A FLEXIBLE DATABASE AUTHORIZATION SYSTEM

Bazyli Blicharski, Krzysztof Stencel
b.blicharski@students.mimuw.edu.pl, stencel@mimuw.edu.pl
Institute of Informatics, Warsaw University
Warsaw
Poland

Abstract
In this paper we present an authorization mechanism for a relational database. It allows defining the user privileges exact to a table row. To implement that we used the INSTEAD triggers installed on views. This authorization system is an interesting application of such triggers. The user privileges are organized into parameterized roles that can be instantiated and/or inherited by other roles.

Key Words
database security, role, privilege, view, INSTEAD trigger

1. Introduction

The users cannot be granted unlimited access to all information in a database. It is true for a lot of reasons — some data must be secret and there is a danger of intentional or accidental data destruction. The heart of an information system is a database. Software engineers have to solve problems concerned with the access privileges granted to the database users. This was also the case of USOS i.e. the student management information system used by many Polish universities [1, 2, 3]. USOS is a large system that encompassed the affairs of the whole school and has many users. Therefore an authorization mechanism for this system was inevitable. Later on we explain why the standard authorization mechanisms provided by database vendors were not sufficient for this purpose. We decided to build a discretionary access control system [4, 5].

Unfortunately at the beginning USOS was a small system designed for one faculty only and was crafted without taking into account the need to authorize the access to data for users. Appropriate features had to be added to the installed and working software. Those features had to be introduced under some assumptions that allowed minimizing necessary changes in the existing source code of the system.

Before the decision to add authorization to the system was taken, one graphical interface had been built that had been intended for all users. This interface had been written in Oracle*Forms and had referenced specific names of database objects. To avoid radical rewriting of the existing code we assumed that we would not change the users interface and allow it to reference the same names of database objects after the introduction of authorization system. We did it by means of views and synonyms. The forms reference the same name. However, before it was the name of a table and now it is a synonym that points to a view based on the same original table. The assumption of not changing the referenced names had another consequence—the solution could not have been based on the client software but it had to be deployed on the database server side. Of course we could not also change the DBMS used and we had to continue with the same vendor (in our case Oracle).

The development of an authorization system under such assumptions was not a trivial task. As a result we obtained very flexible software that allows defining the access privileges exact to one column and one row. During construction of the system we heavily used the capabilities of the modern DBMS, i.e. updateable views with INSTEAD triggers. The produced solution is an interesting application of these capabilities. Furthermore, this authorization system is not dependent on USOS and can be used with any database schema.

The solution presented in this paper is written for DBMS Oracle. Unfortunately SQL standard [6, 7] does not cover things like triggers, so it is impossible to create a portable solution that works with any database that conforms to SQL standard. However, with some effort one can tailor the described authorization system to any DBMS that provides updateable views and INSTEAD triggers.

We designed our solution before Oracle9i with VPD (virtual private database) [8] was released. VPD allows implementing such flexible authorization systems with polices in the form of stored procedures that can be associated with database objects. In our opinion, our solution is more efficient, since in VPD every access to table causes a call to a stored procedure. This is slower than just referencing a view like in our system.

The presented system was built as the result of faulty engineering procedure, i.e. the initial omission of the necessity of authorization system. However, as the effect

we have an innovative solution for the positive scenario. The application programmer creates software for the user that is allowed to do everything. The developer does not care for the access control at all and does not have to include it into the code. He can proceed in this way, because he knows that a transparent authorization system will "customize" the code to the real access rights of the particular user. It is very important because authorization issues concern every little element of any commercial application software. Thus we have a design pattern that can be widely used.

SQL standard [6, 7] includes some mechanisms of access control (roles) but they are insufficient. By means of GRANT commands one can give access rights to the whole table, view or column, but not particular rows. One can say that the solution are views (each user gets the view she can view or modify) and finish discussion here. Many information systems (there is USOS among them) have at least hundreds of users and hundreds of tables that make dozens of thousands of views. The management of such a set of views is not a trivial task. The system presented in this paper allows creating and managing appropriate views easily.

