Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

A fine-grained access control model for relational databases*

Jie SHI, Hong ZHU‡

(*College of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*)

E-mail: {shijie1123, whzhuhong}@gmail.com

Received July 29, 2009; Revision accepted Jan. 11, 2010; Crosschecked May 31, 2010

Abstract: Fine-grained access control (FGAC) must be supported by relational databases to satisfy the requirements of privacy preserving and Internet-based applications. Though much work on FGAC models has been conducted, there are still a number of ongoing problems. We propose a new FGAC model which supports the specification of open access control policies as well as closed access control policies in relational databases. The negative authorization is supported, which allows the security administrator to specify what data should not be accessed by certain users. Moreover, multiple policies defined to regulate user access together are also supported. The definition and combination algorithm of multiple policies are thus provided. Finally, we implement the proposed FGAC model as a component of the database management system (DBMS) and evaluate its performance. The performance results show that the proposed model is feasible.

Key words: Fine-grained access control, Database security, Prohibition, Multiple policies
doi:10.1631/jzus.C0910466 **Document code:** A **CLC number:** TP309

1 Introduction

Fine-grained access control (FGAC) in relational databases has drawn considerable attention from the database community due to the requirements of privacy preserving (Agrawal *et al.*, 2002; 2005; Rizvi *et al.*, 2004; Bertino and Sandhu, 2005; Bertino *et al.*, 2005; Byun *et al.*, 2005) and Internet-based applications (Jain 2004; Dwivedi *et al.*, 2005). FGAC governs data access to a table at the granularity of rows, columns, and even individual cells. The traditional access control in relational databases, however, provides only a way to restrict data access at coarse granularity; namely, it allows access to all rows of a table or none at all.

There have been many approaches to specifying and enforcing FGAC policies in relational databases including, for example, query modification in INGRES (Stonebraker and Wong, 1974), virtual private database (VPD) in Oracle (Oracle Corporation, 2005), and the recent work on Hippocratic databases (Agrawal *et al.*, 2002; LeFevre *et al.*, 2004). FGAC is implemented mainly by query modification (Stonebraker and Wong, 1974). Before being processed, user queries are transparently modified to ensure that users cannot access what they are not authorized to access.

While most research has focused on the enforcement of FGAC, there is little research on FGAC models which provides the specification of FGAC policies and guides the implementation of FGAC. To support privacy-preserving requirements, Agrawal *et al.* (2005) proposed a FGAC model which extends structured query language (SQL) to specify restrictions at the level of rows, columns, or cells. It uses mainly predicates to specify which parts of a table or view can be accessed. The extended SQL language is very simple, however, and it can support only the specification of closed access control policies. The open access control policies were not considered. Agrawal *et al.* (2005) also presented the relative concepts of multiple FGAC policies and their com-

‡ Corresponding author
* Project (No. 2006AA01Z430) supported by the National High-Tech Research and Development Program (863) of China
© Zhejiang University and Springer-Verlag Berlin Heidelberg 2010

bination approaches, but, in practice, the different types of multiple FGAC policies caused by different subjects (user or role) need different combination approaches. Moreover, the combination algorithm was not addressed. Barker (2008) presented a formal well-defined, dynamic fine-grained meta-level access control policy for category-based access control models. By meta-level policy specification, not only the closed access control policies, but also open access control policies can be represented at the fine-grained level. Moreover, such dynamic fine-grained meta-level access control policies can be represented and implemented in SQL. Their work, however, aims to remedy the access control representation problem and not to guide the implementation of FGAC in relational databases. Although it can represent FGAC policies at the row level, it is unable to specify the cell-level FGAC policies. Again, multiple policies were not considered in their work.

Motivated by the limitations of existing FGAC models, we developed a new FGAC model, which supports the specifications of open access control policies as well as closed access control policies at the cell level. For open access control policies, we mean negative authorization. In this model negative authorization (for expressing denials of access) and positive authorizations are both supported. Thus, the deficiencies of the models in Agrawal *et al.* (2005) and Barker (2008) are remedied. Moreover, based on the proposed FGAC model, we clarify the concept of multiple FGAC policies, and divide them into different types, according to which we provide different approaches to combining multiple policies. The combination algorithm is also presented. Finally, we implement the proposed FGAC model as a component of the database management system (DBMS). Its performance is also presented and analyzed.

The major contributions are summarized as follows:

1. We propose a FGAC model which supports the specification of both closed access control policies and open access control policies.

2. We present the definition of multiple FGAC policies and their combination algorithm.

3. We implement the FGAC model in relational databases as a component of DBMS.

2 Related works

Fine-grained access control was first introduced as a part of the access control system in INGRES by Stonebraker and Wong (1974), which was implemented by query modification technology. The basic idea of query modification is that before being processed, user queries are transparently modified to ensure that users can access only what they are authorized to access (Bertino *et al.*, 2005; Wang *et al.*, 2007). Views are used to specify and store access permission for users. When a user submits a query, DBMS first finds all views whose attributes include the attributes of the issued query, and then add the predicates of these views to the predicates of the original query to form a new modified query, which will be carried out.

Oracle virtual private database (VPD) also uses query modification to implement FGAC (Oracle Corporation, 2005). VPD supports FGAC through functions written as stored procedures which are associated with a relation. When a user accesses the relation, the function is triggered to return predicates, and the database rewrites the SQL statement submitted by the user to include these predicates. For providing enhanced access control, in addition to row-level access control, column-level VPD has been added to Oracle to provide column-level access control, which in turn associates functions with columns.

Recently, work on the policy for preserving privacy has boosted the research of FGAC (Agrawal *et al.*, 2002; Bertino *et al.*, 2005). Bertino *et al.* (2005) presented a privacy preserving access control model for relational databases, which needs a basis of FGAC in relational databases. Nevertheless, they did not describe how to implement the model. LeFevre *et al.* (2004) proposed a practical approach to incorporating privacy policy enforcement into an existing application and database environment where the implementation of FGAC at cell level was provided.

