Extending Query Rewriting Techniques for Fine-Grained Access Control

Shariq Rizvi *
University of California, Berkeley
rizvi@cs.berkeley.edu

Alberto Mendelzon *
University of Toronto
mendel@cs.toronto.edu

S. Sudarshan
Indian Institute of Technology, Bombay
sudarsha@cse.iitb.ac.in

Prasan Roy *
IBM India Research Laboratory
prasanr@in.ibm.com

ABSTRACT

Current day database applications, with large numbers of users, require fine-grained access control mechanisms, at the level of individual tuples, not just entire relations/views, to control which parts of the data can be accessed by each user. Fine-grained access control is often enforced in the application code, which has numerous drawbacks; these can be avoided by specifying/enforcing access control at the database level. We present a novel fine-grained access control model based on authorization views that allows "authorization-transparent" querying; that is, user queries can be phrased in terms of the database relations, and are valid if they can be answered using only the information contained in these authorization views. We extend earlier work on authorization-transparent querying by introducing a new notion of validity, conditional validity. We give a powerful set of inference rules to check for query validity. We demonstrate the practicality of our techniques by describing how an existing query optimizer can be extended to perform access control checks by incorporating these inference rules.

1. INTRODUCTION

Access control is an integral part of databases and information systems.

Granularity of access control refers to the size of individual data items which can be authorized to users. There are many scenarios that demand fine-grained access control:

- For an academic institution's database that stores information about student grades, it may be desired to allow students to see only their own grades. On the other hand, a professor should get access to all grades for a course she has taught.
- For a bank, a customer should be able to query her account balance, and no one else's balance. At the same time, a teller should have read access to balances of all accounts but not the addresses of customers corresponding to these balances.
- A teller should be allowed to see the balance of any account by providing the account-id but not the balances of all accounts together.

*Work done while at IIT Bombay

SIGMOD 2004 June 13-18, 2004, Paris, France.

In all the above cases, authorization is required at a very fine-grained level, such as at the level of individual tuples. Also, as in the last example, there can be a policy that defines *how* an access should be made apart from *what* data can be accessed.

Currently, authorization mechanisms in SQL permit access control at the level of complete tables or columns, or on views. There is no direct way to specify fine-grained authorization to control which tuples can be accessed by which users. In theory, fine-grained access control at the level of individual tuples can be achieved by creating an access control list for each tuple. However this approach is not scalable, and would be totally impractical in systems with millions of tuples, and thousands or millions of users, since it would require millions of access control specifications to be provided (manually) by the administrator. It is also possible to create views for specific users, which allow those users access to only selected tuples of a table, but again this approach is not scalable with large numbers of users.

Current generation information systems therefore typically bypass database access control facilities, and embed access control in the application program used to access the database. Although widely used, this approach has several disadvantages:

- Access control has to be checked at each user-interface. This increases the overall code size. Any change in the access control policy requires changing a large amount of code.
- All security policies have to be implemented into each of the applications built on top of this data (e.g. OLTP and decision support applications using the same underlying data).
- Given the large size of application code, it is possible to overlook loopholes that can be exploited to break through the security policies, e.g. improperly designed servlets. Also, it is easy for application programmers to create trap-doors with malicious intent, since it is impossible to check every line of code in a very large application.

For the above reasons, fine-grained access control should ideally be specified and enforced at the database level.

In this paper, we present a security model in which fine-grained authorization policies are defined and enforced in the database. This makes sure that the same policies hold, irrespective of how the data is accessed - through a report-writing tool, a query, or an application program.

The key features of our model are as follows:

1. Access control is specified using **authorization views** (Section 2). An authorization view can be a traditional relational view or a parameterized view. A parameterized authorization view is an SQL view definition which makes use of parameters like *user-id*, *time*, *user-location* etc. The following parameterized authorization view

create authorization view MyGrades **as**
select * **from** Grades **where** student-id = $user-id

lets the user see all tuples in the *Grades* relation where the *student-id* matches her user-id (parameters, such as user-id, are denoted by a $ prefix). Parameterized views provide an efficient and powerful way of expressing fine-grained authorization policies. As views can project out specific columns in addition to selecting rows, this framework allows fine-grained authorization at the cell-level.

We also provide a special form of parameterized views, which we call *access pattern views*, which allows specification of authorizations such as "a teller can see the account information of any one customer at a time, by providing her customer-id".

The model works within the basic SQL framework and does not require the DBA to encode policies using a separate rule language.

2. We allow queries to be written in an **authorization-transparent** manner, that is, queries can be written against the database relations without having to refer to the authorization views.

   Given a user query (phrased in terms of database relations or views), our system checks if the query is **valid**, that is, it can be answered using the information available in the authorization views that are accessible to the user. If found to be valid, the query is allowed to execute *as originally specified*, without any modification, otherwise it is rejected.

   An obvious way to enforce access control using authorization views is to allow queries to be written only against these views, not against the original database relations. However, since different users (or classes of users) may have different authorization views, this would require application programmers to code interfaces differently for each user (or class of users), increasing the cost and complexity of application development.

   Another alternative approach is to allow queries to be written against database relations, but to modify the query by replacing the database relations by the view of the relation that is available to the user. For example, the Virtual Private Database (VPD) feature of Oracle's database server [1] implements fine-grained access control using query modification, by adding extra predicates to where clause of the given query. However, as we describe in Section 3.3, query modification approaches have inherent undesirable side-effects, such as giving erroneous answers to certain queries, instead of rejecting them as unauthorized. In addition, modifying the query can result in the query having very different execution characteristics; for instance the modified query may take much longer to execute.

   We present an authorization model, which we call the Truman model, to understand the logical effect of the VPD and related query modification models (Section 3). We point out its drawbacks, and then propose our alternative model, the Non-Truman model, which avoids these drawbacks (Section 4).

   Authorization transparent models have been previously proposed by Motro [20] and by Rosenthal *et al*. [24, 22, 23]. Our model differs from these in several respects; we outline the differences in Section 7.

3. As a first approximation, a user query q can be answered using the information contained in the authorization views available to the user if there is a query q' using only the authorization views that is equivalent to q, i.e., the two queries give the same result on all database states. We categorize such queries q as **unconditionally valid**.

   The problem of rewriting a query using a set of available relational views [15] has received tremendous attention. It has been studied in the context of finding an efficient query execution plan by rephrasing a query using the set of available materialized views, in data integration systems, and for supporting the separation of logical and physical views of data. Our access control model leverages off these techniques.

   The idea that a query is valid (authorized) if it can be rewritten in terms of authorized views was proposed earlier by Motro [20] and by Rosenthal *et al*. [24, 22].

4. We show that certain queries can be answered using the information contained in a set of authorization views, even if they cannot be rewritten using the views. Unconditional validity of q requires that q and q' give the same results on all database states. The key idea in going beyond unconditional validity is that information in the authorization views available to a user rules out many database states, and we need not require q and q' to give the same result on such states. On the other hand, as we show later (Example 4.3), requiring q and q' to give the same answer only in the current database state is too weak a requirement, and can leak unauthorized information.

5. Our next contribution is therefore to exactly characterize the class of queries, which we call **conditionally valid** queries, that can be answered using the information contained in a set of views in a given database state (Section 4.3). The idea of conditional validity is novel to this paper.

6. We give a set of powerful inference rules which can be used to infer the unconditional and conditional validity of queries (Section 5). We also describe how to efficiently check the validity of a query by incorporating these rules into a query optimizer (Section 5.6); for concreteness, we describe how to incorporate these inference rules into the Volcano query optimizer [13].

