Dynamic Meta-level Access Control in SQL

Steve Barker

Dept Computer Science
King's College London
Strand, WC2R 2LS, UK
steve.barker@dcs.kcl.ac.uk

Abstract. Standard SQL is insufficiently expressive for representing many access control policies that are needed in practice. Nevertheless, we show how rich forms of access control policies can be defined within SQL when small amounts of contextual information are available to query evaluators. Rather than the standard, relational structure perspective that has been adopted for fine-grained access control, we consider instead the representation of dynamic fine-grained access control (DFMAC) policy requirements at the access policy level. We also show how DFMAC policies may be represented in SQL and we give some performance results for an implementation of our approach.

1 Introduction

SQL only permits limited forms of DAC (GRANT-REVOKE) and RBAC policies to be represented and only at coarse levels of granularity. A consequence of this limited expressive power is that many access control policy requirements, that need to be employed in practice, cannot be adequately represented in SQL. Of course, views can sometimes be used, in conjunction with SQL's sublanguages for DAC and RBAC representation, to express access control requirements. However, this combined approach does not provide a complete solution to the problem of SQL's limited support for access policy specification and it can introduce additional problems (e.g., the problem of a mass proliferation of view definitions if a user-based view of access control is adopted).

The problem of SQL's limited provision of language features for expressing access control policies (henceforth referred to as the AC representation problem) has been recognized ever since the first SQL standard was published. To address this problem, the language constructs that have been included in SQL, for representing access control requirements, have been progressively extended. However, the AC representation problem has, in a sense, become more acute because the gap between SQL's sublanguage for access control specification and what database practitioners need has grown wider. Moreover, we believe that the AC representation problem is a multi-faceted and wide-ranging problem that will not be adequately solved by small extensions to SQL's sublanguages for access control. Different aspects of the problem have emerged over time as it has become better understood and as applications of databases have changed. For instance, interest in the AC representation problem has been recently rekindled as a consequence of researchers investigating issues in database privacy; for that, various

V. Atluri (Ed.): DAS 2008, LNCS 5094, pp. 1–16, 2008.

approaches for representing *fine-grained access control (FGAC)* policies have been proposed (see, for example, [13], [17] and [19]). However, these approaches simply deal with a particular manifestation of the AC representation problem under standard assumptions (e.g., that access control policy information is relatively static).

Aspects of the AC representation problem have also been recognized and addressed by RDBMS vendors and relational database programmers. For example, Oracle's *Virtual Private Database (VPD)* [15] allows programmers to define functions, which may be written using PL/SQL, C or JAVA, to express access control requirements that are not expressible in standard SQL. More generally, for all forms of RDBMSs, database programmers will often write application programs to enhance SQL's provision and to thus enable richer forms of access control policies to be represented. However, neither of these approaches is especially attractive. As many researchers have observed, access control requirements must be expressed in high-level, declarative languages that have well-defined semantics, that permit properties of policies to be proven for assurance purposes, and that enable users to express access control requirements succinctly and in a way that makes it possible for them to understand the consequences of making changes to a policy. Conversely, it is inappropriate to use low-level languages that do not have well-defined semantics, that embed *ad hoc* and often hard-coded representations of policies in applications, that compromise attempts at formal verification of requirements, that make it difficult for security administrators to understand the consequences of policy change, and that thus makes policy maintenance a difficult task. Moreover, the programming-based approach requires that two languages (with very different semantics) be used and often results in complex forms of SQL query being executed inefficiently.

The work that we describe in this paper addresses an aspect of the AC representation problem that has not previously been considered in the context of SQL databases. Specifically, we address the issue of providing support for representing, in SQL, formally well-defined, dynamic fine-grained meta-level access control (DFMAC) policies for category-based access control models. By meta-level policy representations we mean representations of closed access control policies, open access control policies, and various forms of hybrid policies. As we allow for open policies, it follows that we admit negative authorizations (for expressing denials of access) as well as positive authorizations in our representation. In our approach, the meta-policies that are applicable to users can change dynamically and autonomously as a consequence of modifications to a database. To accommodate changing policies, we make use of contextual information; specifically, user identifiers and system clock time. By enabling DFMAC policies to be represented and implemented in SQL, we preserve the semantics of SQL and we are able to utilize existing RDBMS technology to implement DFMAC policies. Although FGAC policies have previously been discussed in the access control literature, to the best of our knowledge, no previous work has been described that is based on representing multiple forms of DFMAC policies directly in SQL. For our implementation, we use *query rewriting* [18]. Query rewriting for FGAC has recently been investigated by several researchers (see, for example, [17], [13], and [19]); the novelty of our rewriter stems from the focus we adopt on the exploitation of DFMAC information rather than FGAC information.

The rest of the paper is organized in the following way. In Section 2, a number of preliminary notions are described. In Section 3, we describe the general features of category-based access control models to which our approach may be applied. In Section 4, we describe policy representation in SQL. In Section 5, we explain how our query modifier is used to rewrite SQL queries to enforce DFMAC policies. In Section 6, we use a detailed example to explain more fully what is involved in our proposal. In Section 7, we discuss a candidate implementation for our approach and we give performance measures for this implementation. In Section 8, we discuss the related literature. Finally, in Section 9, conclusions are drawn, and further work is suggested.

2 Preliminaries

In this section, we briefly describe some key technical issues, to make the paper self-contained, to highlight the language that we propose for specification, and to explain the theoretical foundations of our approach. We assume that the reader is familiar with basic relational (SQL) terminology, like relation (table), attribute (column) and tuple (row). Otherwise, we refer the reader to [10].

