Extending SQL's Grant Operation to Limit Privileges

Arnon Rosenthal, Edward Sciore
MITRE Corporation and Boston College

Abstract: Privileges in standard SQL are unconditional, forcing the grantor to trust the recipient's discretion completely. We propose an extension to the SQL grant/revoke security model that allows a grantor to impose limitations on how the received privilege may be used. This extension also has a non-traditional implication for view security. Although our examples are from DBMSs, the results (other than the treatment of views) apply to arbitrary sets of privileges, and to systems without a query language.

keywords: Access controls, SQL, limited privilege, Grant/Revoke

1. INTRODUCTION

Recent database security research has had distressingly little influence on DBMS vendors. The SQL security model has had few extensions in the past 20 years, except for the recent addition of role-based access controls. This paper extends SQL grant/revoke semantics to include a *privilege-limitation mechanism.* Our goal is to present a model which has a small number of new constructs (and thus has a reasonable chance at vendor implementation and adoption), but covers significant unmet security needs.

In standard SQL, a user who has a privilege is able to use it in any circumstance. By attaching limitation predicates to a grant, we refine this "all or nothing" mechanism. For example, a limitation predicate may restrict a command to certain days or hours, to members or non-members of certain groups, to users above a certain rank, to tables above a threshold size, or to requests that are received along a trusted path.

Limitations seem particularly important in large enterprise or multi-enterprise systems, where it may be difficult to impose sanctions for improper use of a privilege. They also can reduce the scope for damage by an intruder who misuses legitimate users' privileges.

Our privilege limitation model is motivated by two principles:

- The system should have a unified, flexible means of limiting privileges, rather than multiple overlapping mechanisms.
- The ability to grant (and to limit grants of others) should respect the natural chains of authority.

The first principle implies that "revoking a privilege" and "imposing additional limits on an existing grant" are facets of the same concept. For example, imposing a limitation predicate of *false* should be equivalent to revoking the privilege. More generally, modifying the predicate of an existing grant should be equivalent to granting the privilege with the new predicate and revoking the old one. We thus aim to simplify [Bert99], which has separate treatments for SQL-like cascading revoke of entire privileges (without reactivation) and negative authorizations (reactivate-able partial limitations, without cascade).

The second principle guides how grant authority is passed. A user, when granting a privilege, must pass on at least as many limitations as he himself has. Moreover, by modifying the limitations on an existing grant G, a user can propagate those modifications to all grants emanating from G. For example, the creator of a table in a distributed system might authorize remote administrators to grant access to their users, subject to some general limitations; these administrators might further limit access by individual users, as appropriate for each site. If the creator subsequently decides that additional limitations are necessary, he only needs to modify his grants to the remote administrators.

The second principle also implies that a subject x can limit *only* those grants that emanate from a grant he has made. If user y has received grant authority independently of x, then x cannot restrict y's use of that power. To see the need for this principle, imagine that the "Vote" privilege has been granted (with grant option) to the head of each United Nations delegation. Imagine the denial-of-service risk if, as in [Bert99], each could impose limitations on others.

Section 2 describes the principles that underly our approach, and extends SQL grant syntax to allow limitation predicates. Section 3 defines and illustrates the semantics of predicate-limited grants. Section 4 extends the theory to views. Section 5 briefly shows how our model addresses many of the needs perceived in previous work. Section 6 summarizes and presents open research problems.

2. ACCESS CONTROL BASICS

We present our new model gradually. In this section we state the SQL model in terminology that will be useful in the general case. We also show how SQL grants are a special case of limited grants.

A *subject* is a user, group, or role. We treat each subject as atomic; inheritance among subjects (e.g., from a group to its members) is left for future research.

A database object is a portion of the database that requires security control, such as a table or procedure. Each database object has a set of operations. For example, table T will have operations such as “insert into T”, “select from T”, etc.

