Fine Grained Authorization Through Predicated Grants

Surajit Chaudhuri
Microsoft Corp.
surajitc@microsoft.com

Tanmoy Dutta
Microsoft Corp.
tanmoyd@microsoft.com

S. Sudarshan*
IIT Bombay
sudarsha@cse.iitb.ac.in

Abstract

Authorization in SQL is currently at the level of tables or columns. Many applications need a finer level of control. We propose a model for fine-grained authorization based on adding predicates to authorization grants. Our model supports predicated authorization to specific columns, cell-level authorization with nullification, authorization for function/procedure execution, and grants with grant option. Our model also incorporates other novel features, such as query defined user groups, and authorization groups, which are designed to simplify administration of authorizations. Our model is designed to be a strict generalization of the current SQL authorization mechanism.

1. Introduction

Fine-grained access control, which restricts access to only the information in some rows of a table, and further to only information in certain columns within those rows, is required in practically all database applications. As an example, a HR application has to ensure that employees can see only rows corresponding to their own data, and managers can additionally see some columns (such as salary) of rows corresponding to their employees' data.

Fine-grained authorization is traditionally implemented by application programs, with no role for the database system. There are several drawbacks to the current approach of implementing fine-grained authorization purely in the application layer:

- Authorization checks are distributed over a large body of code, requiring more programmer effort, and increasing the chances of security problems due to programmer or design errors. In contrast, providing support for fine-grained authorization in the database engine could ensure that authorization policies are uniformly applied to all accesses.
- Applications typically connect to the database using a single database user login. Using an operating system analogy, every query runs with administrator (super user) privileges, with respect to all data managed by the application. Since the surface area to be protected is also very large, the potential for damage due to malicious access is high as a result.
- In an application service provider model, information belonging to different organizations may reside in the same relation. An organization may not be willing to place complete faith in the application logic to protect its data, and may desire a higher level of confidence, with database-enforced protection of individual rows [4]. Further, fine-grained authorization is essential if the service provider allows organizations to write their own SQL queries.

The above issues motivate the need for fine-grained access control at the database level.

The current SQL authorization model is coarse-grained in that it grants access to all rows of a table or none at all. A form of fine-grained authorization can be implemented in the current SQL language definition by using views, (or table valued functions) with built-in functions such as userid() and ismemberof() which provide user-specific parameter values. However, this approach requires queries to be phrased against the views rather than on the original table, which may require rewriting significant parts of an application. Further, different queries would have to be written for users with different degrees of authorizations, causing an unacceptable burden on the programmers and complicating authorization administration.

In this paper we present a proposal to extend the SQL authorization model to support fine-grained authorization. Any such extension to SQL must have the following characteristics:

- Clear and simple semantics.
- Compatibility with existing SQL security model, with minimal changes.
- Ease of specification and administration of authorization..
- The ability to deal with (large numbers of) application users, not just a set of fixed database users/roles.
- Low impact on existing application code.

Our proposal is designed to meet the above requirements, and has several novel aspects:

- An extension of the SQL authorization grant model to include predicates. Predicates can be applied on any form of grant, including read and update of rows, and execution of functions and procedures (with predicates on function/procedure parameters). Current SQL authorization is a special case of predicated authorization, with the predicate being "true".
- Column-level authorization, including variants that allow:

* Work performed while on sabbatical at Microsoft Research.

1-4244-0803-2/07/$20.00 ©2007 IEEE

- nullification of values based on predicates, which enables cell-level security [3], and
- Authorization on aggregates, while restricting authorization on the underlying data.

- Mechanisms to support administration of systems with large numbers of application users and database objects including
  - Query-defined user groups.
  - Authorization groups, which allow a group of tuples that together constitute a business object (such as a purchase order) to be granted as a unit.

Together, these mechanisms enable a compact specification of complex authorization policies, including, as special cases, multi-level security and access control lists. Several of the above-mentioned features were motivated by case studies of applications that we carried out. We were able to concisely specify authorization policies for these applications using our proposed constructs.

There has been a fair deal of work on fine-grained authorization in recent years, including two commercial implementations: Oracle's Virtual Private Database (VPD) [4], and Sybase row-level authorization [5]. Both implementations are based on adding predicates to query where clauses, but both are decoupled from traditional SQL authorization. The only earlier work that addresses SQL extensions is Agrawal et al. [1], which proposes SQL extensions in the context of privacy; however, our paper addresses the issue in a significantly more general setting, and supports systems with complex schemas and multiple categories of users. See Section 7 for a description of related work, and their connections with our proposal.

The main goal of this paper is to present to the community a detailed initial proposal for extending SQL to support fine grained authorization, addressing important issues that arise in this context. Our hope is that this will eventually lead to extensions to the SQL standard, through refinement of the proposal.