In our approach, *tuple relational calculus (TRC)* [10] is used for specifying DFMAC policy requirements. For that, a many-sorted first order language is used over an alphabet Σ that includes:

- Countable sets of (uninterpreted) constants, variables, and n-ary relation symbols (some of which have a fixed interpretation);
- A functionally complete set of connectives;
- The singleton set of quantifiers: $\{\exists\}$;
- The comparison operators: $\{=, <, \leq, \neq, >, \geq\}$;
- The set of parenthetic symbols $\{(,), [,]\}$ and the punctuation symbols ‘,’ and ‘.’.

Rather than restricting attention to a minimal set of connectives that are functionally complete, we assume that policy specifiers will choose some subset of the 2^{2^n} n-ary connectives in 2-valued logic for representing DFMAC policies of interest.

The following sorts are of interest:

- A countable set $\mathcal{U}$ of user identifiers.
- A countable set $\mathcal{C}$ of category identifiers.
- A countable set $\mathcal{A}$ of named *actions* e.g., $SELECT$, $INSERT$, $UPDATE$, and $DELETE$.
- A countable set $\mathcal{T}$ of *table identifiers*.
- A countable set Θ of *time points*.

By tables, we mean base tables and views. On times, we assume a one-dimensional, linear, discrete view of time, with a beginning and no end point. That is, the model of time that we choose is a total ordering of time points that is isomorphic to the natural numbers. Clock times are an important form of contextual information. The function $\$current\_time$ is used to generate the current system clock time, a structured term from which more specific temporal information may be produced. For example,

$\$current\_time.year$, $\$current\_time.month$ and $\$current\_time.day$ can be used to, respectively, extract the current year, month and day from $\$current\_time$. Our focus is on user queries and hence the $SELECT$ action. The extension of what we propose to the case of update operations is straightforward.

A TRC query, expressed in terms of Σ, is of the following general form

$$\{\tau_1.A_1, \tau_2.A_2, \ldots, \tau_n.A_n : \mathcal{F}(\tau_1, \tau_2, \ldots, \tau_n, \overrightarrow{\tau_{n+1}}, \overrightarrow{\tau_{n+2}}, \ldots, \overrightarrow{\tau_m})\}$$

where $\tau_1, \tau_2, \ldots, \tau_n$ are tuple variables (not necessarily distinct), $A_1, A_2, \ldots, A_n$ are attributes, and

$$\mathcal{F}(\tau_1, \tau_2, \ldots, \tau_n, \overrightarrow{\tau_{n+1}}, \ldots, \overrightarrow{\tau_m})$$

is a subformula of TRC, expressed in terms of Σ, with $\tau_1, \tau_2 \ldots \tau_n$ as free variables in $\mathcal{F}$ and with bound variables $\overrightarrow{\tau_{n+1}}, \overrightarrow{\tau_{n+2}}, \ldots, \overrightarrow{\tau_m}$.

In our approach, rewritten queries are formulated with respect to a pair (Q, Π) where Q is an SQL query submitted for evaluation by a user u and Π is the DFMAC policy information that is applicable to u at the time at which u submits its query. For query evaluation, Q and Π are combined to generate a rewritten query Q'.

The model of a DFMAC policy Π, which we assume to be expressed in a language as expressive as hierarchical programs [14], includes a set of *authorizations*. An authorization, in turn, is expressed in terms of *permissions* or *denials*, which may be expressed conditionally in terms of contextual information. A permission is expressed in terms of a 2-place predicate $per(a, o)$ where a is an access privilege and o is a database object (a table); the meaning of $per(a, o)$ is that the a access privilege may be exercised on o. A denial is expressed in terms of a 2-place predicate $den(a, o)$ where a is an access privilege, o is a database object (a table), and $den(a, o)$ denotes that the a access privilege cannot be exercised on o. An authorization is a triple $auth(u, a, o)$ with the semantics that user u may exercise the access privilege a on o.

A Herbrand semantics [1] is applicable to the databases that we consider. In this context, the intended model of a relational database Δ is the set of ground instances of atoms from the Herbrand Base of Δ, $HB(\Delta)$, that are true in the unique least Herbrand model of Δ. In our approach, the set of ground atomic consequences that a user u may retrieve (i.e., $select$) from a database Δ to which the DFMAC policy Π applies is expressed thus:

$$\{A : A \in HB(\Delta) \ \wedge \ \Delta \models A \ \wedge \ authorized(u, select, A)\}.$$

3 Category-Based Access Control

In this section, we briefly discuss the essential features of a range of category-based access control models to which our approach can be applied. We also briefly describe goal-oriented access control for which DFMAC policies are applicable.

In contrast to DAC/GRANT-REVOKE models, a number of access control models are based on the idea of categorizing users on some general criterion. In this context, user-category assignment is adopted as a basis for access control. That is, users are assigned to categories and permissions are assigned to categories of users too. A user

acquires the permissions assigned to a category when the user is assigned to the category. Assigning users and permissions to categories raises the level of abstraction (relative to lower-level policies, like DAC policies) and reduces the number of permission and denial assignments that need to be specified. The downside of the approach is that it is often necessary to express that, for example, exceptions apply to certain users within a general category. Hence, negative authorizations need to be specified as well as positive authorizations. However, the need for conflict resolution strategies then arises.

Category-based access control models include groups (with discretionary assignment of users to a group), sets of users categorized according to their security clearance (as in MAC models [6]), sets of users that are categorized according to the job function that they perform in an organization (as in RBAC models [11]), sets of users catagorized according to a discretely defined trust level, and sets of users that are categorized according to the combination of an ascribed and action status (as in ASAC [4]).