   The idea that validity can be checked with help from a query optimizer is present in the work by Motro and by Rosenthal *et al.* cited above; however, they do not present any formal inference rules. Motro [20] also presents an alternative way of validity checking, for the special case of conjunctive queries and views, which we outline in more detail in Section 7.

The novel contributions of this paper include our model of authorization based on parameterized/access pattern authorization views, the notion of conditional validity, the inference rules for validity, and the techniques for incorporating the inference rules into an optimizer.

The rest of the paper is organized as follows. Section 2 describes (parameterized) authorization views. In Section 3, we describe the Truman and VPD models, and outline the pitfalls of the query modification approach. Section 4 presents our Non-Truman model, including definitions of of our notions of unconditional and conditional validity of queries. Section 5 describes a powerful set of inference rules for conditional and unconditional validity. Section 5.6 describes how to integrate our inference rules into a query

optimizer. Section 6 presents extensions of our basic scheme. Section 7 describes related work in access control. Section 8 concludes the paper and gives directions for future work.

2. AUTHORIZATION VIEWS

Traditionally, normal relational views have been used for access control. In such a framework, the DBA creates different views for each user. This is impractical when the database has thousands of users and the DBA has to encode the same policies for each one of them. Also, with a slight change of the authorization policy, a large number of views will be affected. In contrast, parameterized views provide a *rule-based* framework, where one view definition applies across several users. Hence, some kinds of policies can be more easily expressed using parameterized views.

A parameterized view is like a normal relational view, but with parameters like *user-id*, *time* and *user-location* appearing in its definition. The values of these parameters vary across different users and different accesses. Hence, the same parameterized view can give different results on the same database, depending on the execution context.

Consider a database with the following relations:
Students(student-id, name, type)
Courses(course-id, name)
Registered(student-id, course-id)
Grades(student-id, course-id, grade)
Let us assume integrity constraints that require each *student-id* and *course-id* value in the tables *Registered* and *Grades* to appear in the *Students* and *Courses* tables respectively. We use this schema as a running example throughout the paper. The following parameterized view stands for an authorization policy that allows a student to see grade information for all courses she has registered for (including the grades of other students).

create authorization view Co-studentGrades **as**
select Grades.*
from Grades, Registered
where Registered.student-id = $user-id
and Grades.course-id = Registered.course-id

The user-id $user-id appears as a parameter in this view definition. In general, the view can be a function of other parameters like time and user's location. For example, it may be desired to restrict an authorization for a user from a specific IP address to only a particular time of the day. Authorization views can also be aggregate views (if the DBA wants to allow a user to see only the average grades for all courses).

Given a particular access to the database (by a particular user), the parameters would be fixed, and we can replace all parameters in the authorization views by the actual values. We call the resultant set of authorization views the *instantiated authorization views*. These define exactly what information is visible (authorized) to the query; the validity of the query is thus tested using the instantiated authorization views.

To handle complicated authorization scenarios, it is possible to allow the DBA to define a procedure that takes as arguments the values of the parameters (of a particular access), and returns a set of instantiated authorization views, instead of using parameterized authorization views. As a result, the set of instantiated authorization views can be different for each user. Even in this case, the validity of a query would be tested against the instantiated authorization views, as before.

We also allow a class of authorization views which we call *access pattern views*, which have parameters that must be bound at access time, but can be bound to any value.

create authorization view SingleGrade **as**
select * **from** Grades **where** student-id = $$1

Here, $$1 indicates any specified value. Thus, this view does not allow the user to see the grades of all students but by specifying a particular *student-id*, the user can see the grades of that student. A secretary may be provided with such an authorization, allowing her to see grades of specific students, while at the same time preventing her from getting a list of all students. (The notion of access patterns has been studied in the context of mediator systems, e.g. [27].) Web search forms with mandatory fields, for which a value must be entered, are a typical example of the use of such authorizations. In the initial part of the paper, we consider only the basic parameterized authorization views, and consider how to handle access pattern views in Section 6.

3. TRUMAN'S WORLD MODELS

In this section, we first look at the Virtual Private Database feature of Oracle's 9*i*R2 RDBMS [1]. We then present the Truman model, which generalizes the query modification approach using the parameterized view framework. It should be noted that the VPD and Truman models support fine-grained and authorization-transparent access control. However, they have some major limitations, described in Section 3.3.

3.1 Oracle's Virtual Private Database

The *Virtual Private Database (VPD)* feature of Oracle's 9*i*R2 RDBMS [1] provides fine-grained access control by transparently modifying the user query. The authorization policy is encoded into 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 or view that she is authorized to see. The policy functions can in turn include callouts to C or Java functions. These functions can access operating system files or a central policy store.

When a user logs in, a *secure application context* is created in the database. This is used to store user-specific information, i.e. parameter values, based on which the policy functions will return the appropriate predicates. In general, different applications can define different security policies, depending on their access control needs.

3.2 Truman Model

Our coinage of the name *Truman Model* is inspired by the artificial world spun around the character of Truman Burbank in the movie "The Truman Show". The idea behind the Truman security model is to provide each user with a personal and restricted view of the complete database. User queries are modified transparently to make sure that the user does not get to see anything more than her view of the database. The returned answer is correct with respect to the restricted view, if we assume the database has no other data. However, users may have more information about the database from external sources (e.g., the user may know that there are other students registered for CS101), and the answers could be inconsistent with such information.

In the Truman model, the DBA defines a parameterized authorization view for each relation in the database. This view defines all that the user can access from this database relation. The user query is modified transparently by substituting each relation in the query by the corresponding parameterized view (the user can write queries on base relations in addition to the authorization views). Values of run-time parameters like *user-id*, *time* etc. are plugged in before the modified query is executed.

The parameterized view framework provides a more general way to express authorization policies than the technique of adding *where* clause predicates used in VPD, since it can additionally perform other actions such as hiding or falsifying specific attribute values which cannot be done by VPD (for example, if an authorization policy permits a student to see her grades tuple but only after the *grade* attribute has been modified as - *all A and B grades to 'High', and all other grades to 'Low'*).

3.3 Limitations of VPD and Truman Models

A major drawback of VPD and Truman models results from the fact that the query that is executed on the database is a transparent modification of the user query. This may cause inconsistencies between what the user expects to see and what the system returns.

- Suppose a student is allowed to see only her own grades from the *Grades* relation. Then, the DBA has to deploy the following authorization view *MyGrades* (alternately in VPD, by the use of appropriate functions to generate *where* predicates).

  create authorization view MyGrades **as**
  select * **from** Grades **where** student-id = $user-id

  Let q be the query posed by the user.

  q: **select avg**(grade) **from** Grades

  The system-modified query

  q' **select avg**(grade) **from** MyGrades

  returns the average of the user's own grades, giving her an impression that her average grade is the same as the overall average grade. Such misleading answers can cause further problems when queries are a part of some larger logic or application. The Non-Truman model described later removes this limitation of the Truman model by not adopting the query modification approach. Either the user query is executed without any modification or rejected outright.

- Suppose in addition to the view *MyGrades* given above, there is a view *AvgGrades* which allows the user to see the average grade for each course. If the user wants to find the average grade for a course and is unaware of the view *AvgGrades*, she will write the query on the base relation. However, if the query is transparently rewritten as in the example above, she will get misleading results in spite of having the correct authorizations to run the query.