In the described authorization system privileges are grouped into roles. Roles are granted to the users. Roles can have parameters because usually there are many users with the same template of privileges but concerning different areas. The privilege to see all students of a faculty is an example of such a template. One may need to define a template role with the parameter being the faculty name and instantiate it for different faculty. Another necessary feature is the possibility to combine privileges of several roles in one role (e.g. for roles that are needed to serve multi-faculty curricula). In this paper we call it role inheritance. The presented authorization system includes both features—role parameterization and role inheritance. These mechanisms proved very useful in practice.

More precise information can be found in thesis [9]. The code of the solution is available with the distribution of USOS [10]. There one can also find the license information.

2. Possible solutions

We decided to use as much of SQL as it would by possible. We assign privileges to the SQL *roles*. The users are granted privileges through those roles. The user granted more than one role has all privileges of these roles.

A *privilege* (either to see or modify) is defined by a logical *row condition* that limits the set of rows where the operation could be applied and the list of visible or modifiable columns.

After one had defined a role as a set of privileges, she could grant that role many times to users and make another role inherit from that role. This largely enhances the possibility of reuse and the convenience of using the authorization system.

We considered several possible solutions to the problem. One of them was excluded by our initial assumptions—the one with access control executed at the client side (e.g. by forms). This solution is also unacceptable from the security reasons. An intruder could use another interface (e.g. command-line interpreter) to the database and in this way bypass the security checks hardwired into the forms.

The other idea was to add *auxiliary tables* with the information on access rights. A user would be granted views. Each of those views would be the join of the original table and the auxiliary table with the filtering condition dependent on the username. DML operations would be overridden by INSTEAD triggers that would consult auxiliary tables for security issues. This solution seems is inefficient, because every update of the original table requires a corresponding update of the auxiliary table.

The selected solution was based on *views* [11, 12] that present the user with the data she has access to. The user is granted the views that fake the original tables. Such a view is a selection from the original table with the WHERE expression being the corresponding row condition defined for the role of the user. In fact the views have automatically generated names and users have synonyms that point to those views. The name of such a synonym is the same as the name of the original table. As with previous solution DML operations are overridden by INSTEAD triggers with built-in conditions connected with the privileges of a role. In this case the triggers do not consult any additional data. All necessary security information is hardwired into them.

INSTEAD triggers were necessary in this solution because we allow row conditions with subqueries. In SQL standard [6, 7] there is a limit that updateable views must not have subqueries. If we did not use INSTEAD triggers and rely on the view updateability of SQL, the views with more complex conditions (especially with subqueries) would not be updateable.

3. Implementation

All the database objects (tables, views, sequences and stored procedures) under access control belong to one designated user called the *owner of table*. For security reasons the table owner has not CREATE SESSION privilege, so no one can log on into this schema.

Another special user called the *administrator of roles* is given the read (SELECT) and write (INSERT, UPDATE, DELETE and EXECUTE) privileges to the data in all the objects of table owner. The administrator of roles does not have the privilege to modify the structure of the objects of the owner of tables (ALTER). She is granted those privileges with the possibility to propagate them (GRANT OPTION). It the administrator of roles who is the dispenser of privileges for the regular users.

The administrator of roles owns all the database objects used by the authorization system, i.e. views and INSTEAD triggers. When one defines a privilege of a role a view is created in the schema of the administrator of roles. The view is based on the table concerned by the privilege. The users that have this role are granted access to this view in the administrator's schema. In their schemata synonyms are created that point to this view. The synonyms have the same name as the original table concerned by the privilege. This way all the users reference the database objects by the same name.

The administrator of roles has graphical interface that facilitates creation of the views, INSTEAD triggers and synonyms. All these objects are created automatically after the administrator defines the kind of the privilege, row condition and the limiting list of columns.

The user can be granted more than one role. However at the given moment she can exercise privileges of one chosen role. We call it the *default role*. If a user wants to have access rights of some other of her roles, she can switch the default role in her graphical interface.