Category-based access control models have a fairly common definition of an authorization (u, a, o) and meta-policies are defined in terms of this general interpretation of authorizations. In the case of closed meta-policies for category-based access control models: permissions (each of which is a pair (a, o) where a is an access privilege and o is a database object) are assigned to a category c; a user u is assigned to a category c; when the user is assigned to a category c the user may exercise a privilege a on object o iff the permission (a, o) is assigned to c. For an open meta-policy the authorization (u, a, o) holds iff u is assigned to category c and there is no denial of the permission (a, o) to c. A variety of additional meta-policies may be defined in terms of the open and closed meta-policies. For example, a "denials override" policy may be used to express that the authorization (u, a, o) holds iff u is assigned to category c, the permission (a, o) is assigned to c and there is no denial of the permission (a, o) to c category users.

DFMAC polices are required to provide fine-grained access control where "fine-grained" information is interpreted as fine-grained at the meta-policy level rather than at the level of data. That is, DFMAC policies are applicable when security administrators want to be able to specify that a particular meta-policy (a closed policy, an open policy, a hybrid policy, etc) is to apply to different categories of users in different contexts. The different contexts may be temporally-based (e.g., an open policy is to apply on weekdays but not weekends) or they may be location-based or the policy that applies may depend on the contents of the database.

DFMAC policies are especially important when goal-oriented access control requirements need to be represented. In goal-oriented access control, organizational and individual goals may change as a consequence of the occurrence of events and this, in turn, may cause access control policy requirements to change. For example, an organization may wish to restrict access to information on "special offers" to the category of $preferred$ customers, but may need to change dynamically and autonomously this policy constraint to allow access to all customers if sales figures are "poor".

4 Access Policy Representation

For the representation of our approach, we use four base tables; each of the tables has a fixed purpose. We use a table named *category* to record which users of a DBMS

are assigned to which categories and we use a table named *policy* to store meta-level access control information that our query rewrite procedure uses at runtime to evaluate a user's access request. We also use a table named pca (shorthand for permission category assignment) and a table name dca (shorthand for denial category assignment) to, respectively, store tuples that represent the permissions and denials that apply to categories of users that may request to perform some action on some database object. The $category$, $policy$, pca and dca tables may include conditions that must be satisfied in order for a category, policy, permission or denial to apply. The query rewrite procedure also uses this information at runtime in the evaluation of a user's access request.

The four tables and brief details of their intended semantics may be described thus:

- $category(userID, catID, CC_u) : \langle u, c, cc_u \rangle \in category$ iff $u \in \mathcal{U}$, $c \in \mathcal{C}$ and u satisfies the cc_u condition for assignment to the category c.
- $policy(catID, action, objectID, PC_t) : \langle c, a, t, p \rangle \in policy$ iff $c \in \mathcal{C}$, $a \in \mathcal{A}$, $t \in \mathcal{T}$, and p is a boolean condition defined in terms of pca, dca, or other tables in the database.
- $pca(catID, action, objectID, pca\_condition) : \langle c, a, t, q_{pca} \rangle \in pca$ iff $c \in \mathcal{C}$, $a \in \mathcal{A}$, $t \in \mathcal{T}$, q_{pca} is a boolean condition, and the permission (a, t) applies to c users if q_{pca} is true.
- $dca(catID, action, objectID, dca\_condition) : \langle c, a, t, q_{dca} \rangle \in dca$ iff $c \in \mathcal{C}$, $a \in \mathcal{A}$, $t \in \mathcal{T}$, q_{dca} is a boolean condition, and the denial (a, t) applies to c users if q_{dca} is true.

The set of authorizations $\mathcal{AUTH}$ that is defined by a DFMAC policy is expressed in terms of the core set of tables that are described above. For example, we have for a closed DFMAC policy (expressed using TRC and ignoring PC_t conditions):

$$\begin{aligned}\mathcal{AUTH} = \{\langle t_1.userid, t_2.action, t_3.objectid \rangle : category(t_1) \\ \wedge\, t_1.cc_u \ \wedge\ [policy(t_2) \ \wedge\ t_1.catid = t_2.catid \\ \wedge\, \exists t_3[pca(t_3) \ \wedge\ t_3.catid = t_1.catid \\ \wedge\, t_3.catid = t_2.catid \ \wedge\ t_3.pca\_condition]]\}.\end{aligned}$$

A number of points should be noted. We envisage access policy information being represented in TRC before being transferred into SQL for implementation. Any number of meta-level policies can be expressed in the same way as the closed policy information that is represented above. When a category c of users can access information about all of the attributes in a table t then the subset of the rows in the table that a c category user can access is expressed as a value of PC_t. When only a subset of the attributes of a base table is accessible to a category of users then access to this data is via a view v. However, the subset of tuples accessible to the category of users via v is also defined as a value of PC_t. We make the simplifying assumption that $(catID, objectID)$ is the primary key for $policy$. In practice, an extra attribute (e.g., $policyID$) would be used to allow for multiple DFMAC policies to apply to different categories of users, for different actions on different objects when different forms of contextual information apply. Access control information in the form of certificates can also be naturally accommodated in our approach. Notice too that we represent DFMAC policy information at the

meta-level level and at the level of permissions and denials. That is, conditions on the meta-level policy that applies at any instance of time may be specified in $category$ and $policy$ and conditions that define when permissions and denials are applicable are expressed in pca and dca, respectively. The values that are admitted for CC_u, PC_t, pca and dca are boolean expressions that are formulated using SQL and are automatically appended to a user's SQL query in the process of query modification.