- The rewritten query executed by the system may be different from the query posed by the user, and may have very different execution characteristics. For example, if each base relation is substituted by a view, and the views are complex (as they may be, if they express complex authorization policies), the rewritten query may be quite expensive to execute. In the case of VPD, equivalently, the conditions introduced in the where clause may have complex subqueries, which may be expensive to execute. A user query and an authorization view used to replace a relation in the query may both contain the same test, leading to redundant tests. If the test involves a join, the Truman-modified query may also contain redundant joins. Removal of redundant joins is an extra task for the query optimizer, and if not removed, the redundant joins would result in wasted execution time. The Non-Truman model does not suffer from this problem.

4. NON-TRUMAN MODEL

Although the Truman's world models provide fine-grained and authorization-transparent access control, they suffer from the limitations described in Section 3.3. The Non-Truman model is motivated by these limitations. Under the Non-Truman model, the query is subjected to a validity test, failing which, the query is rejected and the user is notified about this (this can be handled like an exception by the user application). If the query passes the test, it is allowed to execute normally, without modification.

Intuitively, under this model a user query is said to be valid, if it can be answered using only the information contained in the authorization views available to the user. The user can write queries against the database relations. The DBA can create several authorization views, one for each access policy, and any of those views can *testify* for the validity of the user's query.

We first define the notion of *unconditional validity*, which captures the queries that can be inferred to be equivalent to some expression written on just the authorization views. This is the same notion proposed by Motro [20] and by Rosenthal *et al.* [24, 22, 23].

The notion of unconditional validity may seem to capture all queries that can be answered using the information contained in the authorization views. However, this is not the case, and we characterize a class of queries, called *conditionally valid queries*, whose validity is contingent on the current database state, rather than just the authorization information.

4.1 Granting of Authorization Views

In the Non-Truman model, an authorization view can be treated just like other privileges in SQL. For instance a user can use an authorization view only if he/she has been granted (select) access on the authorization view. We use the term *available authorization views* to denote the set of authorization views that have been granted to the user.

We assume that when a user is granted an authorization view, the user can see the definition of the view. Without this information, the user will not be able to understand why a query was deemed to be valid or invalid. Also, this makes our access control model more robust, as users in general may be aware of the access policies in their organization.

4.2 Unconditional Validity

This notion of validity captures those queries that can be declared as valid irrespective of the current database state.

DEFINITION 4.1. **(Unconditionally Valid Query)** *For given parameter values, a query q is said to be unconditionally valid if there is a query q' that is written using only the instantiated authorization views, and is equivalent to q, that is, q' produces the same result as q on all database states.* □

Note that 'equivalence' in the above definition refers to multiset equivalence, as in the case of SQL.

The intuition behind categorizing such queries as valid is that the user could have just as well used q' and got the same information, provided they could have inferred that they would get the same information from q and q'. By categorizing q as valid we save the user (or programmer) the trouble of rewriting q as q'.

EXAMPLE 4.1. Consider the query q

select avg(grade) **from** Grades
where student-id = '11'

Assume that $user-id is 11, and the authorization view *MyGrades* given in Section 1 is available. The above query q is equivalent to

the following query q' that applies aggregation on the *MyGrades* view

select avg(grade) **from** MyGrades

and is therefore unconditionally valid.

Now suppose the user had the following authorization view:

create authorization view AvgGrades **as**
select course-id, **avg**(grade)
from Grades **group by** course-id

The query

q1: **select avg**(grade) **from** Grades where course-id = 'CS101'

can then be rewritten using only *AvgGrades*, and is thus unconditionally valid. □

Note also that if the user is aware of integrity constraints that restrict the set of legal database states, we can treat q and q' as equivalent if they return the same answer on all database states that satisfy the integrity constraints. However, integrity constraints that the user is not authorized to know should not be taken into account when making this inference. Otherwise, the act of declaring q as valid could allow the user to infer the presence of the integrity constraints.

The definition of unconditional validity is essentially the same as the notions of query validity proposed by [20, 24, 22] and [23], except for the issue of authorization to see integrity constraints. However, the notion of conditional validity, explored next, is novel to this paper.

4.3 Conditional Validity

Intuitively, a query q can be considered valid if the user could have written an "equivalent" query using only the authorization views. We have so far assumed that an "equivalent" query gives the same answer on all database states. However, the user may be aware of some information about the current database state. Based on this information, the user may be able to infer that the given query would give the same result as another query q' that uses only authorization views, on the current database state. Intuitively, the given query q should be declared as valid, since the user could have got the same information by executing q', even though the two queries may not give the same result on all database states.

We give two examples of such a situation below; the examples also illustrates some problems that could arise from a naive approach to solving the problem. We then define our extended notion of validity, which we call conditional validity.

EXAMPLE 4.2. Consider Example 4.1 again. Suppose the view *AvgGrades* was modified to only show average grades for courses that had an enrollment of 10 or more students (call this modified view *LCAvgGrades*). The query

q: **select avg**(grade) **from** Grades where course-id = 'CS101'

is then not equivalent to a selection on *LCAvgGrades*, and cannot be declared unconditionally valid. However, if 'CS101' has 10 or more students, then the view *LCAvgGrades* can indeed be used to answer query q. Thus, the validity of q depends on the database state. □

EXAMPLE 4.3. Now suppose the authorization view *Co-studentGrades* defined earlier in Section 2 is available (the view permits a user to see all grades-information of courses for which the user has registered). Consider the following query:

q: **select** * **from** Grades **where** course-id = 'CS101'

It is not possible to apply any operations on the authorization view *Co-studentGrades* to get a query q' that is equivalent to q across all database states. To prove this, suppose we have such a query q'. Then, we can have a database instance which has some CS101 grades but the user is not registered for any course; in that case the result of *Co-studentGrades* and hence q' is empty, whereas that of q is not. So, such a query q' cannot exist.

However, if the user has registered for CS101 (a requirement on the current database state), the user can use the query

q': **select** * **from** Co-studentGrades
where course-id = 'CS101'

to get the result of q. Thus, one may assume that it is safe to declare q as valid in this case.

However, there is a further complication in declaring q to be valid. Suppose there is no authorization view that tells the user what courses she has registered for. If the user has registered for CS101 and there were grades for CS101, the user would be able to infer that she is registered for CS101 by simply looking at the result of *Co-studentGrades*. But suppose that no grades have been entered for CS101. In this case, the user cannot infer her registration status for CS101 using *Co-studentGrades*. In this situation, if the system accepts query q, the user can infer that she is registered for CS101, even though the actual answer set of the query q is empty. Thus, the act of accepting a query can itself reveal information that was not supposed to be revealed by the authorization views. In contrast, rejection is safe here, since if the system rejects q, the user can only infer that one of these is true: (a) she is not registered for CS101 or (b) she is not allowed to know if she is registered for CS101.

Note that in the above case (when the user is registered for CS101), the answer to q as well as to q' are the same in the current database state, but since the user does not know whether she is registered for CS101 or not, she cannot infer that the two queries would give the same answer. The user could have inferred that q' would give the same result as q only if she knew that she was registered for CS101. This shows that the naive approach of testing the equivalence between q and q' over just the current database state, leaks unauthorized information.

In general, q should be declared valid only if the user would be able to infer that q and q' would give the same result, using only the information that the available authorization views provide without knowing the complete current state of the database. In this example, the user would be able to make this inference only if (a) the user is registered for CS101, and (b) the user is authorized to know if she is registered for CS101. □

The above example illustrates the difficulties in going beyond the notion of unconditional validity. Our characterization of conditional validity extends the class of queries declared to be valid, while ensuring that information is not leaked in situations such as the above. Before we proceed to the characterization of conditional validity, we need some definitions.