All works described above focused mainly on the enforcement of FGAC, and did not provide a FGAC model which supports many access control policies. Less work has been done with the FGAC model. The work of Agrawal *et al.* (2005) and Barker (2008) suffered from specific aforementioned limitations. Chaudhuri *et al.* (2007) also extended SQL language to support fine-grained authorization by predicated grants. Not only the column- and cell-level authori-

zations, but also the authorizations for function/procedure execution were supported. Moreover, they designed query defined user groups and authorization groups to simplify the administration of authorizations. Olson *et al.* (2008) presented a formal framework for reflective database access control policies where a formal specification of FGAC policies was supported by Transaction Datalog. The security analysis was also provided. Moreover, they enforced policies by compiling policies in Transaction Datalog into standard SQL views (Olson *et al.*, 2009). The shortcomings are that the negative authorization and multiple policies at fine granularity (which are the major contributions of our model) were not taken into account.

Kabra *et al.* (2006) considered two different aspects of FGAC: efficiency and information leakage of enforcement of FGAC. Using query modification to enforce FGAC, there may exist redundancies in the final executed queries because of the same predicates between the FGAC policies and the queries issued by users. These redundancies include not only cheap comparisons, but also expensive semi-joins, which would increase the execution time. Kabra *et al.* (2006) also considered the potential of information leakage through channels caused by exceptions, error messages, and user defined functions. For remedying the two problems, they proposed methods for redundancy removal, the definition of safety query plan, and the techniques to generate safe query plans.

Wang *et al.* (2007) proposed a correctness criterion of FGAC for databases, which contains three properties: secure, sound, and maximum. They argued that any algorithm used to implement FGAC must be sound and secure, and should strive to be maximum. They also pointed out that no algorithm exists that is both sound and secure. Then, they proposed an algorithm that is sound and secure. In this paper, we do not consider these aspects.

There is another important related work. Bertino *et al.* (1997) proposed an extended authorization model for relational databases, which supports negative authorization. This work inspired us to extend the FGAC model to support negative authorization. The main difference between their work and ours is the granularity of negative authorization: the model they proposed can support only negative authorization at coarse granularity (tables, views), but our model can express negative authorization at finer granularity (rows, columns, or cells).

3 Preliminary concepts

Table 1 gives some of the notations that will be used in the following discussions.

Filter plays a key role in the FGAC model; it is used to specify and determine the accessibility of a data item. Suppose there is a relation employee(emp_id, emp_name, dept_id, addr, phone) and a security policy which requires that each employee can read only the information about himself/herself. We can define a filter as ‘emp_name=USER()’ (USER() is a system function which returns the user name of the current session) to filter the tuples in employee.

Table 1 Notations for the fine-grained access control (FGAC) model

Parameter	Definition
$R(A_1, A_2, \ldots, A_n)$	A relation, and $t \in R$ denotes that t is a tuple in R and $t[A_i]$ is the value of attribute A_i in tuple t
$R.^*=\{A_1, A_2, \ldots, A_n\}$	The set of all attributes of R
$S=\{s_1, s_2, \ldots, s_m\}$	The set of subjects which may be users or roles in relational database systems
$U=\{u_1, u_2, \ldots, u_k\}$	The set of users in relational database systems
$\mathrm{RL}=\{r_1, r_2, \ldots, r_j\}$	The set of roles in relational database systems
$O=\{o_1, o_2, \ldots, o_n\}$	The set of objects, where o_i is a relation $R_i(A_{i1}, A_{i2}, \ldots, A_{im})$ which can be a table or a view
AC={select, insert, update, delete}	The set of actions
$\mathrm{FIL}=\{\mathrm{fil}_1, \mathrm{fil}_2, \ldots, \mathrm{fil}_n\}$	The set of filters, where fil_k (k=1, 2, …, n) is a filter with a tuple as the input parameter and with ‘false’, ‘true’, or ‘error’ as the output. For example, predicates in SQL which are used in the ‘where’ clause of the SQL statement can be considered as filters. There are two special filters, TRUE and FALSE. While TRUE always returns ‘true’ with any tuple as the input parameter, FASLE always returns ‘false’
PFIL	The set of policy filters
F	The set of policy functions

Namely, each tuple in employee is checked whether such a filter is true; when it is true, the tuple is returned to users. Filter makes the FGAC a content-based model, because the content of tuples is used to make access decision.

In our FGAC model, we allow FGAC policies to express negative authorization (we use 'prohibitions' and 'negative authorization' synonymously in the following discussions), which allows security administrators (SAs) to specify what data should not be accessed by certain users. The prohibitions are also expressed by filters. Therefore, for supporting both permissive policies (closed access control policies) and prohibitive policies (open access control policies), there are two filters: Allowed Filter and Prohibited Filter. The Allowed Filter is used to express a permissive policy and the Prohibited Filter is used to express a prohibitive policy. Conflicts between the Allowed Filter and the Prohibited Filter are resolved by applying the denial-takes-precedence policy where the Prohibited Filter overrides the Allowed Filter.

$\mathrm{PFIL}=\{\mathrm{pfil}_1, \mathrm{pfil}_2, \ldots, \mathrm{pfil}_m\}$ is the set of policy filters. A policy filter pfil_i is a tuple $\langle \mathrm{al_pfil}_i, \mathrm{pr_pfil}_i \rangle$, where $\mathrm{al_pfil}_i \in \mathrm{FIL}$ and $\mathrm{pr_pfil}_i \in \mathrm{FIL}$. $\mathrm{al_pfil}_i$ is called 'Allowed Filter' and $\mathrm{pr_pfil}_i$ 'Prohibited Filter'.

For example, suppose there is a security policy which requires that each employee should be prohibited to access the information of others in employee. We can define a policy filter $\mathrm{pfil}_i=\langle \mathrm{TRUE}, \mathrm{pr_pfil}_i \rangle$, where $\mathrm{pr_pfil}_i$='emp_name≠USER()'.

$F=\{f_{o1}, f_{o2}, \ldots, f_{on}\}$ is the set of policy functions. A policy function maps the set of all attributes of an object onto PFIL. For example, f_o is the policy function of object o. For each attribute $A_i \in o.^*$, there is a policy filter pfil_{oi} where $f_o(o.A_i)=\mathrm{pfil}_{oi}$ and $\mathrm{pfil}_{oi}=\langle \mathrm{al_pfil}_{oi}, \mathrm{pr_pfil}_{oi} \rangle$.