5 Query Modification

Recall that an SQL query is of the following basic form:

$$SELECT\ A_1, \ldots, A_m$$
$$FROM\ t_1, \ldots, t_n$$
$$WHERE\ Q;$$

Here, $A_1, \ldots, A_m$ are attributes that define the structure of the required result relation and $t_1, \ldots, t_n$ are the tables on which the condition Q is evaluated. Of course, an SQL $SELECT$ statement can be expressed in terms of other constructs (e.g., aggregate functions) but these elements are not important in the discussion that follows.

In our approach, the query modifier automatically appends references to the $category$ and $policy$ tables to a user's query when the user submits the query to an RDBMS for evaluation. The user has a unique $userID$, which is accessible to the query modifier as soon as a user has been authenticated (a range of different authentication methods may be used). In order for the user's query to be performed, the user must be authorized, according to the information stored in $category$ and $policy$, to access each of the $t_1 \ldots t_n$ tables (or views) that are referred to in Q; otherwise, the user's query will automatically fail. The reason for this should be clear: the join of $t_i, \ldots, t_n$, $t_i \bowtie \ldots \bowtie t_n$, cannot be performed for user u if access to some t_i $(1 \leq i \leq n)$ is not authorized for u.[1] For a user that submits a query Q, the CC_u condition (if any) is appended to Q. For a user that is permitted to access each t_i $(1 \leq i \leq n)$ table that is referred to in Q, the condition on accessing t_i is appended to the query as a conjunct PC_{t_i} $(1 \leq i \leq n)$ where PC_{t_i} denotes the condition from $policy$ that holds on u's access to t_i as a consequence of u's assignment to a category c. Hence, the rewritten query is of the following general form (where $t_i (1 \leq i \leq n)$ is the table name for t_i):

$$SELECT\ A_1, \ldots, A_m$$
$$FROM\ t_1, \ldots, t_n, category, policy$$
$$WHERE\ Q$$
$$AND\ category.userID = \$userID$$
$$AND\ category.categoryID = policy.categoryID$$
$$AND\ category.CC_u$$
$$AND\ (policy.objectID = t_1\ AND\ PC_{t_1}$$
$$\ldots$$
$$AND\ policy.objectID = t_n\ AND\ PC_{t_n};$$

[1] cf. the null-based semantics used in [13].

A query Q is rewritten with respect to a DFMAC policy Π to generate a query Q' for evaluation. If the tables $t_1, \ldots, t_n$ are interpreted as sets of tuples then for Q' to succeed for user u we require that $t_1 \cup \ldots \cup t_n \models Q$ and $(u, select, t_i) \in \mathcal{AUTH}$, $\forall t_i$ such that $t_i \in \{t_1 \ldots t_n\}$. It should also be noted that $Q' \sqsubseteq Q$ holds where $\sqsubseteq$ is a query containment operator [1].

When an SQL query is expressed as a collection of n ($n \in \mathbb{N}$) nested subqueries $Q_1, \ldots, Q_n$ in the form

$$Q_1(Q_2(Q_3(Q_4 \ldots (Q_n))))$$

then the rewritten query (ignoring the CC_u condition) is of the form

$$Q_1\ AND\ C_1(Q_2\ AND\ C_2(Q_3\ AND\ C_3(Q_4\ AND\ C_4 \ldots (Q_n\ AND\ C_n))))$$

where C_i ($1 \leq i \leq n$) is the conjunction of PC_{t_i} conditions that apply to Q_i.

When an SQL query is expressed in terms of the union of two subqueries ($Q_1 \cup Q_2$), or difference ($Q_1 - Q_2$) or interersection ($Q_1 \cap Q_2$) then, respectively, the rewritten queries are of the form $Q_1\ AND\ C_1 \cup Q_2\ AND\ C_2$, $Q_1\ AND\ C_1 - Q_2\ AND\ C_2$ and $Q_1\ AND\ C_1 \cap Q_2\ AND\ C_2$ where C_i ($i \in \{1, 2\}$) is the conjunction of PC_{t_i} conditions that apply to Q_i.

Next, we consider various forms of meta-level specification that may be represented in SQL and that are useful for representing DFMAC policy requirements. Recall that these different meta-policies are stored in the $policy$ table as values of the attribute PC_t and are defined in terms of pca and dca.

The condition for a closed policy on table t_i may represented, as a value for PC_{t_i}, in a fragment α of SQL, thus:

$$EXISTS(SELECT\ *\ FROM\ pca$$
$$WHERE\ policy.catID = pca.catID\ AND\ pca.action = \text{“}select''$$
$$AND\ pca.object = policy.objectID\ AND\ pca\_condition).$$

If the conjunction of conditions for a closed policy to apply are true then a user u assigned to category c will be authorized to select the requested tuples from a table t (where t is the $objectID$) iff c category users are recorded in pca as being permitted to exercise the select privilege on t and the condition $pca\_condition$ on the permission applicable to c category users of t evaluates to true at the time of the access request.

The policy condition for an open policy on table t_i may be represented as a fragment β of SQL and as a value for PC_{t_i}, thus:

$$NOT\ EXISTS(SELECT\ *\ FROM\ dca$$
$$WHERE\ policy.catID = dca.catID\ AND\ dca.action = \text{“}select''$$
$$AND\ dca.object = policy.objectID\ AND\ dca\_condition).$$

In the case of an open policy, a user u assigned to category c will be authorized to select the requested tuples from a table t iff c category users are not currently recorded in dca as being prohibited from exercising the select privilege on t. Here, t is the $objectID$ and the $dca\_condition$ must evaluate to true in order for the prohibition on t to apply to c category users at the time of an access request.