There are two kinds of *action* associated with an operation: *executing* the operation, and *granting* an authorization for it. A subject issues a *command* to tell the system to perform an action.

Standard SQL has two forms of the grant command. The first form,

grant θ **to** s

authorizes s to perform execute commands for operation θ. The second form,

grant θ **to** s **with grant option**

authorizes s to perform both execute and grant commands for θ.

To specify grants with limitations, we extend the syntax to include two (optional) predicates:

grant θ **to** s [**executeif** P_1] [**grantif** P_2]

P_1, called the *execute*-predicate (or *exec_pred*) of the grant, restricts the conditions under which subject s can execute operation θ. P_2, the *grantonward-predicate (or gr_pred)* of the grant, restricts the conditions under which s can grant θ onward to others.

Each operation θ has an *authorization graph*, a rooted labeled digraph that represents the current (non-revoked) grants for θ. We denote it AG_θ, or just AG if θ is implied. The graph has one node for each subject; the root node corresponds to the creator of θ. There is one edge e_G for each unrevoked grant command G (and we refer interchangeably to G or e_G); a grant by s_1 to s_2 corresponds to an edge from node s_1 to s_2. Edge e_G is labeled with G's two predicates.

A *chain* to *s* is an acyclic path $C \equiv <G_1,\ldots,G_n>$ in **AG** such that s_1 is the root, each G_i goes from s_i to s_{i+1}, and s_{n+1} is s. By convention, the predicates associated with G_i are denoted gr_pred_i and $exec\_pred_i$.

In the general case, G's predicates can reference G's command state (as discussed in Section 3). In the special case of grants in standard SQL, the exec_pred P_1 is always the constant true (that is, no restriction applied during execution), and the gr_pred P_2 is either *false* (if no grant option) or *true* (if with grant option). We call AG a SQL *authorization* graph if every exec_pred is the constant predicate true and every gr_pred is one of {*true*, *false*}.

The authorization graph AG determines whether a subject s is allowed to perform a command. In particular, AG must have a chain to s that *justifies* performing the command. The general definition of justification appears in Section 3. In the special case of standard SQL, a chain justifies a grant command if all its edges have the grant option; a chain justifies an execute command if all its edges (except possibly the last) have the grant option. More formally:

Definition (for SQL grants): Let $C \equiv <G_1,\ldots,G_n>$ be a chain to s.

- C is *SQL-valid* if $\{gr\_pred_i \mid i = 1, \ldots,n-1\}$ are all *true*. An edge is SQL-valid if it is part of a SQL-valid chain.
- If C is SQL-valid, then C is said to *SQL-justfy* execute commands.
- If C is SQL-valid and gr_pred_n is true, then C is said to *SQL-justify* grant commands.

There are two purposes to these rather elaborate formalizations of a simple concept. First, validity becomes non-trivial when we include general limitation predicates, and we wish to introduce the terminology early. Second, chains (and hence edges) can become invalid what a grant is revoked. In this case, it is the responsibility of the system to automatically revoke all invalid edges.

SQL grants correspond naturally to SQL authorization graphs. That is, if every edge in AG came from a SQL grant, then the graph is an SQL authorization graph. Also, when revoke deletes edges from a SQL authorization graph, the result is still a SQL authorization graph. Section 3.7 will show that our general concepts of validity and justification reduce to the above, on SQL authorization graphs.

3. PREDICATE-LIMITED GRANTS

Section 2 introduced authorization graphs, and defined, for the special case of SQL (unrestricted) grants, how valid chains can justify commands from a subject. This section considers the general case. In particular, justification becomes command-specific – a chain can justify a command only if the command's state satisfies the appropriate limitation predicates. Definitions are given in Sections 3.1 through 3.5. Section 3.6 considers workarounds for an implementation difficulty that does not arise in standard SQL. Section 3.7 shows that we cleanly extend SQL semantics.

3.1 Command States

Each command has an associated *state*, which consists of:

- values for environment variables at the time the command was issued;
- the contents of the database at the time the command was issued; and
- values for the arguments of the command.