Given a policy function f_R, for any $A_k \in R.^*$, there is one and only one policy filter $\mathrm{pfil}_i=\langle \mathrm{al_pfil}_i, \mathrm{pr_pfil}_i \rangle$ with respect to A_k. We use $A_k \rightarrow \langle \mathrm{al_pfil}_i, \mathrm{pr_pfil}_i \rangle$ or $f_R(A_k)=\mathrm{pfil}_i$ to express the pfil_i which is the only policy filter with respect to A_k. Moreover, we use $f_R(A_k, \mathrm{allow})=\mathrm{al_pfil}_i$ and $f_R(A_k, \mathrm{prohibit})=\mathrm{pr_pfil}_i$ to express the Allowed Filter and the Prohibited Filter, respectively. $f_R(A_k)=\mathrm{pfil}_i$ means that, for a tuple X, $X[A_k]$ is allowed to be accessed if $\mathrm{al_pfil}_i(X)$=true and $\mathrm{pr_pfil}_i(X)$=false, and prohibited to be accessed otherwise (here the denial-takes-precedence policy has been considered).

To illustrate our model in detail, we use the following schemas in our examples: (1) employee(emp_id, emp_name, dept_id, addr, phone), (2) manager(mgr_id, dept_id), and (3) dept(dept_id, dept_name).

4 Fine-grained access control model

4.1 Overview

FGAC controls the access of users in a relational database and the access modes include select, insert, update, and delete. In relational databases, there exist mainly two approaches to granting privileges to users. One is to directly assign permissions to users, and the other is to grant privileges to the roles that users are assigned. FGAC models should support both of these authorization approaches.

In the proposed FGAC model (Fig. 1) there are two sub-models, direct-FGAC and role-FGAC. The FGAC model provides a more complex form of object, where each attribute of an object associates with a policy filter.

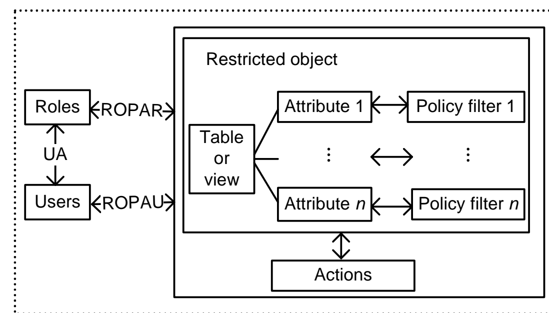

Fig. 1 Fine-grained access control (FGAC) model with two sub-models, direct-FGAC and role-FGAC
UA: user assignment; ROPAR: restricted object privilege assignment with role; ROPAU: restricted object privilege assignment with user

4.2 Direct-FGAC model

The direct-FGAC (D-FGAC) model allows assigning fine-grained and precise privileges to users directly. The D-FGAC model is formally described as follows:

Definition 1 The D-FGAC model is composed of the following components: (1) a set U of users, a set O of objects, a set FIL of filters, a set F of policy functions, a set AC of actions; (2) a set of restricted object $\mathrm{RO}=\{(o, f_o) | o \in O, f_o \in F\}$; (3) a set of restricted object

privileges ROP={(ro, a)|ro∈RO, a∈AC}; (4) restricted object privilege assignment ROPAU⊆U×OP, a many-to-many mapping restricted object privilege to user assignment relation.

In the D-FGAC model, the key component is the restricted object. A restricted object ro is (o, f_o) where o is a relation and f_o is a policy function. f_o assigns policy filters for all attributes of o to specify which data items in o are allowed to access. Each policy filter contains the Allowed Filter and the Prohibited Filter to support permissive and prohibitive policies.

According to the definition above, in the D-FGAC model a FGAC policy is expressed as $p=(u, ((o, f_o), a))$. For simplicity, in what follows, a FGAC policy is expressed as $p=(u, (o, f_o), a)$, and user(p), object(p), pf(a), and action(p) denote the user, the object, the policy function, and the action in p, respectively.

Example 1 Suppose John is an employee, and there is a security policy which requires that John can read only the information about himself in table employee. Then in the D-FGAC model, this policy can be expressed as P^1_{John}={John, (employee, f^1_{John}), select} and the content of f^1_{John} is as follows:

{emp_id→<emp_name='John', FALSE>, emp_ name →<emp_name='John', FALSE>, dept_id→<emp_ name='John', FALSE>, addr→<emp_name='John', FALSE>, phone→<emp_name='John', FALSE>}.

Moreover, suppose a more precise security policy at finer granularity is needed, which requires that John can read the information of attributes emp_id, emp_name, and dept_id of all employees, but to preserve individual privacy, he can read only the information of attributes addr and phone about himself. The D-FGAC policy can be expressed as P^2_{John}= {John, (employee, f^2_{John}), select} and the content of f^2_{John} is as follows:

{emp_id→<TRUE, FALSE>, emp_name→<TRUE, FALSE>, dept_id→<TRUE, FALSE>, addr→<emp_ name='John', FALSE>, phone→<emp_name= 'John', FALSE>}.

In Example 1, all policy filters in f^1_{John} are the same in policy P^1_{John}. When all policy filters in a policy function f are the same, we simplify the expression as $f(R.^*)$=pfil. Therefore, in Example 1, f^1_{John}(employee.*) =<emp_name='John', FALSE>.

4.3 Access decision rule

Before dealing with the access decision rule, we introduce a special symbol Φ which is used to denote the unauthorized value. Φ has the following properties: (1) $\Phi \neq \Phi$; (2) For any constant value c, $\Phi \neq c$.

When a user accesses a table by operation select, update, or delete, in reality it is the tuples in this table that are accessed. When the user inserts a tuple X into the table, we also say that the tuple X is accessed. Moreover, databases allow users to insert values of many, but not all, columns of a tuple into tables. In this study we consider only the case where the values of all columns of a tuple are inserted into a table. The extension to the case of inserting the values of many columns of a tuple is immediately achievable.

Definition 2 (Access decision tuple) For a D-FGAC policy $p=(u, (o, f_o), a)$, where o is a relation R, $R.^*=\{A_1, A_2, \ldots, A_n\}$, and X is the tuple that is accessed, the access decision tuple X_a is constructed as follows: $\forall A_i \in R.^*$, if $f_o(A_i$, allow)=true$\wedge f_o(A_i$, prohibit)=false, $X_a[A_i]=X[A_i]$; else $X_a[A_i]=\Phi$.