The rest of the paper is organized as follows. Section 2 outlines the basic components of our authorization model. Section 3 describes authorization on columns, including nullification and aggregate authorization. Section 4 describes user-groups and authorization-groups. Section 5 discusses issues in predicated grants in cases where the grantor has restricted (predicated) access to relevant relations. Section 6 discusses issues such as application authorization and efficiency of implementations. Section 7 describes related work, and Section 8 concludes the paper.

2. Authorization Model Components

Our authorization model extends the authorization model of the SQL:2003 standard, and introduces several new components, including a user context, authorization predicates, and query-defined user groups, which we describe in this and subsequent sections. We use the following schema in our examples.

- employee(empid, name, deptid, addr, phone)
- manager(mgrid, deptid)
- dept(deptid, deptname)

2.1. Application Users and User Contexts

The notion of users in database systems traditionally maps to database logins. In contexts, such as web applications, with large numbers of users, it is infeasible to have a traditional database login for each user, due to space overheads for storing authorization information, and time overheads for session set-up. Instead, applications employ a notion of an application user, which is distinguished from a database user. Fine-grained authorization has to be done in the context of application users, rather than database users.

We assume that the identifier of the current application user and other information, such as the network address from which the user request was received, may be stored in a *user-context*, and made available through functions. Our user context is the equivalent of the "application context" in Oracle VPD. Thus, we assume that a SQL function userId () ,associated with a schema called UserContext, provides the identity of the application user.

Application users must be authenticated, and their identity and other user-context information made available to the database in a secure manner. Mechanisms to do so are straightforward, but outside the scope of our model.[1]

2.2. Predicates in Grants

Predicates can be used in grants as illustrated in the example below, which specifies that each person is granted access to their own employee record:

```
grant select on employee
      where (empid = userId())
      to public
```

Such a predicated grant statement authorizes access only to rows that satisfy the grant predicate. Note that a grant with a **true** predicate is equivalent to a normal (unpredicated) SQL grant.

We initially assume that, as in standard SQL, grants are made to database users or roles. Later, in Section 2.5, we extend the model to support grants to user groups. Although we do not allow granting of privileges to individual application users (which would have high administrative overheads), the predicates in the grant provide the ability to specify per-application-user authorizations.

For example, suppose we wish to grant each department head access to the records of their employees. The following example shows how this permission can be granted to all department heads.

```
grant select on employee E
```

[1] User context information can be securely conveyed to the database by an extension of existing APIs such as ODBC, ADO.NET or JDBC. It is possible to convey the information using SQL commands as well, but such an approach is vulnerable to SQL injection and other attacks.

```
where (E.deptid in
        (select deptid from manages
         where mgrid = userId()))
to public
```

Upto this point, we have implicitly assumed that the grantor has unpredicated access to all the relations involved in the grant, including the relation on which permissions are being granted and any relations in the associated predicate. We relax this restriction later, in Section 5.

2.2.1. Semantics for Queries

Suppose a particular user U has been given the following grant on R, with predicate P:

```
grant select on R where P to U
```

The semantics of the grant is that all uses of R in any queries issued by the database user U are replaced by the expression:

```
(select * from R where P)
```

Therefore, we informally refer to this as the *filter* semantics for predicated authorization. If a user has multiple authorizations with predicates $P_1..P_n$, then the disjunction (P_1v …v P_n) is used in place of P in the above expression. The filter semantics may change the semantics of the query compared to unpredicated authorizations: it may generate a subset of the answers, or, in case aggregation or negation is used in the query, an altogether different answer set, when compared with unpredicated authorization. For example, the filter sematics may give the sum of a subset when the user asks for the sum of a set of tuples. This corresponds to the Truman semantics [6]; an alternative semantics based on query validation is presented in [6], but it has several drawbacks, which we discuss in Section 7. Other database systems that support fine-grained authorization, such as Oracle VPD and Sybase, also follow the filter semantics.

2.2.2. Semantics for Updates

The example below grants all (select, insert, delete and update) authorization to all employee records with dept-id Sales, to the database role SalesDept.

```
grant all on employee
    where deptid ='Sales'
    to SalesDept
```

Assuming this is the only authorization that the role SalesDept has, we require that any tuple inserted, updated, deleted or selected by a user with the SalesDept role must satisfy the predicate. In general, the predicates can be different for different authorizations. For updates, the old and new value of any updated tuple must satisfy the update authorization predicates. Inserts, deletes and updates on tuples that violate the predicate are rejected. Suppose a SalesDept user executes the following update

```
update employee
set phone = '555-1212', deptid = 'Legal'
where empid = '1234'
```

If the employee 1234 was in the Sales department, the user would have authorization to update the phone number, but the update to deptid would fail since the updated tuple fails the authorization predicate (deptid = 'Sales'). However, if the user was additionally granted update authorization with the predicate deptid = 'Legal', the user can update the department value from Sales to Legal.

If an insert/delete/update affects multiple tuples, each tuple that is inserted/deleted/updated must satisfy the authorization predicates; otherwise the transaction must be rolled back. Note also that if the insert/delete/update had an associated subquery possibly referring other relations, the filter semantics is applied to the subquery.