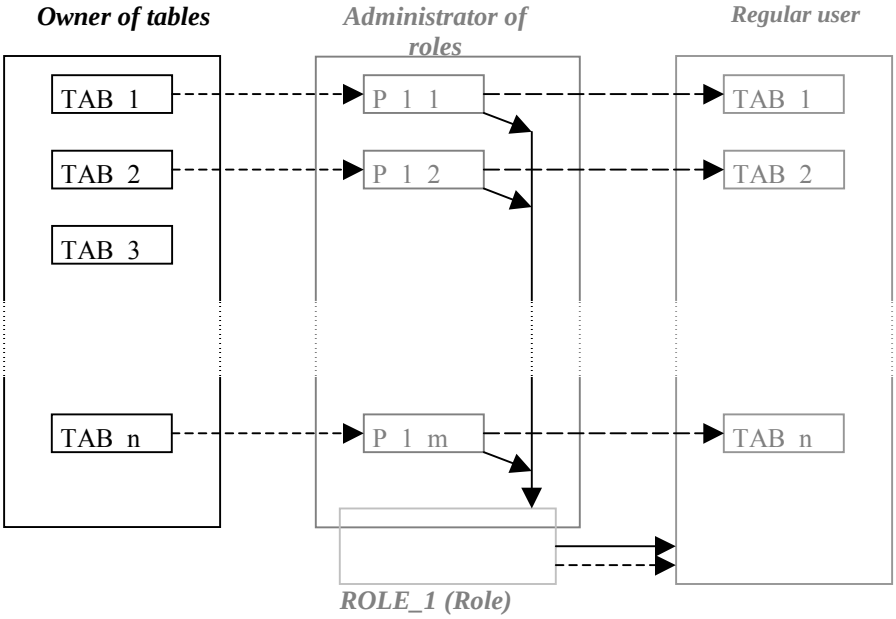
The authorization for all the objects of the owner of the tables is based on the same idea of administrator's mediation. In this chapter we describe the authorization in the most interesting case of the tables. The authorization for other objects is analogous [9, 10].

The users access the tables of the owner of tables by means of the views defined by the administrator of roles. The authorization mechanism is based on the proper construction of these views and INSTEAD triggers installed on them. Such a view is created for every pair role-table (if and only if the role has any privileges to this table). The user has synonyms to all views of her role, so whenever she references a table name, she in fact calls the appropriate view of the administrator of roles.

If a role has the SELECT privilege to the whole table, the view that shows all the data is created and the role is granted the SELECT privilege on this view.

If a role has the SELECT privilege to some subset of the data in the original table (limited by the row condition and/or the list of accessible columns), the created view shows the data from the accessible columns and the NULL values in the non-accessible columns. The row condition becomes the WHERE expression of the view. These NULL columns are required so that the view has the same structure as the original table.

If a role is granted at least one of the write privileges (INSERT, UPDATE and DELETE) to all the data in a table, a view is created with all the data form the original table. This view is very simple (one table and no WHERE expression), thus the view is updateable. The role gets simply the appropriate write privileges to this view.



4. Table privileges

Fig. 1 Access control for tables

If a role is granted INSERT or UPDATE privilege to all the rows but only a subset of columns, this role gets the INSERT or UPDATE privilege to these columns. We use here the native features of DBMS Oracle that allow granting such privileges to a subset of columns.

If a role is granted at least one of the write privileges (INSERT, UPDATE and DELETE) to a subset data in a table defined by the row condition, these privileges are handled by automatically generated INSTEAD triggers. The trigger body includes the row condition. At each update the trigger checks whether the row condition evaluates to false (that means that the update is illegal) and possibly raises an exception. If the update is legal, the trigger introduces it into the original table.

Of course some of those views could be updateable as defined in SQL standard, but we do not make use of it. Every privilege with the row conditions is connected with INSTEAD triggers. This could be seen as a disadvantage. In fact most row conditions in the authorization system deployed for USOS contains subqueries, so this assumption does not diminish the system efficiency.