Definition 3 (Access decision relation) For a D-FGAC policy $p=(u, (o, f_o), a)$, where o is a relation R, $R.^*=\{A_1, A_2, \ldots, A_n\}$, the access decision relation of R (i.e., R_a) is formed through substituting each tuple X in R with its access decision tuple X_a.

Definition 4 (Access decision rule) For a D-FGAC policy $p=(u, (o, f_o), a)$, where o is a relation R, $R.^*=\{A_1, A_2, \ldots, A_n\}$, the access decision rules are:

When a is select, update, or delete, the result of accessing R is the same as the result of accessing R_a, which is the access decision relation of R;

When a is insert, and X is the tuple being accessed and its access decision tuple is X_a, only when the following condition holds, is the access allowed: $\forall A_i \in R.^*$, $X_a[A_i] \neq \Phi$.

Example 2 In Example 1, there is a D-FGAC policy P^2_{John}=(John, (employee, f^2_{John}), select) for John and table employee, which allows John to access the information of emp_id, emp_name, and dept_id of all employees, but only the information of addr and phone for himself. Suppose the table employee is as shown in Table 2. According to Definitions 2 and 3, the access decision relation of employee employee$_a$ is shown in Table 3. Thus, for any SQL statement issued by John to access employee, the result is the same as the result of accessing the relation employee$_a$.

Table 2 employee

Emp_id	Emp_name	Dept_id	Addr	Phone
1	Andy	1101	Brooks	111-1111
2	Mary	1102	Wood	222-2222
3	John	1103	Cricket	333-3333

Table 3 employee$_a$

Emp_id	Emp_name	Dept_id	Addr	Phone
1	Andy	1101	Φ	Φ
2	Mary	1102	Φ	Φ
3	John	1103	Cricket	333-3333

4.4 Role-FGAC model

With the development of the Internet, the number of users in databases has increased sharply. Traditional access control models such as the discretionary access control (DAC) model are often infeasible. Traditional access control models also make the job of an SA to define and maintain security policy tedious. Fortunately, role-based access control (RBAC) overcomes these shortcomings by assigning users to roles and assigning permissions to roles. RBAC simplifies the specification and management of security policies (Ferraiolo *et al.*, 2001). Moreover, almost all relational databases systems support the RBAC model. Therefore, FGAC should be integrated into the RBAC model.

We propose a model named role-FGAC (R-FGAC) where the fine-grained privileges are assigned to users through the roles that the users have in relational databases.

Definition 5 The R-FGAC model is composed of the following components: (1) a set U of users, a set RL of roles, a set O of objects, a set FIL of filters, a set F of policy functions, a set AC of actions; (2) a set of restricted objects RO=$\{(o,f_o)|o\in O, f_o\in F\}$; (3) a set of restricted object privileges ROP=$\{$(ro, a)|ro$\in$RO, $a\in$AC$\}$; (4) user assignment UA$\subseteq U\times$RL, a many-to-many mapping user to role assignment relation; (5) restricted object privilege assignment ROPAR$\subseteq$RL$\times$ ROP, a many-to-many mapping restricted object privilege to role assignment relation.

Each element in the R-FGAC model has the same meaning as the corresponding element in the D-FGAC model.

According to Definition 5, in the R-FGAC model a FGAC policy is expressed as $p=(r, ((o,f_o), a))$. For simplicity, the FGAC policy is expressed as $p=(r, (o, f_o), a)$, and role(p), object(p), pf(a), and action(p) denote the role, the object, the policy function, and the action in p, respectively.

Example 3 Suppose there is a security policy which requires that each employee can read only the information about himself/herself in table employee. In the D-FGAC model, to achieve this goal, we need to define the FGAC policy for each employee, which is infeasible. In the R-FGAC model, it is easier to accomplish this. We first define a role R_EMP which is assigned to all employees, and then define an R-FGAC policy $P_{\text{R_EMP}}$ and assign it to the role R_EMP. The policy $P_{\text{R_EMP}}$ is expressed as $P_{\text{R_EMP}}$={R_EMP, (employee, $f_{\text{R_EMP}}$), select}, where $f_{\text{R_EMP}}$(employee.*)=<emp_ name=USER(), FLASE> (USER() is a system function which returns the user name of the current session).

The access decision rule of the R-FGAC model is similar to that of the D-FGAC model as defined in Definition 4.

In the R-FGAC model, what we consider is the standard RBAC model (Ferraiolo *et al.*, 2001). Currently, there is no RDBMS vendor that supports negative authorization for the RBAC model, and commercial implementations support only flat RBAC, which is very limited and supports only a simple version of RBAC (Bertino and Sandhu, 2005). Although the RBAC model has been extended to support negative authorization (Al-Kahtani and Sandhu, 2004), it cannot govern access at the fine granularity for relational databases. Extending the RBAC model to support negative authorization at fine granularity, however, is our major contribution. Moreover, although DAC can be simulated by RBAC (Osborn *et al.*, 2000), in practice DAC and RBAC are both widely used in relational databases. Hence, we describe D-FGAC and R-FGAC separately.

4.5 FGAC vs. traditional access control

In this subsection, we compare the proposed FGAC model with traditional access control in relational databases, such as DAC and RBAC.

A FGAC model is proposed to control user access to tables or views, and the access modes include only select, insert, update, and delete. Comparing these, we suppose that in traditional access control the subjects are users or roles, that the objects are tables

or views, and that the access modes are select, insert, update, and delete.

Based on this, the FGAC model extends traditional access control to support closed access policies and open access control policies at fine granularity in relational databases. Namely, traditional access controls are special cases of the FGAC model. In the traditional access control model, a policy is expressed as $P_t=(s, o, a)$, where s can be a user or a role, o is an object, and a is an access mode. In the FGAC model, a policy is expressed as $P_{fgac}=(s, (o, f_o), a)$. When all policy filters involved in policy function f_o are <TRUE, FALSE>, P_{fgac} is degraded to P_t. Therefore, the FGAC model can be compatible with a traditional access control model in database. They can be considered together to process policies management, such as detecting policy violation and policy redundancy.