2.3. Authorization on Procedures/Functions

Applications often implement fine-grain authorization on function/procedure invocations by adding a check at the beginning of the function/procedure. However, enforcement at the database layer (in lieu of, or in addition to, authorization checking at application layer) provides a stronger guarantee. Execution authorization is already part of the SQL standard. We extend it here by specifying predicates on parameter values, as illustrated below:

```
grant execute on getsalary(userid)
    where (userid = userId()) to employeeGrp
```

We assume that employeeGrp is a role that is granted to all employees. Thus each employee is authorized to execute the getsalary function, provided the parameter value equals their user id. If the predicate does not evaluate to true, the function is not invoked. In such a case, one possible action is to return a null value to the caller. Similarly, in the case of procedures, if the predicate does not evaluate to true, the procedure is not invoked, and one possible action is to set output-only parameters to null, and leave input-output parameters unchanged. An alternative action in either situation above is to raise an exception when the authorization predicates fail. The approach of returning null is consistent with the filter semantics.

In SQL, stored procedures can run either

- under the privileges of the creator of the stored procedure, or
- under the privileges of the invoker.

Note that the creator or invoker may have only predicated access to some of the relations used in the procedure. If a stored procedure is invoked under different privileges, the queries in the stored procedures may be rewritten differently for different invocations.

2.4. Revocation

In general, a predicated grant has the structure:

```
grant <Perm> on <obj>
where (<Pred>) to <subj> [as <auth-name>]
```

Authorization names are automatically generated if they are not specified, and can be found by querying the set of authorizations.

An authorization can be revoked by name: the following example revokes the authorization named A1.

```
revoke A1 from employeeGrp
```

Revocation of a named authorization would allow other authorizations to still be valid on the same object. All authorizations that were explicitly made on an object to a grantee can be revoked by a standard SQL revoke. For example the statement

```
revoke select on employee
    from employeeGrp
```

removes all authorizations made on employee to employeeGrp. We do not, however, support predicates in revocations. Revocation with predicates introduces additional complexities and it is therefore not part of the current proposal.

2.5. Query-Defined User Groups

Consider a permission that must be granted to all managers, but not to other employees. In SQL it is possible to create a manager role and grant the role to each manager individually. However, this requires a great deal of maintenance effort, since a grant/revoke of the role must be made each time there is a change in the set of managers. Moreover, the role assignment replicates information already present in the database, namely who are managers.

Allowing grants to be made to application users would make authorization checking rather expensive, since each user may have a completely different authorization predicate, and there may be millions of application users. Therefore, as mentioned earlier in Section 2.2, we do not allow grants to be made to application users.

One way around this problem is to make grants to public, and encode all checking in the authorization predicate, as we did in earlier examples. Specifically, consider the example from Section 2.2 where the grant predicate checks if the userid is of a manager, and if not returns the empty set of tuples. In the common case where many or most users are not managers, checking the predicate would cause an unnecessary overhead on every access to the employee relation. The notion of user groups helps tackle the above problems.

Query-defined user groups (or just **groups**, for short) are groups of application users defined using a query. Membership of the group is defined dynamically, based on values in the database, by means of a query. Specifically, a group has an associated query that returns the set of application user-ids that belong to the group. For example, the following creates a group called managers, containing every mgrid in the manager relation:

```
create group managerGrp as
        (select mgrid from manager)
```

A user can belong to multiple groups. As far as the database system is concerned, a userid is just a string value (found by calling the function userId()) which is used to lookup the groups that the user belongs to.

Grants can be made to groups, just as they can be made to users or roles.

```
grant select on employee
where (deptid in (select deptid from managers
                  where mgrid = userId()))
to managerGrp
```

Note that the above grant is identical to one we saw earlier, except that the above grant is to managerGrp, not to public. The predicate thus will not be added to query filters for users who are not in the managerGrp.

Just as it is possible to grant a role a user, it is also possible to grant a role to a user group. All group members then acquire the privileges available to the role.

3. Authorization on Columns

In SQL, permissions may be granted on specified columns, instead of being granted to all columns of a relation. We extend that model by allowing predicates to be specified on such grants. For example to make names of all sales department people visible to all users, one could use the following grant.

```
grant select on employee(name)
  where (dept ='sales') to public
```

If this is the only available authorization, a query that accesses only the column 'name' would see all tuples corresponding to the 'sales' department, while a query that accesses any other column would be rejected.

In general, a user may have multiple grants on different columns of a relation, with different predicates, and a query may access columns covered by different authorizations. There are several possible models for the semantics of such grants:

- If a query accesses multiple columns of a relation, use only authorizations that cover all accessed. Thus, if a user has a particular authorization on column A, another on column B and a third on columns A and B, only the authorization that covers A and B can be used for a query that accesses A and B.
- Allow different columns accessed in a query to be covered by different authorizations, but only return rows that satisfy all the predicates associated with the authorizations that cover those columns. For example, if we have two grants, one that allows access to column A under condition Pa and another that grants access to column B under condition Pb. Then a query that accesses columns A and B should be allowed to access only those rows that satisfy Pa and Pb and return all (A,B) pairs for those qualifying rows..
- Allow different columns accessed in a query to be covered by different authorizations, and return rows where at least one column satisfies the authorization predicate on the column. However, nullify cells for which none of the applicable authorization predicates evaluate to true.