DEFINITION 4.2. **(PA-Equivalent Database States)** *For given parameter values, two database states D_1 and D_2 are said to be parameterized-authorization-view equivalent, if each instantiated authorization view returns the same result for D_1 and D_2.* □

Armed with this definition, we extend the notion of validity as follows.

DEFINITION 4.3. **(Conditionally Valid Query)** *For given parameter values, a query q is conditionally valid in a database state D, if there is a query q' that is written using only the instantiated authorization views, and is equivalent to q (i.e. returns the same (multi)set of answers as q) on all database states that are PA-equivalent to D.* □

This definition of validity takes into account the database state, unlike unconditional validity. The intuition for the above is as follows. The information in the authorization views does not define the exact database state; as far as the user knows, the database may be in any state that is consistent with the results of the authorization views seen by the user; these are the states that are PA-equivalent to the actual database state D. The query q' is unconditionally valid by definition, and as long as q is equivalent to q' in all these possible states, the user can infer that the result of q can be obtained by executing q'. Thus, we declare q to be conditionally valid.

Note that since users have access to the definitions of views that they are authorized to use, an intelligent user could take the query q and come up with the query q' and infer that they would return the same result on the current database state, based on the partial information about the current database state available to that user. In our model, the system makes the inference and declares q as conditionally valid, saving the user the effort of rewriting query q as q'.

EXAMPLE 4.4. Now we come back to Example 4.3, where the user, say with *user-id* 11, wanted all CS101 grades and the authorization policy provided the view *Co-studentGrades*. Consider the following query.

select 1 **from** Registered
where student-id = '11' **and** course-id = 'CS101'

Suppose the above query is valid (and thus its result is visible to the user), and its result on the current database state is non-empty. Then, the user can infer that she is registered for CS101 and hence that all CS101 grades will appear in her view *Co-studentGrades*. Thus, she can get all CS101 grades by writing a query that selects the CS101 grades out of the view *Co-studentGrades*. This query and the given query q requesting all CS101 grades are equivalent under all database states PA-equivalent to the current state, and thus q is conditionally valid.

However, if either the student is not registered for CS101 or is not authorized to know if she is registered for CS101, then there could be PA-equivalent states where q is not equivalent to the selection on *Co-studentGrades*, and (assuming no other view allows q to be declared valid) q would be rejected as not valid. □

The above definition of conditional validity is arguably the weakest definition possible, since

1. Any valid query result has to be computable as a query using only the given authorization views, and as a result the two queries must be equivalent at least on the current database state, and

   A query can thus be labeled as valid only if the user can infer that some query on the authorization views is equivalent to the given query on the current database state.

2. However, the user does not have full information about the current database state; all the user knows about the current database state is the information revealed by the authorization views. Thus, the query should be permitted only if the user could have come up with a query on the authorization views and inferred it to be equivalent to the given query on all PA-equivalent database states. This is exactly the definition of conditional validity.

To summarize, we define the *Non-Truman access control model* as follows: the model allows a query to be executed only if the query has been inferred to be unconditionally or conditionally valid. Of course, if a query is unconditionally valid, it is also conditionally valid.

4.4 Authorization of Updates

We can extend our authorization model to handle update queries by testing them against parameterized conditions such as those illustrated below.

authorize insert on Registered
where Registered.student-id = $user-id

authorize update on Students(address)
where old(Students.student-id) = $user-id

The first condition permits a user to add a registration tuple only if it corresponds to her course registration, while the second one permits any student to update her address in the *Students* relation.

Note that we only consider queries in the rest of the paper. In our model, checking validity of updates is a simpler task than validity checking for queries. We consider updates individually, and checking if the insertion/deletion/update of a particular tuple is authorized only requires evaluation of a (fully instantiated) predicate, rather than the more complex inferencing required for checking queries.

5. TESTING FOR VALIDITY

In this section we first outline (in Section 5.1) prior work on query rewriting using views, which can be used for checking unconditional validity. In Sections 5.2 through 5.4, we then give rules for inferring unconditional and conditional validity of queries. The inference rules are very powerful and handle the multiset semantics of SQL. We discuss issues on completeness of validity checking, in Section 5.5. The inference rules can be implemented as part of a query optimizer, as detailed in Section 5.6.

For simplicity, we assume that there are no nested subqueries in the SQL queries.

5.1 Earlier Results on Query Rewriting

The problem of query rewriting using views has received a lot of attention in the past. Halevy [15] gives a comprehensive survey on the topic. Amongst work on query rewriting, the work closest to our motivation comes from query optimization, where the rewritten query is required to be equivalent to the original one. For example, Chaudhuri et al. [6] consider optimization of select-project-join queries with arithmetic comparisons, using materialized views. Query containment algorithms that take multiset semantics into account are discussed in [7]. Several works describe how view matching can be added as a set of transformational rules in a rule-based optimizer. Goldstein and Larson [12] do it on the SQL Server optimizer, Zaharioudakis et al. [28] on IBM DB2, and Bello et al. [3] on Oracle 8i. There has also been work on rewriting queries when either the views or the queries contain grouping and aggregation [14, 26, 8, 28].

However, in query optimization the goal of the rewriting algorithms is not just to find such a rewriting, but also to produce an efficient query plan for the rewritten query. Since we only use the rewriting to test for validity, we are not concerned with executing the rewritten query efficiently; in fact, it is the *original* query that will be executed. One approach to inferring validity was proposed by Motro [20]; we outline that approach briefly in Section 7. Our approach is different, and in the following subsections we present a set of rules for inferring validity.

5.2 Basic Inference Rules

We begin with some simple inference rules.

Inference Rule U1: *If v is an authorization view, then the query v is unconditionally valid.* □

Inference Rule U2: *If $q_1, q_2, ..., q_n$ are unconditionally valid queries, and E is an expression combining $q_1, q_2, ..., q_n$, with no other relation appearing in it, then E is an unconditionally valid query.*

□

That is, if a query can be expressed as an operation (projection, selection, join etc.) on top of unconditionally valid subexpressions, the query is itself unconditionally valid. Suppose, the authorization policy provides the view *MyGrades* given earlier. Then the following are some queries that can be inferred to be unconditionally valid using this rule (we assume that the user-id parameter value is 11).

select grade **from** Grades
where student-id = '11'

The above query can be expressed as a projection of the instantiated authorization view *MyGrades* (instantiated with $user-id = 11).

select course-id **from** Grades
where student-id = '11' **and** grade = 'A'

The above query can be expressed as a selection $\sigma_{grade='A'}$ on the instantiated authorization view *MyGrades*, followed by a projection on *course-id*.

Although conceptually simple, rule U2 is not trivial to implement, as a query may be written in one of many different equivalent forms. Techniques developed for query rewriting, described in Section 5.1, are needed to implement this rule. We address this issue further in Section 5.6.

5.3 Inference Rules Using Integrity Constraints

Inference rule U2 allowed us to infer the validity of an expression from the validity of subexpressions. In some cases, given that an expression is valid, we can infer the validity of a subexpression, by using integrity constraints as illustrated by the following example.

EXAMPLE 5.1. Consider the following authorization view.

create authorization view RegStudents **as**
select Registered.course-id, Students.name,
Students.type
from Registered, Students
where Students.student-id = Registered.student-id

Suppose the schema includes an additional integrity constraint that says that each student has to register for at least one course. Then, each tuple in the *Students* relation will match some tuple in the *Registered* relation under the join condition, and hence, appear in the result of this view. As a result, the projection (with duplicate elimination) of the above view on the *Students.name, Students.type* attributes will be equivalent to the following query:

q: **select distinct** name, type **from** Students

Thus, given the authorization view *RegStudents*, we can infer the validity of q, in the presence of such an integrity constraint. If q and *RegStudents* were written using relational algebra, q would be a subexpression of *RegStudents* (with an extra duplicate elimination step).