In traditional access control models such as the DAC model or the RBAC model, delegation is supported which allows a user to assign a subset of his/her available rights to another user. Because the FGAC model is extended from a traditional access control model, the FGAC model can also support delegation. In this study we do not deal with this flexible feature. We consider only the cases where the FGAC policy can be defined only by the SA, who also has authority to assign FGAC policies to users or roles. The reasons for these restrictions are summarized as follows: (1) According to the requirements of existing applications, FGAC policies are defined and maintained mainly by the SA, which cannot be intervened upon by users (Agrawal *et al.*, 2005; Oracle Corporation, 2005); (2) The FGAC policy is a very flexible policy. If the authority for definition and assignment of the FGAC policy is granted to users, the burden of management and maintenance of the FGAC policy is too great, and there may be many security risks that cannot be managed. Making the FGAC model support delegation is a challenging work, on which we plan future efforts.

4.6 Summary

In the proposed FGAC model, we use the policy filter with the Allowed Filter and the Prohibited Filter to specify which information can be accessed and which information is prohibited. Filters allow the FGAC model to support closed access control policies and open access control policies.

In our FGAC model, we take the restricted object to replace the object in the traditional access control model in relational databases. Because object is a special case of the restricted object, the FGAC model is extended from traditional access control models, and can be compatible with them.

In DBMS, the D-FGAC model coexists with the R-FGAC model. We denote the D- and R-FGAC policy in a uniform way: a FGAC policy $p=(s, (o, f_o), a)$, where $s \in S$ is a subject which can be a user u or a role r. For simplicity, we use User_Role(u, r) to denote that the user u has the role r.

5 Multiple fine-grained access control policies

Agrawal *et al.* (2005) stated that "if multiple restrictions have been defined for a user u and a table T, then u's access to T is governed by the combination of these restrictions", and considered two combination approaches—intersection and union. They interpreted only the meaning of the two approaches, and did not address how and when to combine multiple restrictions. In this section, we present a definition of multiple policies and propose approaches to composing multiple policies by intersection and union. The combination algorithm is also provided.

Definition 6 For a FGAC policy $p=(s, (o, f_o), a)$, we say p controls the access a of a user u to object o if either of the following conditions holds: (1) s is a user and $s=u$; (2) s is a role and User_Role(u, s).

Definition 7 (Multiple policies) $\forall s_1, s_2, \ldots, s_n \in S$, $\forall o \in O$, $\forall a \in$ AC, the policies $p_1=(s_1, (o, f_1), a)$, $p_2=(s_2, (o, f_2), a)$, …, $p_n=(s_n, (o, f_n), a)$ (where $f_1, f_2, \ldots, f_n$ are policy functions) are multiple policies of user u over object o for action a, if policies $p_1, p_2, \ldots, p_n$ are defined to control access a of user u to object o.

In FGAC policies, the subject can be a user or a role. When combining multiple FGAC policies, we take the feature of subject into account. We comply with the following three principles:

1. User priority principle: the privilege directly assigned to a user is prior to the privilege assigned to a user through roles.

2. Role principle: when a user has many roles, he/she has all privileges assigned to these roles.

3. Secure principle: while the two principles

above are satisfied, additional policies are combined to decrease the user's access to data to guarantee information security.

Following these three principles, we first divide multiple policies into different types and then provide different combination approaches.

According to Definition 7, for two policies $p_1=(s_1, (o, f_1), a)$ and $p_2=(s_2, (o, f_2), a)$, there are multiple policies of a user u over object o for action a, if one of the following four conditions holds:

1. $s_1=s_2=u$.
2. $s_1=s_2=r$ where r is a role and User_Role(u, r).
3. $s_1=r_1$ and $s_2=r_2$ where r_1 and r_2 are different roles and User_Role(u, r_1), User_Role(u, r_2).
4. $s_1=u$ and $s_2=r$ where r is a role and User_Role(u, r).

According to the different conditions, we divide multiple policies into three types:

1. Defined multiple policies: multiple policies caused by conditions 1 and 2.
2. Role multiple policies: multiple policies caused by condition 3.
3. User role multiple policies: multiple policies caused by condition 4.

We use different approaches to combine different types of multiple policies.

When multiple policies satisfy condition 1 or condition 2, they have the same user or the same role. Since in the D-FGAC model or R-FGAC model, all FGAC policies can be defined only by SA, the reason for multiple policies satisfying condition 1 or condition 2 is the definition of security policies caused by SA. Thus, we call these multiple policies 'defined multiple policies', and we use intersection to compose defined multiple policies. There are two reasons for combining defined multiple policies by intersection: (1) to be consistent with the existing work. As far as we know, multiple policies of FGAC in databases are combined by intersection in all existing work (Agrawal *et al.*, 2005; Oracle Corporation, 2005); (2) to be more secure intuitively. With intersection, the user's access to data decreases as additional restrictions are applied, but with a union approach, access to data increases as additional restrictions are applied (Agrawal *et al.*, 2005). Therefore, for security, we use intersection to combine defined multiple policies. If the SA wants to relax the user's access, however, they can do so by revising the original FGAC policies instead of writing new ones.

When multiple policies satisfy condition 3, they control the user's access because of the feature of roles. When a user has two different roles, he/she will have all access permissions of both roles. Therefore, we intuitively use union to compose role multiple policies.

When multiple policies satisfy condition 4, there is a policy directly specified for a user and a policy specified for a role that the user has. In our view, the policy specified for the user is defined to further restrict the access of the user. Therefore, for making the policy directly specified for user precedence, we use intersection to combine user role multiple policies.

The combination approaches above satisfy the three principles. For defined multiple policies, the user priority principle and role principle are definitely held, because the subject is only one user or role. Thus, for satisfying the secure principle, intersection is applied. For role multiple policies, the combination algorithm is union, according to the role principle. User role multiple policies are combined by intersection for satisfying the user priority principle.

In the following, we introduce the definition of intersection and union approaches. For simplicity, the definitions involve only two FGAC policies; extending this to combine more than two policies is immediately achievable. The two multiple policies $p_1=(s_1, (o, f_{o1}), a)$ and $p_2=(s_2, (o, f_{o2}), a)$ control the access of user u to object o for action a.