Let us consider the famous tables EMP and DEPT. A role can insert a row into table EMP if and only if this row references the sales department by its foreign key (EMP.DEPTNO). The row condition for the insert operation will be:

```
exists (select 'x' from DEPT d
  where :new.DEPTNO = d.DEPTNO
    and d.DNAME = 'Sales')
```

In this case the generated INSTEAD trigger will be as follows (`P_564_54531` is the name of the view for table `EMP` and role `SALES_HR`):

```
CREATE TRIGGER P_564_54531_INS_TRI
-- insert trigger for role
-- SALES_HR on view for table EMP
  INSTEAD OF INSERT ON P_564_54531
  FOR EACH ROW
BEGIN
INSERT INTO EMP
  (EMPNO,ENAME,JOB,MGR,HIREDATE,
   SAL,COMM,DEPTNO)
  (select
    :new.EMPNO,:new.ENAME,:new.JOB,
    :new.MGR,:new.HIREDATE,:new.SAL,
    :new.COMM,:new.DEPTNO
   from dual where
        exists (select 'x' from DEPT d
          where :new.DEPTNO = d.DEPTNO
            and d.DNAME = 'Sales');
IF SQL%ROWCOUNT=0 THEN
  RAISE_APPLICATION_ERROR(-20000,'…');
END IF;
END;
```

On Fig. 1 the schema of the access control for tables is shown. The schema of the owner of tables contains tables TAB_1, TAB_2, TAB_3… TAB_n. Objects P_1_1, P_1_2… P_1_m are the views owned by the administrator of roles. TAB_1, TAB_2… TAB_n inside the regular user's schema are her private synonyms pointing to the corresponding views of the administrator of roles. ROLE_1 is a role defined by the administrator.

Dashed arrows between the tables of the owner of tables and the views of the administrator of roles indicate what tables are these views based on. Dashed arrows between these views and the synonyms of the regular user indicate what view these synonyms point to. The solid arrows show the grants—access to views of the administrator is given to role ROLE_1 and this role is granted to the regular user. The dashed arrow between the role and the user means that this role is the default role of the user.

On Fig. 1 role ROLE_1 is granted certain privileges to tables TAB_1, TAB_2…TAB_n. Appropriate views in the administrator's schema has been created (P_1_1, P_1_2… P_1_m). They are based on the corresponding tables. These views have been granted to role ROLE_1. Access rights to these tables are executed by the structure of these views and possibly the INSTEAD triggers installed on them. Although the regular user has not direct privileges to the original tables, she can references their original name, because the synonyms provide the proper mapping between all the objects used by the authorization system.

Role ROLE_1 has no privileges to table TAB_3, thus no view has been created for this role and this table. The user does not have a private synonym for this table as well.

5. Inheritance

The roles are sets of privileges. Data in big databases is usually subdivided into various logical areas and it is reasonable to group privileges the same way. This ought to be done to guarantee security, integrity and logical clarity. Sometimes the user of database should have access to more then one data area at the same time. In the presented authorization system she cannot exercise privileges of more then one role at the same time. Proposals of such solution have already been presented [13, 14].

To solve this problem we introduced hierarchical roles into the system. We let the role's privileges to be inherited by the other roles. A role can inherit from several other roles. The set of privileges of this role is the sum of its privileges and the privileges of the inherited roles. For example, if role A has the select privilege to attributes C1, C2 and C3 of table T and inherits from role B, which has select privileges to attributes C3, C4 and C5 of table T,

then all users having role A should be able to select C1, C2, C3, C4 and C5 attributes from table T. The same idea concerns row conditions. The row condition for a table and a role is the disjunction of row conditions of this role and of all inherited roles for this table. This way we can combine roles into bigger roles. This facilitates reuse and allows building a sound structure of a concrete instance of the authorization system.

The role hierarchy must not contain cycles. Each change of a role definition that is directly or indirectly inherited by some other roles has to be propagated immediately to the objects of inheriting roles. This process might not stop, if the hierarchy has cycles.

6. Parameterization

The presented authorization system also contains the parameterization facilities. Row conditions often contain some constants. These constants are characteristic for certain roles (e.g. the faculty code for a role for a clerk at this faculty). If there were no parameters, the administrator of roles would have a hard time while changing the code or creating the role for a brand new faculty.