The first model is rather restrictive. The second model is consistent with column authorizations in SQL; we adopt it as the default model, and describe it in Section 3.1. The third model using nullification is also useful in many settings, and we allow it to be specified using additional syntax; we describe this model in Section 3.2.

3.1. Column Authorization Without Nullification

In this semantics, grants on multiple columns are viewed as a collection of grants, one on each column. Thus a grant on R(A,B) is equivalent to two grants, one on R(A) and another on R(B).

Multiple grants on a column are treated as defining a disjunctive condition. That is, the corresponding column can be accessed provided one of the relevant predicates is satisfied. Thus, the two grants "R(A,B) **where** (P1)" and "R(A) **where** (P2)" are equivalent to the two grants "R(A) **where** (P1 or P2)" and "R(B) **where** (P1)".

A query that accesses multiple columns sees only rows that satisfy the conjunction of the predicates for the grant on each column.

Note that the above behavior is non-monotonic, in that if a query accesses more columns, it may get potentially fewer tuples. In contrast, without predicates, if a query accesses more columns, it may get rejected (and return no tuples). In contrast, column authorization with nullification can have monotonic behavior, as explained in the next subsection..

3.2. Cell-Level Authorization with Nullification

Consider the following authorization scenario: allow access to the address attribute of employees who have 'opted-in' to allow their addresses to be made public, but return a null value for the address attribute of all other employees

To handle such scenarios, a grant can specify else nullify as illustrated below

```
grant select on employee(addr)
    where (P1) else nullify to public
grant select on employee(phone)
    where (P2) else nullify to public
grant select on employee(empid)
    where (P0) to public
```

P0, P1 and P2 denote predicates (left unspecified in the example). The else nullify clause can only be specified on columns whose types permit null values; it cannot be used on primary key columns, for example.

If a grant with nullify is specified on a column, queries can access that column, but the value returned for a row will be null unless the predicate is true for at least one of the grants. Thus, if predicates P1 (resp. P2) in the above grants evaluate to false for a particular row for a particular user, that user will see null values for the address (resp. phone) attribute of the row. On the other hand, if a query accesses an attribute on which else nullify is not specified, such as empid in the above example, the entire tuple will become inaccessible (including attributes for which else nullify has been specified).

One effect of the above semantics is that users may get rows where all referenced columns are null. We follow the null-row suppression model of [3], which eliminates rows that are null on all attributes. Such cell-level nullification is required to support privacy policies such as P3P; see for example, [1],[3].

3.3. Aggregate Authorizations

Authorization can be granted on aggregated values, instead of individual values. For example, if we wish to allow a salesperson to see the aggregate of sales in their region, we can use the following grant.

```
grant select on sales(region, category,
                      anyagg(units), anyagg(price))
where (region = getUserRegion())
to salesGrp
```

We assume that the function getUserRegion() returns the sales region corresponding to the current user, and salesGrp is a role (or user group to which salespersons belong). The aggregate anyagg stands for the SQL aggregate functions min, max, sum, count, avg, It is also possible to allow a set of the above aggregates to be specified, for example, [**sum,avg**](units). Additional aggregate functions can be supported as well, but we don't discuss those extensions in this paper.

Note that the above grant is similar to grants on specific columns, except that an aggregate authorization is only applicable to a query only if (a) the query accesses and groups-by only the columns listed in the authorization, and further (b) columns listed only within an aggregate function in the grant are used only within a corresponding aggregate function in the query (expressions, e.g. units/price, are not allowed on such columns).

Given the above grant, a query

```
select region, sum(units) from sales
group by region
```

submitted by a sales user would retrieve total sales of the region he is responsible for; a query "**select sum**(units) **from** sales" would return the same aggregate value, instead of the total sales across all regions. However, suppose we give the additional authorization

```
grant select on sales(anyagg(units),
                      anyagg(price))
to salesGrp
```

Then a query "**select sum**(units) **from** sales" submitted by a sales user would return the total sales across all regions. If there are multiple aggregate authorizations applicable to a query, their conditions get combined by disjunction.

