which is satisfied by $\boldsymbol{D}$, if $S(\boldsymbol{D}) \subseteq S(\boldsymbol{W})$**. The view C is a *completeness constraint* which is satisfied by $\boldsymbol{D}$, if $C(\boldsymbol{D}) \supseteq C(\boldsymbol{W})$. Hence, soundness and completeness are treated with perfect duality.

If both the database and the real world environment it models are *static*, then a database with integrity will always retain its integrity. In general, however, real-world environments are subject to changes, and databases must change accordingly. Such changes might *violate* the integrity of databases. This integrity may be violated in two ways. First, through changes to the database that do not reflect actual changes in the real world; for example, an employee who works for Herman is deleted from the database, thus violating the aforementioned completeness constraint, or the salary of a junior employee is changed in the database, thus violating the aforementioned soundness constraint. Second, through changes in the real world that have not been incorporated into the database; for example, a new employee starts to work for Herman, or the salary of an junior employee is raised. Both kinds of violations can only be avoided by human supervision: the database administrator must monitor closely the views of the database that are governed by soundness and completeness constraints, updating them to reflect all the changes in the real world, and nothing else.

Assume now soundness and completeness constraints with the following restrictions: (1) views of soundness constraints do not include any true tuples, and (2) views of completeness constraints do not include any false tuples. For example, assume that it is mandatory that senior employees earn a salary of more than 40,000; then the view of the senior employees who earn less than 40,000 is guaranteed not to include any true tuples. Similarly, assume that it is mandatory that every programmer knows Pascal; then the view of the programmers who know Pascal is guaranteed not to include any false tuples[6]. The integrity of such views can be monitored without human supervision: any update that inserts a tuple into a soundness view of this kind (clearly, it is impossible to delete from such views), or any update that deletes an existing tuple from a completeness view of this kind (clearly, it is impossible to insert tuples into such views) can be rejected automatically. Clearly, soundness constraints of this kind correspond to "traditional" integrity constraints. Thus, the "traditional" integrity model is *subsumed* in the soundness component of the new integrity model.

[6] We assume that only values from the domain of programmers are used.

This broader concept of integrity can be incorporated easily into the security and integrity model described in Section 2. Soundness and completeness constraints would be entered into the *VP* table as views with properties S and C, respectively, and the screening procedures would be extended to apply the view inference algorithm to these new properties as well. The properties of soundness and completeness can also be refined to include additional details, such as the *party* guaranteeing the maintenance of the properties, or the *time* at which these properties have last been verified. These complex properties would then yield answers such as: "according to Smith the following view is valid as of 15-March-91 17:15 ...".

The new integrity constraints further demonstrate the tight relationship between integrity constraints and access permissions. As mentioned earlier, except for particular subtypes that can be maintained by the system, the management of the new integrity constraints requires, in general, human supervision. Specifically, permission to modify soundness and completeness views should be granted only to the warrantors of these constraints.

7. Conclusion

We described a model for implementing protective restrictions in relational databases. The restrictions protect the database from violations of both security and integrity. All protective restrictions are stated in terms of database views. These protective restrictions are treated as *knowledge*, from which transaction screening procedures *infer* the restrictions that apply to individual transactions. A transaction may be allowed or denied in its entirety, or specific non-violating subtransactions may be identified. This process is an application of the view inference problem, to which we offered two alternative solutions. We showed how knowledge of integrity constraints could be combined with information available through access permissions to infer additional information, and have provided designs of constraints and permissions that prevent such security breaches. Finally, we showed how the model can accommodate a broader concept of integrity.

Comparing the two alternative solutions to the view inference problem, we note that both deal with views of the same general class. The algebraic method will probably prove to be more efficient; the logic method employs refutation to detect subsumption in the process of residues computation. On the other hand, the logic method generates the complete set of property views. Incompleteness of the algebraic method is mostly due to the fact that the method is concerned only with property views that are expressible in the attributes of the transaction view; the logic method may generate property views that involve predicates and attributes that were not specified in the transaction view.

Whichever method is adopted, there is strong evidence that having a database subsystem that implements a view inferencing mechanism will be very beneficial. In this paper we discussed the applications of this mechanism to detect violations of security (or non-violating subtransactions), unsatisfiability of transactions (or constraints on the answer), soundness of answers (or partial soundness), and completeness of answers (or partial completeness). Yet, other applications may be conceived. For example, assume that a database system saves in memory ("caches")

the results of the last n queries. The view inference mechanism can then be used to determine whether a new query is computable from these saved results. This strategy may improve performance in applications where new queries are often related to recently processed queries.