Note that a modified version of q with the keyword *distinct* dropped, is not multi-set equivalent to a projection (with or without duplicate elimination) on *RegStudents*. Suppose there are n students with a given name John, and type FullTime, and each is registered for m courses. Then the projection on *RegStudents* would have $n * m$ copies of the tuple (John, FullTime), whereas the query q would have only n copies. The view does not provide enough information to retrieve the value of n. As a result, we cannot infer the validity of the modified version of q using *RegStudents*. □

We formalize the inference in the above example using the following rule.

Inference Rule U3a: *Suppose that the following conditions are true.*

1. *The following query is unconditionally valid.*

   q: **select** A **from** R **where** P

   Where R is a set of relations, A is a set of constants or attributes from the relations in R, and P is a predicate involving the attributes from the relations in R.

   Further, there exists a disjoint partition of R into two sets of relations R_c and R_r, a disjoint partition of A into two sets of constants and attributes A_c and A_r, and $P = P_c \wedge P_r \wedge P_j$, such that:

   (a) *All attributes in A_c come from the relations in R_c, while all attributes in A_r come from the relations in R_r.*
   (b) *All attributes in the predicate P_c come from the relations in R_c, while all attributes in the predicate P_r come from the relations in R_r.*
   (c) *P_j is a predicate that joins attributes of relations that come from across the two sets R_c and R_r.*

2. *Let*

   v_c: **select** * **from** R_c **where** P_c
   v_r: **select** * **from** R_r **where** P_r

   The schema has integrity constraints such that for queries v_c ("view-core") and v_r ("view-remainder"), for every tuple in the result of v_c, there is a tuple in the result of v_r, such that the join conditions expressed by P_j are satisfied by these two tuples.

   Further, the relevant integrity constraints are visible to the user.

Then, the following query is unconditionally valid.

q': **select distinct** A_c **from** R_c **where** P_c □

The conditions of the inference rule U3a help ensure that q' is equivalent to a projection (with duplicate elimination) of q on the attributes A_c, and as a result we can infer that q' is unconditionally valid. Note that q' without the 'select distinct' may not be multi-set equivalent to a projection (without duplicate elimination) of q on A_c, as illustrated in Example 5.1.

EXAMPLE 5.2. Applying Rule U3a to Example 5.1, we would have

q: $RegStudents$
v_c: **select** * **from** Students
v_r: **select** * **from** Registered
q': **select distinct** name, type **from** Students

and the integrity constraint ensures condition 2 of Rule U3a. □

Inference Rule U3b: *Inference rule U3b is the same as U3a except that the top level 'select' in query q is replaced by a 'select distinct'.* □

This shows that the query q in U3a gave more information to the user than what is needed to infer the results of the query q'. A set version of the result of q is sufficient for the same. Note that we can infer U3a from U2 and U3b, since if we are given a query q without a 'distinct' clause, we can always add the clause (using U2), and then apply U3b.

Under some additional conditions over those in U3a, we can reconstruct the multiplicity of the tuples in q' without the *distinct* keyword. Under these conditions, q' without the *distinct* keyword can be inferred to be valid, as formalized below.

Inference Rule U3c: *This inference rule is got from rule U3a by:*

1. *Adding an extra condition 1d: all the attributes from R_r that appear in the predicate P_j should appear in the set A_r.*
2. *Adding an extra condition 3: the following query must be unconditionally valid.*

   q_{rj}: **select** A_{rj} **from** R_r **where** P_r

   Where, A_{rj} is the set of all attributes from R_r that appear in the predicate P_j.
3. *Replacing the top level 'select distinct' in q' by a 'select'.* □

The extra conditions introduced by this rule ensure that the user is able to reconstruct the multiplicity of the tuples in the view-core; this can be done for each value of A_{rj} by dividing the number of tuples with that value in $\Pi_{A_c \cup A_{rj}}(q)$ by the number of tuples with that value in q_{rj}.

THEOREM 5.1. **(Soundness of Rules)** *Inference rules U1, U2, U3a, U3b, and U3c are sound.* □

Inference rule U3a and its extensions U3b and U3c are quite powerful, and can be combined with U2 to make complex inferences as illustrated by the following examples.

EXAMPLE 5.3. Consider the following query.

q_f: **select distinct** name **from** Students
where Students.type = 'FullTime'

Suppose the authorization view *RegStudents* is available, but there may be students who are not registered for any course. As a result we cannot infer the validity of the query that selects names of all students. However, suppose we have an integrity constraint that all full-time students must have registered for a course. We can then infer the validity of the above query, by using rules U2 and U3a, as follows. Given the validity of *RegStudents*, the following selection query on *RegStudents* must be valid

select distinct name **from** RegStudents
where Students.type = 'FullTime'

Expanding the view, and pushing in the selection condition, we get the following query

select distinct Students.name
from Students, Registered
where Students.student-id = Registered.student-id
and Students.type = 'FullTime'

We can now apply Rule U3a as we did in Example 5.2, except that v_c would be

v_c: **select** * **from** Students
where Students.type = 'FullTime'

and thus infer the validity of q_f. □

EXAMPLE 5.4. As another example of the power of these rules, suppose we have a relation *FeesPaid(student-id)* recording all students who have paid the fees. Consider a query

q_j: **select distinct** name **from** Students, FeesPaid
where Students.student-id = FeesPaid.student-id

and suppose that there is an integrity constraint that anyone who has paid the fees must be registered for some course. Further suppose that *FeesPaid* is visible (i.e., authorized). Let q denote the natural join of *RegStudents* and *FeesPaid*. Using U2, we can infer that q is valid; expanding out the view *RegStudents* in q, and setting

v_c: **select** * **from** Students, FeesPaid
where Students.student-id = FeesPaid.student-id
v_r: **select** * **from** Registered

we can infer the validity of q_j using U3a (followed by a projection on just the *name* attribute). □

5.4 Inferring Conditional Validity

We now give a set of rules for inferring the conditional validity of queries.

Inference Rule C1: *If a query q is unconditionally valid, it is conditionally valid in all database states.* □

Inference Rule C2: *If queries $q_1, q_2, ..., q_n$ are conditionally valid in a database state D, and E is an expression combining $q_1, q_2, ..., q_n$ with no other relation appearing in it, then E is conditionally valid in D.* □

The following inference rule formalizes the reasoning used in Example 4.4.

Inference Rule C3a: *Suppose in a database state D*

1. *The following query is conditionally valid in D.*

   q: **select** A **from** R **where** P

   Where R is a set of relations, A is a set of constants or attributes from the relations in R, and P is a predicate involving the attributes from the relations in R. There exists a disjoint partition of R into two sets of relations R_c and R_r, a disjoint partition of A into two sets of attributes A_c and A_r, and $P = P_c \wedge P_r \wedge P_j$ such that:

   (a) All attributes in A_c come from the relations in R_c, while all attributes in A_r come from the relations in R_r.
   (b) All attributes in the predicate P_c come from the relations in R_c, while all attributes in the predicate P_r come from the relations in R_r.
   (c) P_j is the predicate that joins attributes of relations that come from across the two sets R_c and R_r.
   (d) All attributes of the relations in R_c appearing in the predicate P_j should also appear in the set A_c.