Recall that one of our aims is to define complex policies in terms of more primitive forms. For example, a "denials override" policy can be defined as the conjunction $\alpha\ AND\ \beta$. However, multiple forms of meta-policies can be similarly constructed using the specification language that we admit. On that, policy authors will make use of a subset of the 2^{2^n} n-ary operators in 2-valued logic for specifying DFMAC policies. For instance, a conditioned disjunction operator $[c, \alpha, \beta]$ (where $[c, \alpha, \beta] \equiv c \wedge \neg\phi_1 \vee \neg c \wedge \phi_2$) may be used with c denoting $sales > 1000$ to specify that a closed policy applies when $sales$ are greater than 1000 and an open policy otherwise. Similarly, $(c \rightarrow \phi_1) \wedge (\neg c \rightarrow \phi_2) \equiv (\neg c \vee \phi_1) \wedge (c \vee \phi_2)$ can be used to express an IF-ELSE condition. Notice too that by combining subconditions, conditions of arbitrary complexity may be generated to enable expressive forms of policy algebras to be defined.

In addition to meta-level representation via the $policy$ table, policy information that applies to the permissions and denials of access, expressed via pca and dca, respectively, are expressed as values of $pca\_condition$ and $dca\_condition$. As in the case of the conditions that apply to meta-level information included in $policy$, the values of $pca\_condition$ and $dca\_condition$ are boolean conditions expressed in SQL.

To conclude this section, we outline the 4-step procedure for the query rewrite method that we use for DFMAC policy enforcement (see also Section 7):

Step 1: Rewrite a user's query Q to give the following modified query Q_1:

$$
\begin{array}{l}
SELECT\ A_1, \ldots, A_m \\
FROM\ t_1, \ldots, t_n, category, policy \\
WHERE\ Q \\
AND\ category.userID = \$userID \\
AND\ category.categoryID = policy.categoryID \\
AND\ category.CC_u \\
AND\ policy.objectID = t_1\ AND\ PC_{t_1} \\
\ldots \\
AND\ policy.objectID = t_n\ AND\ PC_{t_n};
\end{array}
$$

Step 2: Expand Q_1 by PC_{t_i} $(\forall i \in \{1 \ldots n\})$ to give Q_2.
Step 3: Expand Q_2 by $pca\_condition$ or $dca\_condition$ to give Q'.
Step 4: Evaluate Q'.

It should be clear from the rewrite method and previous discussion that a query expansion approach is used by us. That is, if $\rightsquigarrow_i$ is read as "expands to using i" (where i is some source of DFMAC information) then the rewrite sequence is

$$[Q \rightsquigarrow_c Q_1, Q_1 \rightsquigarrow_t Q_2, Q_2 \rightsquigarrow_m Q']$$

where c denotes the category condition, t denotes the table access conditions and m denotes the meta-policy information. Moreover, $Q\ \wedge\ c\ \wedge\ t\ \wedge\ m \rightsquigarrow Q'$.

6 DFMAC Policy Examples

Before we consider the implementation of the 4-step query modification procedure, we provide a (quite general) example to illustrate our approach. Our example is based

on a variant of Date's *supplier-part-project* database [10] and includes the following relational schemes:

$$supplier(s\#, sname, status, scity),$$
$$part(p\#, pname, color, unitcost, stock),$$
$$project(j\#, jname, jcity),$$
$$spj(s\#, p\#, j\#, qty).$$

Consider next the following DFMAC policy requirements on $part$:

During the month of March (in any year), a closed access control policy on the retrieval of part information is to apply to users that are categorized as $preferred$ *but only if the* $stock$ *is greater than 1000 units and the user has not been suspended. At all other times, an open policy on retrieving information from* $part$ *applies to* $preferred$ *users, but only if the stock level of an item is less than 450 units and unit cost is greater than 0.9.*

In TRC, the full expression of the policy information is:

$$\{\langle t_1.p\#, t_1.pname, t_1.color, t_1.unitcost, t_1.stock\rangle : part(t_1) \wedge \exists t_2[category(t_2) \wedge t_2.userid = \$userid \exists t_3[policy(t_3) \wedge t_2.policyid = t_3.policyid \wedge [\$currenttime.month = \text{“}march'' \wedge t_1.stock > 1000 \wedge \exists t_4[pca(t_4) \wedge t_3.catid = t_4.catid \wedge t_4.action = \text{“}select'' \wedge t_3.objectid = t_4.objectid \wedge \neg\exists t_5[suspended(t_5) \wedge t_5.userid = t_2.userid]]] \vee [\$currenttime.month \neq \text{“}march'' \wedge t_1.stock < 450 \wedge t_1.unitcost > 0.9 \wedge \neg\exists t_6[dca(t_6) \wedge t_3.catid = t_6.catid t_6.action = \text{“}select'' \wedge t_3.objectid = t_6.objectid]]]\}.$$

Notice that two exclusive disjuncts are required to express the DFMAC policy requirements. For the closed policy, which applies in March, the TRC translates into the following fragment γ of SQL code:

$$\$current\_time.month = \text{“}march'' \; AND \; stock > 1000$$
$$AND \; EXISTS(SELECT *$$
$$FROM \; pca$$
$$WHERE \; policy.catID = pca.catID \; AND \; pca.action = \text{“}select''$$
$$AND \; pca.object = policy.objectID \; AND \; pca\_condition)$$

where the $pca\_condition$ is represented by the following fragment δ of SQL code:

$$NOT \; EXISTS(SELECT *$$
$$FROM \; suspended$$
$$WHERE \; suspended.userID = category.userID).$$