Note that unlike regular column level authorization, we cannot combine aggregate authorizations across different columns, since that can reveal more fine-grained informa-

tion than the individual authorizations provide. For example, the grants

```
grant select on sales(region, sum(units),
        sum(price))
grant select on sales(category, sum(units),
        sum(price))
```

do not imply

```
grant select on sales(region, category,
        sum(units), sum(price))
```

3.4. Semantics of Multiple Authorizations

In general, there can be multiple grants on a relation, including multiple grants on a single column. We define the semantics of multiple grants on relation R by defining an authorized view of R under a given set of authorizations. Note that the semantics is in the context of a query, which defines the set of columns accessed. For each relation R accessed by Q, let C_Q be the set of columns of R accessed by Q.[2]

The authorized view Vr of relation R can be definedas:

```
select L
from R
where  Pa and Pb
```

Where

- For each column Ci in C_Q that does not have any else nullify authorization, define Pi as the OR of predicates ionality. in all grants authorizing Ci; include in Pi any aggregate authorizations applicable to Q. If any column in C_Q has no authorizations, the query is rejected as unauthorized. Let Pa be the AND of all the resulting Pi's. If Pa is empty, set Pa to TRUE.
- L is defined as follows: for each column Ci in C_Q, L contains either just Ci (if there are no else nullify rules on Ci), or

  ```
  (case  when Oi then Ci
         else null
   end) as Ci
  ```

  where Oi is the disjunction of all authorizations on column Ci
- The clause **and** Pb is required to implement null-row suppression, if all columns in C_Q have an else nullify clause. Pb is defined as the OR of the authorization predicates on all columns in C_Q.

Given the earlier defined authorization on employee, for a query that accessed empid and phone, the resultant view would be

```
select empid,  (case when P2 then phone
                     else null end) as phone
from employee
where P0
```

While for a query that accesses address and phone number, the view would be

```
select (case when P1 then addr
             else null end) as addr,
       (case when P2 then phone
             else null end) as phone
from employee
where true and (P1 or P2)
```

The above view can be alternatively defined using outer-joins on the primary key column, as described in [3].

4. Authorization Administration Features

Query defined user groups, introduced in Section 2.5, enables administration of authorizations without having to instantiate authorization for each user. In this section, we further build on user groups. We also introduce the notion of authorization groups, which allow related groups of objects to be treated as a single unit for authorization..

4.1. Query-Defined User Groups Revisited

Hierarchies of groups are conceptually straight-forward: to have group B inherit all members of group A, the group A can be used in the query defining group B as below.

```
create group A as (…)
create group B as  A union (…)
```

User groups can be thought of as roles that are granted to application users by means of membership rules (defined by queries); in contrast, SQL roles are granted to database users (or other roles) explicitly by individual grant statements. Although our discussion treats groups as distinct from roles, it is possible to integrate the two, by extending the role grant mechanism to allow query-defined membership of application users. Such an integration must be carefully designed to be faithful to SQL semantics of roles.

From the view point of efficient authorization checking, it is a good idea to materialize with each application userid the set of groups to which the user belongs, which can be done easily by materializing the queries defining the user groups.[3]

However, note that since groups are defined by queries, it is possible for a user's group membership to change during a user session..

User-group membership gives additional authorizations to a query, in addition to those that have been granted to the database login under which the query is run. It is advisable to provide minimal authorizations to the database

[2]. A formal definition of the set of columns accessed by a query is presented in [9]. The special case of C_Q being empty (e.g. **select** 1 **from** R) poses problems even for regular SQL column authorization without predicates. We follow the (somewhat arbitrary) SQL Server approach of replacing an empty C_Q by the set of all columns.

[3] The queries defining groups may have to be constrained in their expressivity, in order to ensure that they can be efficiently maintained as a materialized view. This should not pose a problem since group definitions are usually not very complex.

login used by an application, and provide other authorizations through grants to user groups.

The authorization to create/modify/delete the definition of a user group is treated in the same fashion as the authorization to create or delete roles.

We note that a concept of groups already exists in SQL Server 2005 and Oracle, but group membership is externally determined from LDAP/Active Directory. Such externally defined group information can be made available through the user context and used just like query-defined user groups. The notion of query-defined user groups is widely used in LDAP, but has not been part of SQL.

4.2. Authorization Groups

Granting of permissions in the real world is often done with respect to business objects, such as medical reports or purchase orders. Each such conceptual object may span multiple rows across multiple tables in the database.

Authorization groups define a set of authorizations on a group of related objects. Each authorization group must have a root relation, whose purpose is explained shortly. Each authorization in a group may be predicated on the authorization of other objects in the group, so long as the dependency is acyclic.

```
create authorization select_purchaseorder
  with root order O as (
     select on order O,
     select on lineitem L where
              (L.order_id=O.order_id)),
     select on part P where
              (P.part_id=L.part_id),
     select on partsupp PS where
              (PS.part_id = P.part_id),
     select on supplier S where
             (S.supplier_id = PS.supplier_id))
```

Note that each component of the authorization (including the authorization on the root object, which is the relation order in the above example) can be predicated.