2. *Consider an instantiation of all the attributes in P_j, such that P_j is satisfied. Let this instantiation be given by predicates P_{ic} which instantiates all attributes in P_j that come from R_c, and P_{ir}, which instantiates all attributes in P_j that come from R_r.* [1]
3. *The following query is conditionally valid in database state D, and produces a non-empty result on D.*

   v_r: **select distinct** *1* **from** R_r **where** $P_r \wedge P_{ir}$

Then, the following query is conditionally valid in D.

q': **select distinct** A_c **from** R_c **where** $P_c \wedge P_{ic}$ □

The conditions of the inference rule C3a help ensure that q' is equivalent to the following query q'' for all database states that are PA-equivalent to D. Note that q'' is written using only the conditionally valid query q.

q'': **select distinct** A_c **from** q **where** P_{ic}

[1] As a special case, if the view has only equi-joins, then (the instantiation defined by) P_{ic} uniquely determines (the instantiation defined by) P_{ir}

EXAMPLE 5.5. Applying the above rule to Example 4.4:
P_c = *true*
P_r = (Registered.student-id = '11')
P_j = (Grades.course-id = Registered.course-id)
P_{ic} = (Grades.course-id = 'CS101')
P_{ir} = (Registered.course-id = 'CS101')
The rest of the reasoning in applying the rule is as described in Example 4.4, leading us to infer the validity of the query **select distinct * from** *Grades* **where** *course-id* = 'CS101'. Since the *Grades* table has a primary key, the distinct keyword can be dropped. □

We note that if the query v_r in Condition 3 above can be inferred to be non-empty regardless of the database state (e.g., based on integrity constraints), then we can infer q' to be unconditionally valid using U3a.

Under some additional conditions over those in C3a, we can reconstruct the multiplicity of tuples in q' without the *distinct* keyword. The next inference rule deals with this.

Inference Rule C3b: *This inference rule is got from rule C3a by:*

1. *Adding an extra condition 1(e): The instantiation defined by P_{ic} uniquely determines the instantiation defined by P_{ir} (for e.g., if P_j is an equi-join).*
2. *Replacing the top level 'select distinct' in v_r by a 'select'.*
3. *Replacing the top level 'select distinct' in q' by a 'select'.* □

THEOREM 5.2. **(Soundness of Rules)** *Inference rules C1, C2, C3a and C3b are sound.* □

5.5 Completeness of Inference Rules

Although we have given a fairly powerful set of inference rules, they are not complete. In fact, although unconditional validity is decidable for restricted languages, such as conjunctive queries, in the general case it is undecidable (the question of whether a query is identically empty can be reduced to it) (see, e.g. [15]). Thus, we cannot expect a complete set of rules that would lead to an algorithm for unconditional validity in the general case. The decidability of conditional validity is open, even for conjunctive queries. While we believe that our inference rules are likely to handle most common queries, there will be some queries that the rules cannot infer to be valid, even if they are valid. As a result, the queries would be incorrectly rejected, although unlike in the case of VPD, results of accepted queries would never be incorrect. This does not mean that the user cannot execute such queries: the user can rephrase the queries, rewriting them to use the authorization views instead of the database relations. The queries would then be trivially recognized as valid. Although the benefit of authorization-transparency may be lost for such queries, it is retained for the vast majority of simpler queries, which are successfully handled.

5.6 Validity Testing Using Inference Rules

We have given a set of inference rules which derive unconditionally and conditionally valid queries, given a set of instantiated authorization views. We now address the problem of inferring if a given user query q is valid in the presence of instantiated authorization views $v_1, v_2, ..., v_n$. This is a problem of *goal-directed* inferencing. Specifically, we describe how a Volcano-based query optimizer can be extended to perform validity tests.

5.6.1 The Volcano Query Optimizer

The *Volcano query optimizer* [13] is used for generating the best evaluation plan for a given query.

Volcano works by first generating an AND-OR DAG representation of all possible evaluation plans for a given query, and then choosing the plan with the least estimated cost. Figure 1 (a) shows a query $A \bowtie B \bowtie C$, and Figure 1 (b) shows its its initial AND-OR DAG representation. The rectangular nodes in the DAG are called *equivalence nodes*, and they represent a logical expression, while the circular nodes are called *operation nodes*, and they represent an operation (like selection, projection, join).

An equivalence node may have multiple children (each of which must be an operation node). Equivalence nodes are called OR nodes since any of their children operation nodes may be chosen to compute the result of the equivalence node. The queries defined by the different choices are equivalent, i.e., they return the same result. Operation nodes are called AND nodes, since all their children need to be evaluated in order to compute the result of the operation; the children of an operation node must be equivalence nodes.

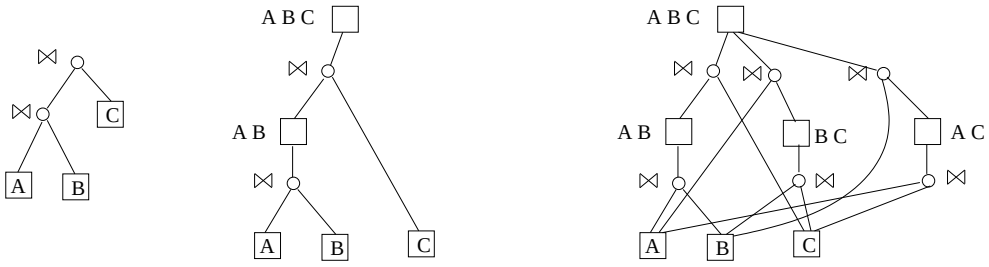
Given an initial query, algebraic equivalence rules, such as join associativity and commutativity, can be applied to the initial query DAG. Applying an equivalence rule to an operation node results in an alternative equivalent expression, which is added as another child of the parent equivalence node (if it is not already present). The equivalence rules are applied repeatedly, till no new expression can be generated. The resultant DAG is called the *expanded AND-OR DAG*, and it compactly represents all the alternative query evaluation plans that can be inferred using the equivalence rules. Figure 1(c) shows the expanded DAG for the query shown in Figure 1(a). For this example, only the join associativity transformation rule is used. Note that, disregarding join commutativity, there are three ways of evaluating this query. For the case of join ordering, the AND-OR DAG is at worst exponential in the number of relations, but represents a much larger number of query plans. For query optimization, the AND-OR DAG based on "logical" operations such as join has to be further expanded to represent alternative "physical" operations, such as merge-join or hash-join, which can be used to implement the join operation. This step is irrelevant for our purposes.

The multi-query optimization framework of [25] extends the Volcano algorithm to check if a query can be rewritten using materialized views (or materialized intermediate results). An important aspect of this framework is *unification* of nodes. If during the expansion of a node by applying a transformation rule it is found that the subtree generated is the same as another one in the DAG, the two are unified. Essentially, this means that when two or more queries are represented in a single expanded DAG, their common subexpressions are detected and unified. Thus, if a materialized view is equivalent to a subexpression of a query, their equivalence nodes would get unified. The materialized view can then be used, instead of evaluating (one of) the expressions represented by the equivalence node. "Subsumption derivations", which allow a selection to be evaluated from a weaker selection or a coarse-grained aggregation from a finer-grained one, are also added to the DAG.

In our context, we can use authorization views in place of materialized views, and use the above technique to detect when a query can be evaluated using the authorization views. We describe the idea in more detail in the following subsections.