Definition 8 (Intersection) The combined policy is $p_\wedge=(u, (o, f_\wedge), a)$, denoted as $p_\wedge=p_1\wedge p_2$, which is formed with the following approach:

for each $A_i\in R.^*$ **do**
  $f_\wedge(A_i$, allow$)=f_{o1}(A_i$, allow) and $f_{o2}(A_i$, allow);
  $f_\wedge(A_i$, prohibit$)=f_{o1}(A_i$, prohibit) or $f_{o2}(A_i$, prohibit);
end for

Definition 9 (Union) The combined policy is $p_\vee=(u, (o, f_\vee), a)$, denoted as $p_\vee=p_1\vee p_2$, which is formed with the following approach:

for each $A_i\in R.^*$ **do**
  $f_\vee(A_i$, allow$)=f_{o1}(A_i$, allow) or $f_{o2}(A_i$, allow);
  $f_\vee(A_i$, prohibit$)=f_{o1}(A_i$, prohibit) and $f_{o2}(A_i$, prohibit);
end for

When there are more than two policies, we can first combine two policies to form a temporary policy and then combine the temporary policy with other policies. Then, for multiple policies including finite policies, we can combine them into a single policy.

When the access a of a user u to an object o is controlled by defined multiple policies, role multiple policies, and user role multiple policies simultaneously, we compose all these policies by the following steps: (1) Combine all FGAC policies that are directly defined to control the access a of user u to object o (i.e., policies only in the D-FGAC model) into policy p_{dfgac} by intersection; (2) Search all roles of u: r_1, r_2, …, r_n, namely $\forall i\in\{1, 2, …, n\}$, User_Role($u$, r_i); (3) For each role r_i, combine all FGAC policies into p_{ri}, which are defined to control access a of role r_i to object o by intersection; (4) Compose policies p_{r1}, p_{r2}, …, p_{rn} into a policy p_{rfgac} by union; (5) Combine policies p_{dfgac} and p_{rfgac} by intersection to control access a of user u to object o. Algorithm 1 is the algorithm of combining multiple FGAC policies.

Algorithm 1 Multiple policies combination algorithm

Input: user U, relation R, action A, database D.
Output: the combined FGAC policy P_{out}.

1 RS←∅; /* RS is a set recording roles */
2 PS_{temp}←∅; /* PS_{temp} is a set recording FGAC policies */
3 PS_{role}←∅; /* PS_{role} is a set recording FGAC policies */
4 PS_{temp}=GetDFGACPoliciesSet(U, R, A, D);
 /* Search all D-FGAC policies for the user U over relation R about access A in database D, and add these D-FGAC policies into the set PS_{temp} */
5 P_D=InterSection(PS_{temp});
 /* Combine all D-FGAC policies in PS_{temp} by an intersection approach */
6 RS=GetRoleSet(U, D);
 /* Search all roles for user U in database D, and add these roles into the set RS */
7 **for** all $r\in$RS **do**
8 PS_{temp}←∅;
9 PS_{temp}=GetRFGACPoliciesSet(r, R, A, D);
 /* Search all FGAC policies for the role r over relation R about access A in database D, and add these policies into the set PS_{temp} */
10 PS_{role}←InterSection(PS_{temp});
 /* Combine all FGAC policies in PS_{temp} by an intersection approach and add the combined policy into the set PS_{role} */
11 **end for**
12 P_R=Union(PS_{role});
 /*Combine all FGAC policies in PS_{role} by a union approach*/
13 $P_{\text{out}}=P_D\wedge P_R$.

Example 4 Suppose there is a user U in the database, and the user has three roles R_1, R_2, and R_3, namely User_Role(U, R_1), User_Role(U, R_2), and User_Role(U, R_3). For the select operation and relation R, there are three FGAC policies for user U—P_U^1, P_U^2, and P_U^3, two policies for R_1—P_{R1}^1 and P_{R1}^2, three policies for R_2—P_{R2}^1, P_{R2}^2, and P_{R2}^3, and a policy for R_3—P_{R3}. Therefore, when U accesses relation R with the select operation, according to Algorithm 1, the combined policy is $P_{\text{out}}=(P_U^1\wedge P_U^2\wedge P_U^3)\wedge((P_{R1}^1\wedge P_{R1}^2)\vee(P_{R2}^1\wedge P_{R2}^2)\vee P_{R3})$.

6 Implementation

The proposed FGAC model supports the specification of closed access control policies, open access control policies, and multiple policies at fine granularity for relational databases. In this section, we present the implementation approaches of this model. Implementing the FGAC model means enforcing the access decision rules introduced in Section 4.3. Access decision rules include two sub-rules: one for select, update, and delete operations, and the other for the insert operation. In the following, we first introduce how to enforce the first rule, taking the select operation as an example, and then briefly present the enforcement of the second rule.

As mentioned, there are many works dealing with the implementation of FGAC in relational databases. Query modification is a typical approach to enforcing FGAC. LeFevre *et al.* (2004) presented an efficient approach to enforcing FGAC policies at cell-level granularity. We follow their approach to enforce our FGAC models by query modification for the select operation. Note that when implementing the FGAC model for the select operation, we use NULL to replace the unauthorized data as similar to the previous work, which is different from the proposed model where we use Φ to replace unauthorized data.

Query modification means that, before being processed, user queries are modified transparently into other queries, which will be enforced finally. The enforced queries ensure that users do not access what they are unauthorized to access according to FGAC policies. For example, when a user U issued a query Q to database systems, the query Q will be automatically transformed to query Q' (which is constructed according to FGAC policies for user U through the

query modification approach) to be processed.

In the following discussion, we assume that a user U submits query Q to database systems, and the relations (tables or views) involved in Q are $R_1, R_2, \ldots, R_n$; moreover, the FGAC policies for U and R_i (i=1, 2, …, n) are $P_1=(U, (R_1, f_1)$, select), $P_2=(U, (R_2, f_2)$, select), …, $P_n=(U, (R_n, f_n)$, select). Note that, we do not consider multiple FGAC policies for U and R_i, because multiple FGAC policies can be combined into a single FGAC policy according to Algorithm 1. Namely, the extension to enforcing multiple FGAC policies is immediately achievable.