Authorization groups may include authorization on other authorization groups, creating a (non-recursive) hierarchy of groups as illustrated below.

```
create authorization sel_update_purchaseorder
  with root order O as (
     update on order O,
     update, insert, delete on lineitem L
          where (L.order_id = order.order_id),
     select_purchaseorder O2
          where (O2.order_id = O.order_id));
```

When an authorization group is granted to a user/user-group or role, it can be predicated further, using predicates on the root relation, as illustrated below (assuming employeeGrp has been defined earlier). The following grants allow purchase orders to be viewed and updated by employees who made the purchase, and by those employee's managers.

```
grant select_purchaseorder
  where(purchaser_id = userId())
  to employeeGrp
grant sel_update_purchaseorder
  where (purchaser_id in
       (select user_id from employee, manager
        where employee.deptid=manager.deptid
            and  manager.mgrid = userId()))
  to managerGrp
```

Authorization groups can be expanded out as a set of authorizations, so they do not introduce extra expressive power. However, they can greatly simplify the task of authorization administration. Specifying equivalent authorizations without authorization groups would require multiple grants, each with a complex subquery, and with significant overlap between the subqueries.

5. Stacked Grants

We had assumed earlier that the grantor of a grant had unpredicated access to all relations involved in the grant (the relation on which the grant is being made, as well as all relations in the grant predicate). This is a reasonable assumption in many cases since fine-grained authorization policies will be set by a security administrator, and not by users. However, if a hierarchy of administrators with different rights is to be supported, we must address the issue of further granting of predicated grants, which we refer to as *stacked grants*.

5.1. Acyclic Stacked Grants

The following is an example of a stacked grant.

```
grant select on R where P1 to A
  with grant option
```

and user A executes

```
grant select on R where P2 to B
```

In this section we assume there are no cycles in the chain of grants (we define grant acyclicity formally in Section 5.2, taking into account grants to public).

We now discuss the semantics of such stacked grants. Two properties have to be satisfied by any scheme that allows a restricted grant to be passed on:

1. A grantor A can only pass on authorizations that had been granted **with grant option** to the grantor, and
2. The grant predicate cannot reveal information to the grantee B that was not visible to the grantor A, or was not granted with grant option to A. (Note, however, that the grant predicate can involve authorizations available to A, even if the authorizations are not granted to B.)

In the above example, the naïve approach of giving to B select authorization predicated by (P1 $\wedge$ P2) would be incorrect, since P1 is only authorized to see data satisfying P1 *with values from his/her user context*. Suppose P1 is of the form "empid=userId()", the userId() value for A is 1234, and that of B is 2345. Then A is authorized only to

access (and grant access to) tuples with empid='1234', whereas the naïve approach would allow A to grant B authorization with empid='2345'.

Giving B authorization predicated by (P1' ∧ P2), where P1' is the result of replacing instances of userId() in P1 by the user-id of A (and similarly for other user-context functions) is incorrect as well. This is because P2 may include a subquery, and unless the relations it uses are filtered by authorizations available to A, it may reveal information to B that A is not authorized to view.

The semantics that assigns to B select authorization on R predicated by (P1' ∧ P2') where P1' is as before, and P2' is the result of replacing relations in P2 by the views available to user A, is intuitive, and easy to implement.

However, it has a subtle form of leakage, since predicate P2' may reveal information to B which A had not received with grant option. For example, if A had select authorization with grant option on employee names, and select without grant option on employees working on a secret project. A could grant B select on all employees, predicated on their working on the secret project; B has thus been granted (at least partial) access to information that A was not permitted to pass on.

This subtle form of leakage is acceptable in most situations as an adequate level of consistency. In cases where it is not acceptable, predicate P2' would have to be formed by using only authorizations available to A with grant options, instead of using all authorizations available to A.

Note also that if the authorizations granted to A change, so should the authorizations that A has granted to B. Thus, at any point in time the grant predicates for a grant with grantor A must be based on the current authorizations available to A. For example, if A had select authorization on R predicated by P1 (with grant option), and passed on this authorization to B. If the authorization available to A changes, with P1 replaced by P2, the tuples available to B change correspondingly. This is in keeping with authorization propagation in standard SQL.

5.2. Cycles in Stacked Grants

Grants in standard SQL can contain cycles, which are relatively easy to handle without predicates. However, the semantics of grants gets very complicated if there are cycles of predicated grants. The following example illustrates problems due to cycles in predicated grants..

1. DBA grants **select on** R and
   select on S (**where** (S.X < 100)
   to A
2. DBA grants **select on** S and
   select on R **where** (R.X <= 100)
   to B
3. A grants **select on** R
   where (**exists**(**select** * **from** S
   where S.X +1 >= R.X))
   to C
4. B grants **select on** S
   where (**exists**(**select** * **from** R
   where R.X +1 >= S.X))
   to C
5. C grants **select on** S to A
6. C grants **select on** R to B

The set of rows accessible to A, B and C is thus defined recursively, which would make efficient implementation rather difficult. To further complicate matters, a grant predicate may in general be non-monotonic; a grant is non-monotonic when an addition to one relation can cause a decrease in the granted permissions on another relation. For example, if the grant from A to C used 'not exists' instead of 'exists', the grant predicate is non-monotonic; non-monotonicity may also arise due to aggregation. If there is a grant cycle involving such a non-monotonic predicate, the semantics of the grant is hard to define.