That is, a $preferred$ user is authorized in March to access tuples in $part$ where the $stock$ value is greater than 1000 if there is a permission to allow u access on $part$ and u is not a suspended user. For the open policy, the required fragment ϵ of SQL code is as follows:[2]

$$
\begin{array}{l}
NOT\ \$current\_time.month = \text{“}march''\\
AND\ stock < 450\ AND\ unitcost > 0.9\\
AND\ NOT\ EXISTS(SELECT\ *\ FROM\ dca\\
WHERE\ policy.catID = dca.catID\ AND\ dca.action = \text{“}select''\\
AND\ dca.object = policy.objectID).
\end{array}
$$

The PC_{part} value that is stored in $policy$ is the disjunction of the fragments $\gamma\ \wedge\ \delta$ and ϵ; the SQL fragment δ is stored in pca.

Next, consider the following query Q for user κ with $preferred$ status:

$$
\begin{array}{l}
SELECT\ p\#\\
FROM\ part\\
WHERE\ unitcost > 0.25
\end{array}
$$

but where $preferred$ users are only permitted to see a subset (view) of $part$ such that $unitcost \geq 0.5$ (thus making the $unitcost > 0.25$ condition redundant) and only if these users are located in Europe (as recorded in a table named $region$).

It follows, from the discussion above, that the query rewriter that we use will generate the following modified form of Q for κ:

$$
\begin{array}{l}
SELECT\ p\#\\
FROM\ part, category, policy\\
WHERE\ unitcost > 0.25\\
AND\ category.userID = \$userID\\
AND\ category.categoryID = policy.categoryID\\
AND\ policy.objectID = \text{“}part''\ AND\ unitcost \geq 0.5\\
AND\ EXISTS(SELECT\ *\ FROM\ region\\
\quad WHERE\ \$userID = region.ID\ AND\ region.name = \text{“}europe'')\\
AND\ \$current\_time.month = \text{“}march''\ AND\ stock > 1000\\
AND\ EXISTS(SELECT\ *\ FROM\ pca\\
\quad WHERE\ policy.catID = pca.catID\\
\quad AND\ pca.action = \text{“}select''\ AND\ pca.object = policy.objectID\\
\quad AND\ NOT\ EXISTS\ (SELECT\ *\\
\qquad FROM\ suspended\\
\qquad WHERE\ suspended.userid = category.userid))\\
OR\ NOT\ \$current\_time.month = \text{“}march''\\
AND\ stock < 450\ AND\ unitcost > 0.9\\
\quad AND\ NOT\ EXISTS(SELECT\ *\ FROM\ dca\\
\quad WHERE\ policy.catID = dca.catID\\
\quad AND\ dca.action = \text{“}select''\ AND\ dca.object = policy.objectID);
\end{array}
$$

[2] Notice that there is no $dca\_condition$ value that is applicable.

Thus:

If κ submits its query on March 1st (at which point κ has $preferred$ status and is located in Europe) then κ is authorized to access part numbers for all tuples in $part$ where $unitcost \geq 0.5$ but only if $stock$ values are greater than 1000 units and a permission holds for κ that is not overridden as a consequence of κ being a suspended user at the time of κ's access request. In contrast, if, on the 1st April (say), κ submits its query then it can access all part numbers where $unitcost > 0.9$ provided that $stock$ level is less than 450 units and κ is not explicitly prohibited from accessing this information.

For the example DFMAC policy requirements described above, the SQL query that is generated is quite complex and requires that a number of subqueries be used. However, for many practical queries the rewritten form will be much simpler. For all approaches that are used for access control there is an overhead involved in checking access constraints, and there will always be a trade-off between complex policy representation and query efficiency. In the next section, we provide arguments to suggest that many queries, that require the representation of some quite complex DFMAC policy requirements, can be evaluated without significant enforcement overheads.

7 Practical Considerations

In this section, we describe the testing of an implementation of our approach and we give some performance measures.

For the testing, we use large-scale versions of the tables that we previously described. Specifically, we use the *supplier-part-projects* database with (of the order of) 100000 tuples in each of the $part$, $supplier$ and $project$ tables and with (of the order of) 500000 tuples in the spj table.

For implementation, we use PostgreSQL 8.3 [16]. We use rewrite rules to transform a query tree for a user query Q into a modified form Q' that incorporates access control information that may be stored in $category$, $policy$, pca or dca. Our testing is performed using a 1.9GHz AMD Athlon X2 Dual-Core machine (with a 128KB Level 1 data cache, a 512KB level 2 cache, and 1GB of memory) running Red Hat Linux 7.3. The results are generated by using the PostgreSQL $timing$ function.

The principal purpose of our testing is to determine the extent to which the DFMAC policy information, which is added to Q to generate Q', affects performance. No effort was made to tune the implemented system (to avoid the results becoming set-up specific). We perform our tests using data and queries for "expensive case" evaluation. Hence, we also test the scalability of the approach.