When modifying query Q into Q', we replace each relation involved in Q with a temporary view (which is dynamically created according to the FGAC policy) and keep the other clauses of Q unchanged. Therefore, there are two steps:

Step 1: creating a temporary view for each relation involved in query Q. Suppose the relation in Q is R_i, where the attributes set is $\{A_{i1}, A_{i2}, \ldots, A_{im}\}$. The dynamically created view V_i is structured as follows:

(SELECT CASE WHEN ($f_i(A_{i1}$, allow) and not ($f_i(A_{i1}$, prohibit))) THEN A_{i1} ELSE NULL END AS A_{i1},

CASE WHEN ($f_i(A_{i2}$, allow) and not ($f_i(A_{i2}$, prohibit))) THEN A_{i2} ELSE NULL END AS A_{i2},

…,

CASE WHEN ($f_i(A_{im}$, allow) and not ($f_i(A_{im}$, prohibit))) THEN A_{im} ELSE NULL END AS A_{im}

FROM R_i)

Step 2: replacing relation R_i in Q with temporary view V_i, for $i \in \{1, 2, \ldots, n\}$, to obtain the final executed query Q'.

In a dynamically created view, the data unauthorized to access is replaced with NULL by the ‘CASE’ clause. Therefore, the users cannot access what they are unauthorized to access according to the FGAC policy when the modified query Q' is processed. Note that, only the data that satisfies the Allowed Filter and does not satisfy the Prohibited Filter is not replaced with NULL. Hence, we use ‘($f_i(A_{i1}$, allow) and not ($f_i(A_{i1}$, prohibit)))’ as the condition of the ‘CASE’ clause.

Example 5 Suppose there is only one FGAC policy P^2_{John} for user John and the table employee in Example 1. When John submits a query Q to search all the information of employees, Q will be automatically modified into Q':

Q=“SELECT * FROM employee”;

Q'=“SELECT * FROM (SELECT emp_id, emp_name, dept_id,

CASE WHEN ((emp_name=‘John’) and not FALSE) THEN addr ELSE NULL END AS addr,

CASE WHEN ((emp_name=‘John’) and not FALSE) THEN phone ELSE NULL END AS phone

FROM employee)”.

Note that for emp_id, emp_name, and dept_id, the ‘CASE’ clause is not used, because their filters are <TRUE, FALSE>, which means all data can be accessed.

The implementation of update and delete operations is similar to the enforcement of the select operation introduced above.

For the insert operation, the enforcement approach is simple. When a user U inserts a tuple X into a relation $R(A_1, A_2, \ldots, A_n)$, DBMS searches all FGAC policies over U and R and combines these policies into a single policy, and then obtains all policy filters for each attribute of R in the single policy. For all policy filters, take the tuple X as the input parameter to check Allow Filters and Prohibit Filters. If all Allow Filters return true and Prohibit Filters return false, the X is allowed to be inserted into R; otherwise, it is prohibited.

Wang *et al.* (2007) presented a concept of ‘security’ for the enforcement of the select operation of FGAC in relational databases. If the answer returned does not depend on any information not allowed by FGAC policies for all queries, the enforcement approach is secure. Moreover, they stated that the enforcement by query modification with a dynamically created view is secure. This means the enforcements for our model are secure. Actually, since in the dynamically created view all unauthorized data is replaced with NULL, and queries are modified to access the view, the result is definitely independent of the information not allowed by FGAC policies.

7 Experiments

In this section, we briefly present the performance of the enforced FGAC model, which was implemented in DM6 (Da Meng Database Corporation, 2000).

LeFevre *et al.* (2004) presented a comprehensive set of performance experiments, showing that the overhead of the query modification with the 'CASE' clause to enforce cell-level FGAC for the select operation is small. We used this type of approach to enforce our FGAC model, so we can state that the cost of implementation of the select operation is also small. Although our enforcement is similar to that in LeFevre *et al.* (2004), our further contribution is obvious, as we provide a new FGAC model which can support closed access control, open access control, and multiple access control policies at fine granularity. We remedy the problem of access control policies specification in relational databases stated, for example, by Barker (2008).

When FGAC is enforced in relational databases, not only should the cost of the implementation of the select operation be evaluated, but also the performance of the DBMS. There has been little work evaluating the performance of DBMS with FGAC. In the following, we introduce the performance of DBMS where FGAC for select, update, delete, and insert operations was implemented.

The performance of DBMS with FGAC was tested using the TPC-W testing tool (Zhu *et al.*, 2006), which was developed according to the TPC-W benchmark specification (TPC, 2002). TPC-W evaluates the throughput of a database with an average number of Web interactions per second (WIPS). Using the TPC-W testing tool, we compared the system performances in two situations: with and without the control of FGAC policies. The test environment is shown in Table 4.

Table 4 The TPC-W testing environment

	Operating system	Database	Testing program	CPU	Memory
Client	Windows XP	No	TPC-W	Intel 2.8 GHz	1 GB×2
Web server	Windows XP	No	Weblogic	Intel 2.8 GHz	1 GB×2
Database server	Windows XP	DM6	No	Intel 2.8 GHz	1 GB×2

We created 100 users in the TPC-W testing database, and divided them into three roles: manager (MAN), customer (CUST), and regional customer (REGION). The FGAC policies are defined as follows:

P_1=(CUST, (orders, f_1), select), where f_1(orders.*) =<o_c_id in (select c_id from customer where c_uname=USER), FALSE>;

P_2=(CUST, (orders, f_2), select), where f_2(o_bill_addr_id)=<o_bill_addr_id in (select c_addr_id from customer where c_uname=USER), FALSE> and the policy filters for other attributes are <TRUE, FALSE>;

P_3=(CUST, (order_line, f_3), insert), where f_3(order_line.*)=<TRUE, ol_i_id<100>;

P_4=(REGION, (orders, f_4), select), where f_4(orders.*)=<o_c_id in (select c_id from customer where c_uname=USER), FALSE>;

P_5=(MAN, (item, f_5), update), where f_5(item.*)= <i_stock<10000, FALSE>.

We measured the system performances with and without FGAC policies by varying the number of emulated browsers. Experimental results are reported in Fig. 2. Although the FGAC policies did affect the performance, the system performance was rational, almost linear and thus acceptable.