Note that cyclicity may be implicit: in the above example, if C were **public**, then grants 5 and 6 above are implicit, and the cyclicity problem occurs although there are no explicit cycles in the grants.

Grant acyclicity condition: The simplest solution to the cyclic predicated grant problem is to disallow cycles in authorization grants. Formally, we define an authorization graph as follows. Nodes in the graph are of the form (subject, object) where subjects are users/groups/roles and objects are relations/procedures. The graph edges are derived from grants as follows: let A be the grantor and B the grantee, R be the granted relation and Si be relations used in the grant predicate. Then there is an edge from (A,R) $\rightarrow_{\gamma}$ (B,R), as well as an edge (A, Si) $\rightarrow_{\theta}$ (B, R) for each Si used in the grant predicate. To account for grants to public, we create a node corresponding to public, and add edges labeled γ from (public, T) to (A, T) for each subject A (other than DBA) and each object T.

Grants must check for cycles in the above graph, and any grant that creates a cycle must be disallowed. Note that the grant that completes a cycle may not even be predicated, but the presence of other predicated grants in a cycle can cause cyclicity.

A weaker form of the acyclicity condition can be defined as follows: when checking for cycles we can drop edges from (A,R) to (B,R) if A has unpredicated access to R and the grant to B was unpredicated as well. Such edges cannot cause any change in the permissions on R available to B.

6. Discussion

In this section, we discuss several considerations in adopting our SQL authorization model proposal.

6.1. Other SQL Authorization Mechanisms

Our proposed mechanisms are orthogonal to view authorization. Authorizations to views can be predicated, and can be granted to query defined user groups.

Reference authorization and schema-level authorizations are not meaningful for application/database users who have restricted views of data. Therefore, we did not consider predicated version of such authorizations.

Access to metadata tables is highly restricted in the SQL standard, since SQL currently lacks row-level authorization. Predicated authorization has the potential to be used as a mechanism to give restricted access to such metadata.

6.2. Other Security Models

The multi-level security (MLS) model is used in certain high-security applications, while access control lists (ACLs) are used extensively in file systems. Predicated grants can be used to implement both MLS and ACL in a straightforward way, but can be useful in significantly generalizing both these models. We omit details for lack of space.

6.3. Query Optimization

Authorization predicates added to queries are often redundant, since queries typically only attempt to access authorized information. Techniques for detecting and removing redundant authorization checks are described in Kabra et al. [2].

Query modification to implement fine-grained authorization must be done each time a query is submitted, which can have a non-trivial cost. To reduce this cost, we can cache the rewritten query Q' derived from Q, as well as the recompiled plan. We reuse Q' when Q is submitted again, provided it is submitted by a user with the same or an equivalent set of authorizations. A sufficient condition for this is that the user belongs to the same set of query-defined user groups; the condition can be refined taking into accounts only groups with authorizations relevant to Q. Caching of optimized query plans can be done as usual on the rewritten query.

6.4. Application Level Authorization

Applications often implement their own fine-grained authorization manager, which is used to decide what web pages or user controls/interfaces should be shown to a particular user, as well as to check for authorization before executing each externally invokable procedure. With our authorization extensions, organizations can now store their policies in the database as part of the database authorization system, allowing central administration, and uniform enforcement, across different applications that access the same data,.

Application level procedures can be modeled in the database as external procedures, and the predicated authorization scheme for procedures proposed here can potentially be used to handle authorization of the application procedures. Since the procedures are externally invoked, our authorization mechanism cannot enforce authorization, but our model permits applications to query authorization information stored in the database, to check if a procedure execution is authorized.

Querying of authorizations can also be very useful to answer queries such as "who is authorized to view salary information for employee X", or "is Y authorized to execute the create employee procedure with department=CS"? The design of a meta-data schema for authorization information, which would enable queries such as those discussed above, is an area of future work.

In our model, grants are made by database users/administrators or roles. We do not expect application users to grant privileges at the level of the database schema. However, delegation of authorization (see, e.g. [10]) is an important requirement for many applications. Granting of privileges by application users can form the underlying basis for supporting delegation. The issue of predicates in delegation is outside the scope of this paper, but is an important area for future work.

7. Related Work

The most closely related work is Oracle's Virtual Private Database (VPD) model [4]. VPD allows the system administrator to specify functions for each relation (different functions can be specified for different modes of access). The functions can take the application context as input, and return a predicate as a string; the strings generated from the different relations in a query are ANDed to the where clause of the query.

VPD was an early effort in the area, but it has several limitations, which our model addresses. In particular, policy specification is decoupled from the SQL grant model in VPD, whereas they are integrated in our proposal. We believe our approach will also be more efficient, since it avoids the overheads of calling policy functions on each query; VPD has a mechanism to cache policy function results, but caching is applicable only if the function is guaranteed to return the same result for all users.