Example environment variables include $USER (the subject performing the command), $TIME (the time of day when the command was submitted), $LOCATION (from which the command was submitted), $AUTHENTICITY (the certainty that the requestor is authentic), $TRUSTEDPATH (whether the command arrived over a secure connection), and $GLOBALSTATUS (e.g., whether the system is in "emergency mode").

Example argument values are $NEW_TUPLE (for insert and modify commands), $OLD_TUPLE (for modify), and $GRANTEE (for grant commands).

Interesting portions of the database state include group and role memberships (is this worker on the night shift?), local status (is a particular patient in emergency mode?), and cardinality (are the tables underlying an aggregate view large enough to preserve anonymity?).

3.2 Limitation Predicates

A *limitation predicate* is a Boolean function without side effects. The inputs to this function are references to a command state – arguments, environmental variables, and database contents. If P is a limitation predicate and C is a command, we write *P(C)* to denote the truth value of P given inputs from the state of C.

We do not propose a particular syntax for predicates. For a DBMS, SQL-like expressions (with embedded environment variables and functions) seem appropriate.

Example. A predicate that evaluates to *true* if the time of the command is during normal working hours or if the subject works the night shift:

($TIME between <8am, 6pm>) or ($USER in NightShift)

Example. A predicate that evaluates to *false* if the grant command is between a particular pair of users:

not ($USER = Boris and $GRANTEE = Natasha)

Example. A predicate that restricts a user from inserting high-priced tuples into a table:

($NEW_TUPLE.Price < 100)

3.3 Motivating Examples

Suppose the creator of table *Items* issues the following grant G_1:

grant insert **on** Items **to** joe
executeif ($TIME **between** <8am, 6pm>)
grantif ($USER **in** Manager) **and** (**not** $GRANTEE = mary)

Let θ be the operation “insert on Items”. Subject *joe* is allowed to execute θ only between 8am and 6pm. Joe is allowed to grant θ only when he is in the *Manager* role, and the (potential) grantee is not *mary*.

In the above grant G_1, the table creator (who has unlimited authority on its operations) issued the grant, and so Joe’s privileges are limited exactly by the specified predicates. Now suppose that Joe issues the following grant G_2 while he is a manager:

grant insert **on** Items **to** amy
executeif $DAY = monday
grantif $TRUSTEDPATH

We first note that Joe was allowed to issue G_2, because he was a manager at the time of the grant, and the grantee is Amy, not Mary. Now suppose that after issuing G_2, Joe is removed from the *Manager* role. Although Joe is no longer able to issue grant commands, grant G_2 will not be invalidated – G_1's predicates are evaluated using the state of G_2, which is taken from *the time the command was issued*.

We next consider what privileges Amy has. Since Joe cannot give Amy fewer restrictions than he has, his restrictions must be added to those of G_2. Thus Amy's effective exec_pred is *($TIME between <8am, 6pm>) and ($DAY=monday),* and her effective gr_pred is *($USER in Manager) and (not $GRANTEE = sue) and ($TRUSTEDPATH).*

3.4 Semantics

As before, and throughout this section, let C =<G_1,…,G_n> denote a chain to subject s. Each G_i has predicates gr_pred_i and exec_pred_i.

Definition (general case for grants):

- C is *valid* if n=1 (that is, C consists of a single edge from the root).
- C is *valid* if for each i, the initial subchain <G_1,…G_{i-1}> justifies G_i.
- C *justifies a grant* G from s if C is valid and for each edge G_k in the chain, gr_pred_k(G) = *true*.

This definition has a subtle mutual recursion. The following theorem provides an alternative characterization.

Theorem 1: C is valid iff for each G_j, for all its predecessors k<j, $\text{gr\_pred}_k(G_j)$ = *true.*

Definition: An edge G is *valid* if it is justified by some chain. An authorization graph is *valid* if all its edges are valid.