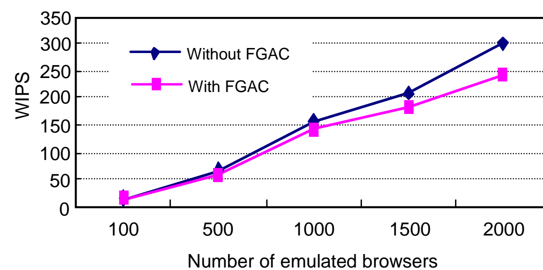

Fig. 2 The TPC-W test results, Web interactions per second (WIPS), under different numbers of emulated browsers

8 Conclusions

For integrating FGAC into relational databases, it is necessary to provide a FGAC model which can support the specification of many access control policies. In this paper, we proposed a FGAC model which has two key features: supporting the specification of open access control policies as well as closed access control policies, and supporting multiple policies. Based on this model, we presented the definition of multiple FGAC policies and the combination algorithm. Then we implemented this model as a component of the database management system (DBMS). Performance results showed that the proposed model and the implementation approach are feasible.

References

Agrawal, R., Kiernan, J., Srikant, R., Xu, Y., 2002. Hippocratic Databases. Proc. Very Large Data Bases, p.563-574.

Agrawal, R., Bird, P., Grandison, T., Kiernan, J., Logan, S., Rjaibi, W., 2005. Extending Relational Database Systems to Automatically Enforce Privacy Policies. Proc. 21st Int. Conf. on Data Engineering, p.1013-1022. [doi:10.1109/ICDE.2005.64]

Al-Kahtani, M.A., Sandhu, R., 2004. Rule-Based RBAC with Negative Authorization. Proc. 20th Annual Computer Security Applications Conf., p.405-415. [doi:10.1109/CSAC.2004.32]

Barker, S., 2008. Dynamic meta-level access control in SQL. *LNCS*, **5094**:1-16. [doi:10.1007/978-3-540-70567-3_1]

Bertino, E., Sandhu, R., 2005. Database security-concepts, approaches, and challenges. *IEEE Trans. Depend. Secur. Comput.*, **2**(1):2-19. [doi:10.1109/TDSC.2005.9]

Bertino, E., Samarati, P., Jajodia, S., 1997. An extended authorization model for relational database. *IEEE Trans. Knowl. Data Eng.*, **9**(1):85-101. [doi:10.1109/69.567051]

Bertino, E., Byun, J.W., Li, N., 2005. Privacy-preserving database systems. *LNCS*, **3655**:178-206. [doi:10.1007/11554578_6]

Byun, J.W., Bertino, E., Li, N., 2005. Purpose Based Access Control of Complex Data for Privacy Protection. Proc. 10th ACM Symp. on Access Control Models and Technologies, p.102-110. [doi:10.1145/1063979.1063998]

Chaudhuri, S., Dutta, T., Sudarshan, S., 2007. Fine Grained Authorization Through Predicated Grants. Int. Conf. on Data Engineering, p.1174-1183. [doi:10.1109/ICDE.2007.368976]

Da Meng Database Corporation, 2000. DM Database. Available from http://www.dameng.com/dmweb/ [Accessed on Feb. 14, 2009].

Dwivedi, S., Menezes, B., Singh, A., 2005. Database Access Control for E-business: a Case Study. Proc. Int. Conf. on Management of Data, p.168-175.

Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D., Chandramouli, R., 2001. Proposed NIST standard for role-based access control. *ACM Trans. Inform. Syst. Secur.*, **4**(3):224-274. [doi:10.1145/501978.501980]

Jain, U., 2004. Seminar Report Fine-Grained Access Control in Databases. Technical Report, Bernard Menezes KReSIT, IIT Bombay.

Kabra, G., Ramamurthy, R., Sudarshan, S., 2006. Redundancy and Information Leakage in Fine-Grained Access Control. Proc. ACM SIGMOD Int. Conf. on Management of Data, p.133-144. [doi:10.1145/1142473.1142489]

LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., DeWitt, D., 2004. Limiting Disclosure in Hippocratic Databases. Proc. Very Large Data Bases, p.108-119.

Olson, L.E., Gunter, C.A., Madhusudan, P., 2008. A Formal Framework for Reflective Database Access Control Policies. Proc. 15th ACM Conf. on Computer and Communications Security, p.289-298. [doi:10.1145/1455770.1455808]

Olson, L.E., Gunter, C.A., Cook, W.R., Winslett, M., 2009. Implementing reflective access control in SQL. *LNCS*, **5645**:17-32. [doi:10.1007/978-3-642-03007-9_2]

Oracle Corporation, 2005. Oracle Virtual Private Database. Technical Report. Available from http://www.oracle.com/technology/deploy/security/db_security/virtual-private-database/index.html [Accessed on Jan. 10, 2009].

Osborn, S., Sandhu, R., Munawer, Q., 2000. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inform. Syst. Secur.*, **3**(2):85-106. [doi:10.1145/354876.354878]

Rizvi, S., Mendelzon, A., Sudarshan, S., Roy, P., 2004. Extending Query Rewriting Techniques for Fine-Grained Access Control. Proc. ACM SIGMOD Int. Conf. on Management of Data, p.551-562. [doi:10.1145/1007568.1007631]

Stonebraker, M., Wong, E., 1974. Access Control in a Relational Database Management System by Query Modification. Proc. ACM Annual Conf., p.180-186. [doi:10.1145/800182.810400]

Transaction Processing Performance Council (TPC), 2002. TPC BENCHMARK^{TM} W (Web Commerce) Specification, Version 1.8. Available from http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf [Accessed on May 8, 2009].

Wang, Q., Yu, T., Li, N., Lobo, J., Bertino, E., Irwin, K., Byun, J.W., 2007. On the Correctness Criteria of Fine-Grained Access Control in Relational Databases. Proc. Very Large Data Bases, p.555-566.

Zhu, H., Fu, X., Lin, Q.H., Lu, K., 2006. The design and implementation of a performance evaluation tool with TPC-W benchmark. *J. Comput. Inform. Technol.*, **14**(2):149-160. [doi:10.2498/cit.2006.02.06]