An example of the type of query that we use is: *retrieve all suppliers names for suppliers that supply red parts to no project in London*, to wit:

$$\{\langle t_1.sname\rangle : supplier(t_1) \wedge \exists t_2[part(t_2) \wedge t_2.color = \text{“}red\text{''} \wedge \neg\exists t_3[project(t_3) \wedge t_3.jcity = \text{“}london\text{''} \wedge \exists t_4[spj(t_4) \wedge t_1.s\# = t_4.s\# \wedge t_2.p\# = t_4.p\# \wedge t_3.j\# = t_4.j\#]]]\}$$

Various policy combinations were tested with various queries for a single user that is assigned to a single category to which $SELECT$ access is defined in $policy$ on

the tables in the *supplier-parts-projects* database. We also performed some queries that generate large numbers of tuples in intermediate tables in the process of query evaluation. For example, we perform a query that involves computing the cartesian product of three subsets of the tuples in $part$ (each subset having a cardinality of 100). The meta-policy information, based on the example from Section 6, is stored in $policy$ and the pca conditions and dca conditions that are defined in terms of $suspended$ are, respectively, stored in pca and dca.

The key measure for our implementation is the overheads that are incurred as a consequence of adding the DFMAC policy information in the process of query evaluation. On that, we have observed typical extra overheads of the order of 10-15% (for the majority of our test queries). For example, for the query above, the query evaluation time (averaged over 10 runs) is 1.9s. In contrast, when the DFMAC policy information, described in Section 6, is compiled into the query, the average time is 2.2s. (The overheads involved in rewriting are negligible.) Similar results were generated for a range of queries on the *supplier-part-project* database. Nevertheless, a test of a hybrid policy with an expensive subquery evaluation that involves accessing an instance of $suspended$ with 15000 tuples did push the DFMAC overhead up to 26% over the non-DFMAC case. For the processing of this query (with $\mid suspended \mid \; = 15000$) the computational overheads are pushed towards a bound of unacceptability. However, if this type of query were performed frequently, in practice, on a table of similar cardinality then the possibility of optimizing access to $suspended$ would be considered by a DBA. Clearly, it is always possible to find worst case scenarios that incur high costs in terms of processing DFMAC information. In all cases, a DBA must consider the trade-off between automatic DFMAC policy enforcement and the computational overheads that are incurred as a consequence of evaluating queries that are rewritten to incorporate DFMAC policy information.

8 Related Work

In this section, we describe the literature that relates to our approach. We first describe work from the access control community that is concerned with the specification of flexible forms of access control information for protecting databases. Thereafter, we consider how our approach relates to work that has been focused on the specific issue of query rewriting on SQL databases for implementing FGAC policy requirements.

The importance of developing access control models, in terms of which flexible forms of access control policies may be defined, has long been recognized. The work by Bertino, Jajodia and Samarati [9] is especially significant in this respect. In [9], a well-defined authorization model that permits a range of discretionary access policies to be defined on relational databases is described. However, the emphasis in [9] is principally on discretionary policies rather than the range of DFMAC policies for category-based access control models that we have considered. The importance of using contextual information in access control for helping to protect databases is also well-known. For example, the idea of utilizing temporal restrictions on access to information in databases has been discussed by Bertino et al. [7], and an event-based approach for flexible access to databases that is based on triggers has been presented in [8]. An extension of

the approach, which allows for the dynamic enabling and disabling of roles, has been described in [12]. However, these approaches are not concerned with the direct representation of DFMAC policies in SQL, they do not consider the range of category-based models that we do, and they are not concerned with query modification. We also note that an approach for the representation of flexible forms of access control policies for deductive databases has been described [2,3] as well as flexible specifications of access control meta-policies when contextual factors like system clock times may be taken into account [5]. However, the approaches that are described in [3],[2], and [5] are theoretical and cannot be naturally used with SQL databases (not least because query evaluation with respect to policy specification is tied to the operational semantics that are used in deductive databases and constraint databases).

On the issue of using query rewriting for access control, we discuss three recent contributions to the literature: the work by Rizvi et al. [17], Wang et al. [19], and LeFevre et al. [13]. The work by Rizvi et al. [17] has certain similarities with ours in that context is taken into account when evaluating access requests. However, the key contribution of Rizvi et al.'s work is to define validity rules that control access to data via parameterized views. A user's access request can be performed if and only if the access is consistent with the validity rules that apply to the access; otherwise, the user's query is rejected. Rizvi et al.'s work is concerned with FGAC policies at the level of data whereas our approach is focused on fine-grained access control from a policy-level perspective. It should also be noted that Rizvi et al.'s concern is with developing what the authors call a "non-Truman" approach to database access control. The issue of distinguishing queries that users can execute from those that they cannot execute, to ensure that non-Truman databases can be supported, is a different aspect of the AC representation problem than we have addressed. Rizvi et al.'s motivation for non-Truman databases is what they suggest to be possible misinterpretations of query answers that arise as a consequence of security restrictions. However, it is not clear that such confusions are exhibited by users. Moreover, Rizvi et al.'s work aims to resolve "inconsistencies" between what a user "expects" and what a system returns (in terms of answers to a query). These terms are not well defined by Rizvi et al. and it is not clear that rejecting queries as "inconsistent" is what users would "expect". A number of more specific problems apply to Rizvi et al.'s approach. For example, legitimate forms of queries can be rejected and the query validation problem is undecidable, in general, for the inference rules that are proposed for identifying acceptable queries.

The work by Wang et al. [19], is also concerned with FGAC for relational databases. More specifically, Wang et al. consider the problem of defining and demonstrating the correctness of FGAC enforcement. In contrast, our approach is concerned with DFMAC policies. Moreover, our approach is correct in the sense that the set of atomic consequences that are accessible by a user of an SQL database are those that SQL computes to satisfy the definition of correctness that we specified in Section 2. That is, the set of ground atomic consequences that a user u may retrieve (i.e., $select$) from a database Δ to which the DFMAC policy Π applies is:

$$\{A : A \in HB(\Delta) \ \wedge \ \Delta \models A \ \wedge \ authorized(u, select, A)\}.$$

We have not considered the richer interpretation of correctness that Wang et al. consider, but that has not been the focus in this paper.