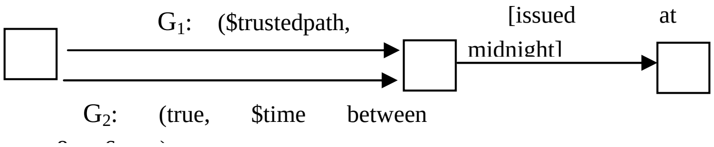
Theorem 2: In a valid authorization graph, all nodes with outgoing edges have at least one incoming valid chain.

Definition (general case for execution): C *justifies an execution command* E if C is valid and for each edge G_k in the chain, $\text{exec\_pred}_k(G)$ = *true*.

A grant G with an exec_pred of *false* is useless, regardless of the gr_pred. The grant authorization can be passed on to others, but the effective exec_pred along any chain involving G will be *false*, and thus the operation can never be executed.

3.5 An Example to Illustrate the Subtleties

Privileges are passed along valid chains, each of which effectively carries a pair of predicates, the conjunction of its gr_preds and the conjunction of its exec_preds. Even in a valid graph, some chains can be invalid, and one cannot then use them to justify commands. To illustrate this, consider the following authorization graph:



The graph depicts two grants from x to y: G_1 gives limited execution authorization but unlimited grantonward authorization; and G_2 gives unlimited execution authorization but limited grantonward authorization. G_3 is issued at midnight. To understand the privileges z holds, one must consider the *valid* chains.

Each chain with a single edge is valid, i.e., <G_1> and <G_2>. <G_1, G_3> is valid because <G_1> justifies G_3 (since gr_pred_1 is the constant *true*). <G_2, G_3> is not valid, because G_3 is not justified by <G_2>, because the state of G_3 has $time=midnight. Hence, commands by z can be justified only by <G_1, G_3>. An execution command by z can be justified only if it satisfies the exec_preds from <G_1, G_3>, i.e., arrived with *$trustedpath=true*. (If G_3 had instead been issued at 10am, then <G_2, G_3> would be valid, and the effective exec_pred for z would be *true*.)

3.6 Maintaining Validity

To evaluate incoming commands, one needs to know which chains are valid. Since grants' states do not change, one can evaluate each relevant predicate just once, at some time after the grant state is created. When w executes a grant command $G \equiv (w,x)$, each newly-created acyclic chain in AG involving G needs to be tested for validity. There are two cases:

- G is the last edge on the chain;
- G is in the middle of the chain.

In both cases, there are issues of algorithmic efficiency, which are outside our scope. The first case is somewhat easier, because the command state for G is currently available to be used in checking the chain's gr_preds. In the second case, there is a more basic complication: We cannot expect the entire system state from the last grant on the chain to have been retained (including the database of that time).

For example, consider the authorization graph of Section 3.5, and suppose subject x issues the following grant command (call it G_4):

grant θ **to** y **executeif** true **grantif** ($USER in Accountant)

In order to determine if the new chain $C' \equiv <G_4, G_3>$ is valid, we need to see if G_4 justifies G_3, i.e., to evaluate whether G_3 satisfies the predicate *$USER in Accountant*. To do so, we must have retained enough state of the earlier grant G_3 to know whether y was in *Accountant* at the time G3 was issued.

Consequently, both the semantics and pragmatics need to adjust. Pragmatically, an organizational policy could specify what portion of the system state that will be retained, and writers of Grant predicates would endeavor to use only that portion. The saved portion of the state may be extended over time, as the need for additional information is better understood.

Formally, if an edge predicate in C_{new} references state information that was not saved, then the system must determine how to assign a value to the predicate. We propose that the system treats unknown state information as a no-information SQL Null. If such grants are permitted, then the order in which Grant commands are received affects what information is available to evaluate predicate validity. To keep the system sound (i.e., not allowing grants that users would not want), we require that predicates be monotonic in information state – i.e., do not use "is Null".