The views we assumed (in the *VP* table and in transactions) are views that *assemble* information extracted from across the database into a single table. In practice, integrity constraints, access permissions, and transactions often involve *aggregates* of information. For example, an integrity constraint may state that the *total* of all employee salaries should not exceed a certain amount, access permission may be given only to the *average* salary, and a transaction may inquire about the *highest* paid employee. Work is underway to extend the model to allow views that incorporate aggregate functions.

An issue related to the security and integrity model presented here is that of *consistency* of the protection restrictions, and their *implications*.

Consistency of integrity constraints is relatively well-understood. As a trivial example, the constraints "all employees earn over 35,000", "each department must have at least 3 employees", and "the salary budget of each department should not exceed 100,000" are inconsistent. Detection of inconsistencies is relatively simple (though not computationally inexpensive), when the constraints are stated in logic. Still, even when a set of constraints is consistent, it may be instructive to find out some of its implications. For example, adding the constraint "junior employees earn under 50,000" to a set of constraints that includes "employees of the strip department earn over 50,000", does not introduce a contradiction, but would imply the constraint "the strip department may not have junior employees".

Essentially, access permissions are "positive" statements that cannot be inconsistent. If, however, we were to introduce "negative" access permissions, i.e., statements indicating that particular users are *not* to be provided access to specific information, then the issue of consistency of a set of permissions would arise (and would be quite similar to that of integrity constraints). In practice, however, the positive permission model is usually employed with the *intention* of implementing negative permissions; that is, the grantor permits access to certain information, assuming all other information becomes inaccessible. In such situations, it would be instructive to find out the implications of the explicit permissions, and verify that they do not conflict with the implicit intentions.

When access permissions and integrity constraints are considered together, the issue of implication is even more intriguing. This issue was addressed in part in Section 5, where the interaction between *individual* constraints and permitted views was considered.

References

[1] U. S. Chakravarthy, J. Grant, and J. Minker, "Logic-based approach to semantic query optimization", *ACM Transactions on Database Systems* **15**(2), (June 1990), 162–207.

[2] U. S. Chakravarthy, J. Grant, and J. Minker, "Foundations of semantic query optimization for deductive databases", pp. 243–273 in *Foundations of Deductive Databases and Logic Programming* (J. Minker, ed.), Morgan Kaufmann, Los Altos, California, 1988.

[3] U. Dayal and P. Bernstein, "On the updatability of relational views", in *Proceedings of the Fourth International Conference on Very Larg Data Bases* (West Berlin, Germany, September 13–15), pages 368–377, ACM, New York, New York, 1978.

[4] D. E. Denning, *Cryptography and Data Security*, Addison Wesley, Reading, Massachusetts, 1982.

[5] M. Gasser, *Building a Secure Computer System*, Van Nostrand Reinhold, New York, New York, 1988.

[6] P. P. Griffiths and B. W. Wade, "An authorization mechanism for a relational database system", *ACM Transactions on Database Systems* **1** (3), (September 1976), 242–255.

[7] *SunINGRES Manual Set*, Sun Microsystems, Mountain View, California, Release 5.0 (Part Number 800-1644-01), 1987.

[8] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, 1983.

[9] A. Motro, "An access authorization model for relational databases based on algebraic manipulation of view definitions", in *Proceedings of the IEEE Computer Society Fifth International Conference on Data Engineering* (Los Angeles, California, February 6–10), pages 339–347, IEEE Computer Society, Washington, DC, 1989.

[10] A. Motro, "Integrity = validity + completeness", *ACM Transactions on Database Systems*, **14** (4), (December 1989), 480–502.

[11] A. Motro, "Intensional answers to database queries", *IEEE Transactions on Knowledge and Data Engineering*, to appear in 1992.

[12] A. Motro, "Using integrity constraints to provide intensional responses to relational queries", in *Proceedings of the Fifteenth International Conference on Very Large Data Bases* (Amsterdam, The Netherlands, August 22–25), pages 237–246, Morgan Kaufmann, Los Altos, California, 1989.

[13] R. Reiter, "On closed world data bases", pp. 55–76 in *Logic and Databases*, Plenum Press, New York, New York, 1978.

[14] M. Stonebraker and E. Wong, "Access control in a relational database management system by query modification", in *Proceedings of ACM Annual Conference* (San Diego, California, November), pages 180–186, ACM, New York, New York, 1974.

[15] J. D. Ullman, *Database and Knowledge-Base Systems, Volume I*, Computer Science Press, Rockville, Maryland, 1988.

[16] J. D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, Maryland, 1982.

[17] P. Valduriez and G. Gardarin, *Analysis and Comparison of Relational Database Systems*, Addison-Wesley, Reading, Massachusetts, 1989.

[18] M. Zloof, "Query-by-Example: a database language", *IBM Systems Journal* **16** (4), (December 1977), 324–343.