The model we propose allows significantly more caching and reuse of rewritten queries. Authorization predicates in our model can include calls to user-defined functions, enabling the full power of a programming language to be used when required, but without having to pay the higher cost when simpler policies are deployed..

Our proposal is designed to simplify administration of complex authorization schemes. The notion of query-defined user groups, which plays a key role in this task, is not present in VPD, nor is the concept of an authorization group. VPD (as of Oracle 10g) also does not have a mechanism for predicate based authorization of function and procedure calls. A form of nullification is supported by VPD in Oracle 10g, but by a more complex scheme.

The policy based security management feature of Sybase Adaptive Server Enterprise [5] allows predicates to be associated with columns of tables. Different policies can be specified on different columns, and are automatically

combined using OR or AND (as specified with the policy) and added to the query where clause. To the best of our knowledge, our model is a strict generalization of their scheme. Their model does not support any column authorization, or features designed to simplify administration, such as user groups and authorization groups.

SQL Server 2005 Analysis Service provides a form of row-level authorization on aggregate results. However, their authorization model is independent of the database authorization model, and is more restrictive than ours.

The approach presented here, as also those of Oracle VPD and Sybase row-level authorization, are based, at their core, on the idea of providing a per-user view of each relation, filtered by predicates (called a Truman model, in the terminology of [6]). As noted in [6], the predicates added by the filtering (Truman) model can change a query result, resulting in misleading/erroneous answers to a user query. The non-Truman model described in [6], on the other hand, guarantees correctness; that is, if a query is accepted, it will give the same result as if the user had full authorizations on all relations. However, the non-Truman model requires a powerful query inferencing mechanism. Since such inferencing is not decidable in general, implementations would heuristics, and a query that is accepted by one database implementation may be rejected by another (perhaps even a different version of the same database system). Such unpredictability is highly undesirable for applications; we have therefore followed the filtering model.

Cell-level authorization is described by LeFevre et al. [3], along with a study of alternative implementation techniques, and optimization techniques. However, their technique is restricted to handling privacy policies, and does not constitute a general purpose authorization mechanism. Our nullification component follows the "query semantics" model of [3]. A proposal to use predicated grants to manage cell-level authorization is described by Agrawal et al. [1]. Their proposal shares with us predicated grant and cell-level authorization with nullification features of our proposal. However, they do not consider any of our other features, such as aggregate authorization, user groups, authorization groups, and interaction with other SQL authorization components. Rosenthal and Sciore [8] propose the use of predicates in grants, to control not only what data can be seen, but even to whom further grants can be made. However, the predicates they consider are simple predicates, based for example on environment conditions such as time-of-day.

Forms of redundancy removal are present in commercial optimizers, and also discussed in Kabra et al. [2]. Techniques from [6] and [2] can help check whether or not a query is semantically affected by authorization rewriting. Kabra et al. [2] also address the problem of information leakage due to user-defined functions with side effects, exceptions and error messages, and discuss how to get optimal "safe" plans.

8. Conclusions

We presented a comprehensive proposal for extending SQL's authorization model to support fine-grained authorization. We have carried out case studies of two applications, and found that our proposed authorization schemes could concisely represent authorizations for both these applications.

The next step is to initiate a discussion among database vendors and application developers to refine the proposal, and reach a consensus on the SQL extensions. Although elements of the proposal have been implemented in prototypes, a full fledged reference implementation needs to be developed.

Much work remains to be done in the area of managing authorizations. Extensions to efficiently handle hierarchies of various types, such as organizational hierarchies, user hierarchies and user-interface hierarchies are required, as is database support for application level authorization. Integration of database and application level authorization remains an important longer term goal.

References

[1] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, W. Rjaibi: Extending Relational Database Systems to Automatically Enforce Privacy Policies. In ICDE, pages 1013–1022, 2005.

[2] G. Kabra, R. Ramamurthy and S. Sudarshan, Redundancy and Information Leakage in Fine-Grained Access Control, SIGMOD 2006.

[3] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu and D. DeWitt, Limiting disclosure in Hippocratic databases, In VLDB, 2004

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

[5] New Security Features in Sybase Adaptive Server Enterprise. Sybase Technical White Paper, 2003.

[6] S. Rizvi, A. Mendelzon, S. Sudarshan and P. Roy, Extending query rewriting techniques for fine-grained access control. In SIGMOD, 2004

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

[8] A. Rosenthal and E. Sciore. Extending SQL's Grant and Revoke Operations, to Limit and Reactivate Privileges. IFIP Workshop on Database Security, 2000.

[9] A. Rosenthal and E. Sciore. Abstracting and Refining Authorization in SQL. In Secure Data Management (SDM) workshop, VLDB 2004.

[10] Xinwen Zhang, Sejong Oh, and Ravi Sandhu, Access Control Models and Mechanisms: PBDM: a flexible delegation model in RBAC, Procs. 8th ACM Symp. On Access Control Models and Technologies, June 2003.