In such a situation parameterization can be helpful. A role can have several parameters. These parameters are assigned values in the same role. One could treat these parameters as constants, because they are just symbolic names for constant values in roles. These parameters can be used inside row conditions. Each change of the value of parameter causes rebuilding of administrator's views and triggers connected with row conditions containing the parameter's reference.

We decided to call such constants parameters, because of the possibilities offered by the combination of them and inheritance of roles. A role with parameters behaves like a template and an instance. It can be "called", because it has parameters; it is a stand-alone instance, because all the parameters are bound to values.

When a role inherits from some other role, it can override values of the parameters of the inherited role. The inherited role becomes then a template and the inheriting role is its instance.

Figure 2 shows an example of such an inheritance hierarchy with parameters. We use UML notation with roles as classes and inheritance relations as the dependencies with stereotype «calls». Role A inherits from roles B and C. It assigns `param1` as "LAO" and does not assign anything to `param2` (we indicate it by assignment of NULL to `param2`). Thus role A "calls" both roles B and C with `param1` equal "LAO" and with empty value of `param2`. In the inherited row conditions from B and C parameter `param1` is equal "LAO" but the values of `param2` are retained the original values and are equal to "ELLE" and "RORK" respectively. The row condition for role A and a given table is the disjunction of the row conditions from A (with `param1` = "LAO"), the row condition from B (with `param1` = "LAO" and `param2` ="ELLE") and the row condition from C (with `param1` = "LAO" and `param2` ="RORK").

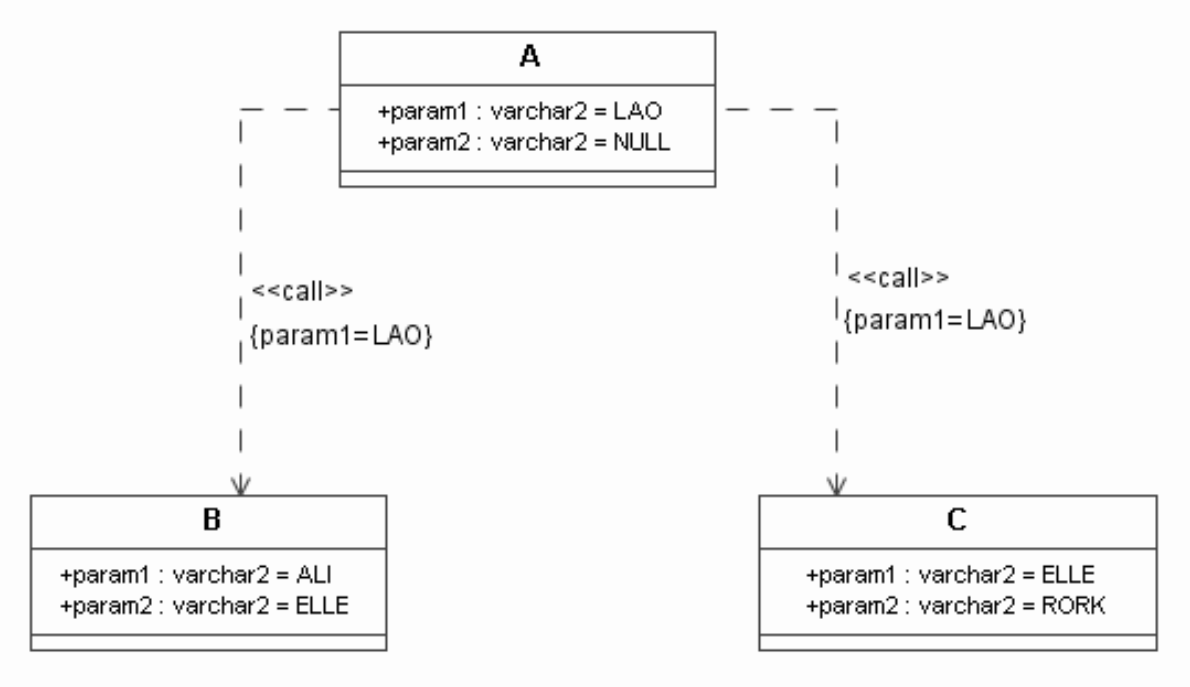


Fig. 2 The inheritance hierarchy and parameters

This way the innocent constants turned into parameters and constitute a powerful facility to structure the roles.