5.6.2 Implementing Basic Inference Rules

Under the basic inference rules $U1$ and $U2$ stated earlier, given a set of authorization views $V = \{v_1, v_2, ..., v_n\}$ and a query q, we would like to test if the query can be completely rewritten using the available views. This is done as follows. Equivalence rules are applied to the query to get an expanded DAG. The unexpanded DAGs for the instantiated authorization views are unified with the



(a) Initial Query (b) DAG representation of query (c) Expanded DAG after transformations (Commutativity not shown)

Figure 1: Volcano Data Structures

expanded query DAG (there is no need to expand the authorization view DAGs, i.e., apply equivalence rules to the authorization views, to implement the basic inference rules). The root equivalence nodes for all views are marked as *valid*. The following rules are applied bottom-up to the DAG:

1. An equivalence is marked as valid if any of its children operation nodes is marked as valid.
2. An operation node is marked as valid if all its children equivalence nodes are marked as valid.

If the root equivalence node of the query is marked as valid, then the query is inferred to be unconditionally valid. Rules C1 and C2 can be implemented in the same way as U1 and U2.

Although the above algorithm is sound, it may not be able to infer validity of some unconditionally valid queries. Given the set of views $V = \{A \bowtie B, B \bowtie C\}$, it is possible that a query of the form $A \bowtie B \bowtie C$ can be rewritten completely using the views only if we decompose the query as $(A \bowtie B) \bowtie (B \bowtie C)$. Volcano does not generate such query plans as adding redundant relations in the join increases query complexity. Extending the algorithm to handle such cases is a topic of future work.

5.6.3 Implementing Complex Inference Rules

We now consider how to implement the U3 and C3 family of rules. Unlike the case of basic inference rules, here we want to infer the validity of a subexpression from a complete view. In this case, we need to apply equivalence rules to authorization views in addition to the user query. The technique of [25] can be used to unify the DAGs generated from the query and the views.

Inference rule U3a and U3b can be implemented as follows. Any equivalence node marked as *valid* represents an unconditionally valid query. For any such node n, we can traverse down the DAG to a child that represents a join operation. This can be seen as the join of a core and a remainder. We need to make sure that for each tuple in the core, there is a matching tuple in the remainder, based on the join condition. The most natural case when this happens is when the remainder is a single relation and the join is a foreign key join. In such a case, we can create a new operation node on top of the core's equivalence node. This operation node will project (with duplicate elimination) only the attributes of the view core that are visible at the node n. Finally, this operation node can be attached to a new equivalence node marked as *valid*.

Inference rule U3c can be implemented by a slight modification of the above scheme, to ensure the other conditions of U3c are checked.

Now we come to the conditional validity rules. Any equivalence node marked as unconditionally valid can be marked as conditionally valid in the current database state. We find all equivalence nodes n that have two different parent operation nodes – one selection node (call it s) and one join node (call it j). Further, the parent of j must have been marked as (conditionally or unconditionally) valid, and the selection s should instantiate all the attributes that appear in the join predicates in j. Here, the selection on top of n represents the query q' of inference rule C3a. The query v_r can be got by instantiating the join variables of the remainder (here, the remainder is the other child of node j), and adding an operation node on top that projects a '1' for every tuple in this modified remainder. As required by condition 3 of C3a this query should be conditionally valid (marked as conditionally valid after unification with the rest of the DAG) and should return a non-empty result on the current database state.

We have implemented the basic inference rules and are currently working out the implementation details of the complex inference rules. We intend to carry out performance tests subsequently.

5.7 Optimizations of Validity Checking

One major concern about using the Non-Truman model is the overhead of validity checking, especially for queries with a small execution time. Validity checking with the basic inference rules does not require equivalence rules to be applied to the views, and hence does not increase the cost significantly beyond normal query optimization. The complex inference rules do require equivalence rules to be applied to the views, which can be somewhat expensive in the presence of a large number of authorization views.

We are currently working on techniques to reduce the overheads. Among the techniques we are considering are the following. Given a query, we can eliminate authorization views that cannot possibly be of use in validating the query. Most uses of a database are from application programs, which execute the same queries repeatedly, albeit with different constant values, for different users. For ODBC/JDBC prepared statements, we can analyze the query without the actual parameters when the query is prepared, and come up with a cheap test that is used each time the query is executed (e.g. a particular parameter value matches the current user-id). Even if the application issues the query without explicitly using prepared statements, the workload can be analyzed to find equivalent patterns generated implicitly by the application. If the same query is reissued multiple times in a session, we can cache the results of the validity check (assuming no underlying data on which it depends changes during the session).

6. EXTENSIONS

We now consider some extensions of the authorization view model that we have proposed.

Recall access pattern views from Section 2; such views had special parameters prefixed by $$ to indicate that queries got by instantiating these to any value are all authorized. Conceptually, access pattern views can be handled by considering the set of all instantiated versions of the access pattern views, and checking validity against this set of instantiated views.

Our inference procedures can be used by simply treating $$ parameters as constants. The queries that are inferred to be valid would contain the $$ parameters and represent a class of queries that are inferred to be valid. Specifically, let q be a query with $$ parameters that is inferred to be valid; any query q' obtained by instantiating the $$ parameters of q can correspondingly also be inferred to be valid.

Consider a query $(r \bowtie_{r.B=s.A} s)$ where r is valid, and an access pattern authorization view

select * **from** s **where** s.A = $$A.

Although we cannot see all of s, the above query can be evaluated by stepping through each tuple of r and finding matching tuples of s; thus the query $(r \bowtie_{r.B=s.A} s)$ is valid since it can be computed from available authorized information. The above technique for joining r and s is called a *dependent join* in the context of query processing under access pattern restrictions (see e.g. [27]). Techniques developed for query processing under access pattern restrictions, can be used for the task of inferring validity; we omit details.

Further, if we know that every tuple in s has a matching tuple in r, we can easily generate all of s by projecting the above join on the attributes of s, and we can thus infer that the query (**select distinct** * **from** s) is also valid. Again, we omit further details.

Delegation of authorization is important in many settings. Delegation can be done outside of our inferencing system: we can use any delegation specification technique to collect all available authorization views, whether directly granted or delegated, and then run our inferencing techniques on the resulting set of authorization views.

7. RELATED WORK

Several models have been proposed in the literature for specifying and enforcing access control in databases [5].

Much of the prior work on using views as the basis for access control is not authorization-transparent. However, as mentioned above, authorization-transparent access control models using views have been presented earlier by Motro [20] and by Rosenthal *et al.* [24, 22, 23]. They propose and motivate authorization (validity) inference, show multiple applications for validity inference, and discuss benefits and difficulties of using query processing technology to test for equivalence.

In comparison to these proposals, our novel contributions include fine-grained authorization using parameterized/access pattern authorization views, the notion of conditional validity, our inference rules, and our techniques for implementing the inference rules on an optimizer.

In the model proposed by Motro [20], depending on the authorization, the user may get only a part of the answer to a query; however, unlike with the Oracle VPD model, instead of just getting a partial answer, the user also gets a description indicating in what way the answer is partial (e.g., "only grades of user-id 11 have been returned"). Such an answer may be preferable, in many situations, to just rejecting a query as unauthorized.

However, the inference technique in Motro's work uses a separate language (domain relational calculus) for specifying authorizations; only conjunctive queries/views are handled. The inference procedure is based on a QBE-like representation of views, mapping the view representation to "meta-relations" and processing the meta relations using standard relational algebra operators to perform the inference. There is no description of how to extend the procedure to allow, e.g., disjunction, set difference, or aggregation; in particular, set difference and aggregation can turn a partial answer into an incorrect answer. This inferencing technique was generalized by Motro [21] to deal with query result properties other than authorization, but the above limitations remain.