The work that we have described is perhaps closest is spirit to Lefevere et al.'s work [13]. Like LeFevre et al., we make use of a high-level specification language for representing policy requirements and we translate this specification into SQL for implementation. However, the specification language that we use is tuple relational calculus (rather than P3P) and our concern is with access control policies for SQL database in general (rather than privacy policies for Hippocratic databases). What is more, our concern is with the representation and processing of very different types of meta-level information because we do not interpret fine-grained access control at the data level as LeFevre et al. do. Instead, our concern has been to represent DFMAC policies. As such, we address a different aspect of the AC representation problem (as we explained in Section 1). The work in [13] is based on a nullification-based semantics (i.e., a null value is substituted for a data value that should remain private); this semantics may be appropriate in the privacy context, but is less obviously so in the DFMAC case.

In more general terms, it should be noted that our approach is focused on DFMAC policies for category-based access control models, specifically. By focusing on category-based models, the potential problem of view proliferation is much more manageable than it is when a user-based view is adopted (as in, for example, [17]). Neither [17] nor [13] identify well defined access control models to which their approaches apply. Moreover, negative authorizations are not considered and so neither [17] nor [13] discusses issues like policy overriding. It is also worth noting that, unlike [13], [17] and [19], our concern is with dynamic and autonomous $changing$ of access control policy requirements.

9 Conclusions and Further Work

The contributions that we have described in this paper can be summarized in the following way: we introduced an approach for dynamic, fine-grained access control policy representation (in tuple relational calculus and SQL) that differs from related work in terms of its focus; we demonstrated how DFMAC policy information may be used by a query modifier for enforcing access control policy requirements; and we discussed an implementation and performance results for a real application of the approach. The approach that we have described can be applied to various category-based access control models and, as such, is quite general. As far as we are aware the particular aspect of the AC representation problem that we have considered (the representation and implementation of DFMAC policy requirements) has not been previously addressed in the literature on representations of access control policy requirements in SQL.

In future work, we intend to investigate yet finer grained DFMAC policy representations where, for example, DFMAC policies may be expressed on individual columns within a table. We also want to consider extending our approach to accommodate additional information of relevance to extended forms of DFMAC policies (e.g., DFMAC policies that include specifications of obligations on users and constraints on policy specifications). The efficient implementation of these forms of extended DFMAC policies in SQL is also a matter for further work as is an investigation of the feasibility of combining DFMAC and FGAC policies proposed by other authors.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. Barker, S.: Access Control for Deductive Databases by Logic Programming. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 54–69. Springer, Heidelberg (2002)
3. Barker, S.: Protecting deductive databases from unauthorized retrieval and update requests. Journal of Data and Knowledge Engineering 23(3), 231–285 (2002)
4. Barker, S.: Action-status access control. In: SACMAT, pp. 195–204 (2007)
5. Barker, S., Stuckey, P.: Flexible access control policy specification with constraint logic programming. ACM Trans. on Information and System Security 6(4), 501–546 (2003)
6. Bell, D.E., LaPadula, L.J.: Secure computer system: Unified exposition and multics interpretation. MITRE-2997 (1976)
7. Bertino, E., Bettini, C., Ferrari, E., Samarati, P.: An access control model supporting periodicity constraints and temporal reasoning. ACM TODS 23(3), 231–285 (1998)
8. Bertino, E., Bonatti, P., Ferrari, E.: TRBAC: A temporal role-based access control model. In: Proc. 5th ACM Workshop on Role-Based Access Control, pp. 21–30 (2000)
9. Bertino, E., Jajodia, S., Samarati, P.: A flexible authorization mechanism for relational data management systems. ACM Trans. Inf. Syst. 17(2), 101–140 (1999)
10. Date, C.: An Introduction to Database Systems. Addison-Wesley, Reading (2003)
11. Ferraiolo, D.F., Sandhu, R.S., Gavrila, S.I., Kuhn, D.R., Chandramouli, R.: Proposed nist standard for role-based access control. ACM Trans. Inf. Syst. Secur. 4(3), 224–274 (2001)
12. Joshi, J., Bertino, E., Latif, U., Ghafoor, A.: A generalized temporal role-based access control model. IEEE Trans. Knowl. Data Eng. 17(1), 4–23 (2005)
13. LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., DeWitt, D.J.: Limiting disclosure in hippocratic databases. In: VLDB, pp. 108–119 (2004)
14. Lloyd, J.: Foundations of Logic Programming. Springer, Heidelberg (1987)
15. Oracle. Oracle 11g, `http://www.oracle.com`
16. PostgreSQL 8.3: User Manual, `http://www.postgresql.org/docs/`
17. Rizvi, S., Mendelzon, A.O., Sudarshan, S., Roy, P.: Extending query rewriting techniques for fine-grained access control. In: SIGMOD Conference, pp. 551–562 (2004)
18. Stonebraker, M., Wong, E.: Access control in a relational data base management system by query modification. In: Proc. 1974 Annual Conf (ACM/CSC-ER), pp. 180–186 (1974)
19. Wang, Q., Yu, T., Li, N., Lobo, J., Bertino, E., Irwin, K., Byun, J.-W.: On the correctness criteria of fine-grained access control in relational databases. In: VLDB, pp. 555–566 (2007)