This works on each level of inheritance hierarchy. Of course inheritance relation does not influence the views and triggers of the inherited roles (B and C in our examples), which use the original values of their parameters (`param1` = "ALI" ∧ `param2` ="ELLE" in B and `param1` = "ELLE" ∧ `param2` ="RORK"). The change of the parameter's value in the inherited role may be a change of the definition of the inheriting role; thus it may be propagated down the hierarchy (unless this parameter is immediately overridden).

7. Conclusion

In this paper we presented a flexible database authorization system, which was built as a part system USOS. However, the obtained system is independent of USOS and can be applied to any database schema.

One of the most interesting aspects of this software is the use of INSTEAD triggers as the tool to code updateable views. Such triggers made it possible to update views of any complexity and to abstract from the syntax conditions forced on them by SQL standard. The limitations on updateable views defined in [7] are too severe and such conceived view updateability has marginal meaning. The only general solution is based on INSTEAD triggers that allow any view to be updateable.

The system is very flexible since it contains powerful combination of inheritance and parameterization that allows building clear and sound structures of privileges.

8. Acknowledgement

We are grateful for Janina Mincer-Daszkiewicz who strongly encouraged us to write this paper and present the results of our labor.

References

[1] J. Mincer-Daszkiewicz, USOS: Student Management Information System for Polish Universities, *SAIAC' 2002. Joint Int. Conference on State of The Art in Administrative Computing*, Tartu, Estonia, 2002. See: http://usos.mimuw.edu.pl/tartu.pdf.

[2] J. Mincer-Daszkiewicz, Student Management Information System for Polish Universities, *Eunis 2002, The Eighth Int. Conference of European University Information Systems*, Porto, Portugal, 2002. See: http://usos.mimuw.edu.pl/eunis-2002/eunis2002.pdf

[3] Home page of USOSweb, Warsaw, Poland, http://usosweb.mimuw.edu.pl.

[4] E. Bertino, P. Samarati, S. Jajodia, An Extended Authorization Model for Relational Databases, *IEEE Transactions on Knowledge and Data Engineering (1)*: 1997, 85-101.

[5] R. S. Sandhu, Q.Munawer, How to Do Discretionary Access Control Using Roles. *ACM Workshop on Role-Based Access Control*, 1998: 47-54

[6] C. Date, J. Darwen, *A Guide to the SQL Standard, 4-th Edition* (Addison-Wesley Longman 1997).

[7] International Organization for Standardization (ISO), *Database Language SQL*. ISO/IEC 9075:1992.

[8] K. Browder, M. A. Davidson, *The Virtual Private Database in Oracle 9i R2 Understanding Oracle 9i Security for Service Providers – An Oracle White Paper*, January 2002, http://otn.oracle.com/deploy/security/oracle9iR2/pdf/VPD9ir2twp.pdf

[9] M. Makaroś, *USOS. Roles* (in Polish), M.Sc. thesis. Institute of Informatics, Warsaw University, Warsaw, Poland, 2002. See: http://usos.mimuw.edu.pl/PraceMagisterskie/makaros/makaros.zip.

[10] *USOS Distribution* http://usos.mimuw.edu.pl/Dystrybucje/dystrybucja.html.

[11] S. Barker, A. Rosenthal, Flexible Security Policies in SQL, *DBSec 2001*, 167-180.

[12] D. E. Denning, S. G. Akl, M. Morgenstern, P. G. Neumann, R. R. Schell, M. Heckman: Views for Multilevel Database Security. *IEEE Symposium on Security and Privacy*, 1986, 156-172.

[13] C. Ionita, S. Osborn, Privilege Administration for the Role Graph Model, *DBSec 2002*, 15-25.

[14] S. Osborn, Integrating role graphs: a tool for security integration, *Data & Knowledge Engineering 43 (3)*, 2002, 317-333.