The use of authorization inference as the basis for security in data warehouses is also proposed by Rosenthal *et al.* [24, 22, 23]. In their model, a query Q is inferred to be authorized if there is an equivalent query Q' which uses only authorized views; this is identical to our model of unconditional validity. Although their authorization techniques are presented in the context of distributed data warehouses, they are applicable also to centralized settings. However, although they give examples of inferences, no formal inference procedure or rules are defined.

The *Access Matrix Model* [16] stores the authorizations using a matrix that correlates subjects (like users and their applications), objects (tables, tuples) and the authorizations held by the subjects on the objects. Access control lists are a special implementation of the access matrix model, which store a list of subjects and their authorizations with each object. Theoretically, with the access matrix model and access control lists, we can achieve arbitrarily fine-grained authorizations, but in practice the matrix (or the access control list) would be extremely large, and constructing it will be a tedious task. Also, maintaining this security information is difficult as it needs to be updated with each update of the subject list (addition of users) and requires a new entry for every object.

Most access control list based models support a "deny-semantics". It is straightforward to create authorization views with negation conditions to implement (and generalize) deny-lists. However, equivalence testing may be a bit more complicated under this setting.

The *Flexible Authorization Framework* [19, 17] supports multiple access control policies. It allows the specification of positive and negative authorizations using Prolog-style rules. This framework can represent arbitrarily fine-grained policies for access control. However, they do not address the challenges of efficiently incorporating this inferencing scheme into a database server.

The concept of *multilevel relations* [9, 18] extends the standard relational model by introducing classification labels for each tuple/attribute, and specifying a class for each user as well. The classification is hierarchical (*unclassified*, *secret*, *top secret*, ...) and there is an effective instance of the relation for each class - representing the version of the relation visible to the users assigned to that class. The effective instance of a relation r is in essence a secure view of r. In Denning's technique [9], a user's select-project-join query is modified by replacing each relation by its secure view, as in the Truman model that we described in Section 3. However, multi-level models cannot provide each user with a different authorization, such as access to only their own grades. Thus, they are not useful for our target application scenarios, with fine-grained authorization requirements, with large numbers of users.

Models for *Role-Based Access Control (RBAC)* (see, e.g., [11, 2]) do not have any inherent support for fine-grained authorization, and thus do not directly address the problem we have tackled. However, role-based access control can be used in conjunction with authorization views, e.g. by granting authorization views to roles.

The *inference problem* in database security refers to the possibility of disclosing sensitive information indirectly via inferences that may take into account channels such as integrity constraints, outside domain knowledge, or query correlations. The problem is

surveyed by Farkas and Jajodia in [10]. Although the inference problem is related to ours at an abstract level, the goal underlying the inference problem is to ensure that the users do not get enough information to infer data that they are not allowed to see. Amongst work on inferencing, the most closely related work is by Brodsky et al. [4] who study the problem of inferring facts from the results of other queries, but in the context of multi-level security. Given the query history of a user and a current query, if it is possible to infer a fact that is not accessible to the user (based on the multi-level security model), then the query is rejected. However, their model as well as their inference algorithms are very different from our model and algorithms.

8. CONCLUSIONS AND FUTURE WORK

We have addressed the problem of fine-grained access control in databases. We asserted that current access control models do not achieve the crucial goals that we set out with. We described two models for fine-grained access control - the Truman and Non-Truman models. Both models support authorization-transparent querying. Unlike the Truman model, the Non-Truman model avoids the pitfalls of the query modification approach and allows a great deal of flexibility in authorization, such as authorization of aggregate results. We defined the notions of unconditional and conditional validity, and presented several inference rules for validity. We outlined an approach to validity testing, based on extending an existing query optimizer to carry out validity checking, minimizing the extra effort required during coding as well as during validity testing.

Future work includes implementing and testing the efficiency and degree of completeness of our inference rules. More work is also needed on the implementation of the complex inference rules, on inferencing with access-pattern authorization views, and on handling nested queries. We also plan to explore the use of our techniques for the inferencing problem. Finally, we plan to build a software layer that can add fine-grained authorization to an existing database or application.

Acknowledgments: We wish to thank Soumen Chakrabarti, Rushi Desai, Arvind Hulgeri, Hrishikesh Karambelkar and Navneet Loiwal for discussions and feedback, and the anonymous referees for their detailed feedback and suggestions. Alberto Mendelzon's work was supported by a grant from Natural Sciences and Engineering Research Council of Canada.

9. REFERENCES

[1] The Virtual Private Database in Oracle9ir2: An Oracle Technical White Paper *http://otn.oracle.com/deploy/-security/oracle9ir2/pdf/vpd9ir2twp.pdf*.

[2] G.-J. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM Trans. on Information and System Security*, 3(4), November 2000.

[3] R. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in ORACLE. In *VLDB Conf.*, pages 659–664, 1998.

[4] A. Brodsky, C. Farkas, and S. Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *IEEE Trans. on Knowl. and Data Engg.*, 12(6):900–919, 2000.

[5] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, 1995.

[6] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[7] S. Chaudhuri and M. Vardi. Optimizing real conjunctive queries. In *PODS*, pages 59–70, 1994.

[8] V. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *PODS*, pages 155–166, 1999.

[9] D. Denning. Commutative filters for reducing inference threats in multilevel database systems. In *IEEE Symp. on Security and Privacy*, pages 134–146, 1985.

[10] C. Farkas and S. Jajodia. The inference problem: A survey. *SIGKDD Explorations*, 4(2), Mar. 2003.

[11] V. Gligor. Characteristics of role-based access control. In *ACM Symp. on Access Control Models and Technologies*, 1996.

[12] J. Goldstein and P. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *SIGMOD Conf.*, pages 331–342, 2001.

[13] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[14] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB Conf.*, pages 358–369, 1995.

[15] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[16] M. A. Harrison, M. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communication of the ACM*, 19(8)(Pages 461-471), August 1976.

[17] S. Jajodia, P. Samarati, M. Sapino, and V. Subrahmaniam. Flexible support for multiple access control policies. *ACM Trans. on Database Systems*, 26(4), June 2001.

[18] S. Jajodia and R. Sandhu. Towards a multilevel secure relational data model. In *SIGMOD Conf.*, pages 50–59, 1991.

[19] S. Jajodia and D. Wijesekera. Recent advances in access control models. In *IFIP Working Conference on Database and Application Security (DBSec)*, 2001.

[20] A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *ICDE*, pages 339–347, 1989.

[21] A. Motro. Panorama: A database system that annotates its answers to queries with their properties. *Journal of Intelligent Information Systems*, 7(1):51–73, Sept. 1996.

[22] A. Rosenthal and E. Sciore. View security as the basis for data warehouse security. In *Intl. Workshop on Design and Management of Data Warehouses (DMDW)*, 2000.

[23] A. Rosenthal and E. Sciore. Administering permissions for distributed data: Factoring and automated inference. In *IFIP 11.3 Working Conf. in Database Security*, 2001.

[24] A. Rosenthal, E. Sciore, and V. Doshi. Security administration for federations, warehouses, and other derived data. In *IFIP WG11.3 Conf. on Database Security*, 1999.

[25] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD Conf.*, pages 249–260, 2000.

[26] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *VLDB Conf.*, pages 318–329, 1996.

[27] R. Yerneni, C. Li, H. Garcia-Molina, and J. D. Ullman. Computing capabilities of mediators. In *SIGMOD Conf.*, pages 443–454, 1999.

[28] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD Conf.*, pages 105–116, 2000.