3.7 Standard SQL as a Special Case

We now consider the connection between limitation predicates and standard SQL. An SQL grant without grant option gives arbitrary execute privilege and no grant privilege; thus it should be equivalent to

grant θ **to** s **executeif** true **grantif** false

An SQL grant with grant option gives arbitrary execute and grant privilege, and thus should be equivalent to

grant θ **to** s **executeif** true **grantif** true

This correspondence is confirmed in the following theorem:

Theorem 3: Consider a SQL authorization graph AG. Then:

- A grant or execute command, or an edge, or a graph, is valid iff it is SQL-valid.
- *AG* can be constructed by a sequence of SQL grants.
- The validity of a grant or execute command is independent of the command state. It depends only on the valid chains to the issuing subject (i.e., the subject's privileges).

If we use the conventions that an omitted executeif clause has a default value of *true*, that "with grant option" is an alternate syntax for grantif *true*, and an omitted grantif clause has a default value of *false*, then standard SQL syntax is incorporated seamlessly into ours.

4. LIMITED PERMISSIONS ON VIEWS

Databases have a rich theory for views; in this respect, they are more expressive than operating systems, information retrieval systems, and middleware. Several guidelines drove our extension of "limited privileges" theory to views. We wish again to satisfy the principles of Section 1, notably, to have recognizable chains of authority. We also preserve the usual amenities of view permissions: Grant/revoke from a view must behave as if the view were an ordinary table, and one can grant access to a view but not to the whole underlying table.

We present only an abbreviated treatment, largely through example, due to page limits. Specifically, we examine only the privileges that the creator has, and assume that only grants have limitation predicates (i.e., all exec_preds are *true*). These restrictions, and the dictatorial power of the view creator, will be relaxed in future work.

For each view, we define an authorization graph as if the view were an ordinary table, except that the creator does not get unlimited privileges. Let $V=Q(T_1,\ldots,T_m)$, and suppose for the moment that the exec_pred of each T_i is simply *true.* Then the semantics are: the view creator (and the initial node of the view's authorization graph) is initialized with a grant limitation that is the intersection of these predicates, (If a view creator were not subject to these limitations, a user with limited access to table T could escape the limitations by creating the view “Select * from T”.)

We now sketch several directions for extending the model of view privileges.

First, consider the view V defined by “Select A_1, A_2 from T”. In conventional SQL, a view creator may want to grant the privilege on the operation *select from view V* to users who do not have authority on the base table T. But suppose the creator suffers from a limitation predicate on T, and hence also on V. Who is empowered to make grants that loosen the limitations on the view? Thus far, nobody has this very useful right.

To cure this (and several other ills), in future work we will move away from treating a view as an object to be owned. Instead, it will be seen as derived data, for which certain permissions are justifiable. To start with, any possessor of a right on T can grant that right on V. (We are currently assuming the view definitions to be readable by all interested parties. Under this assumption, any subject with access to T could just define their own view, identical to V, and then grant the privilege.)

Next, consider a view over multiple tables, e.g., “Select A_6, A_7 from T_1 join T_2”. Oracle SQL specifies that the creator’s privileges on the view are the intersection of the creator’s privileges on the input tables. In [Ro00] we apply the same rule to non-creators. It extends easily to handle grants with limitation predicates on just execute – the creator’s limitations are the intersection of the limitations on all inputs. For the general case, a more complex graphical treatment is needed to capture that a privilege on a view requires a valid chain (appropriately generalized) on every one of the view’s inputs.

5. COMPARISON WITH PREVIOUS WORK

We compare our work with several recent, ambitious approaches. We consider only the part of each work that seems relevant to predicate-limited grants.

[Bert99] is the culmination of a series of papers that offer powerful alternatives to SQL. In [Bert99], two rather independent ways to lessen privileges are proposed. First, there is SQL-like cascading Revoke, without explicit predicates. Second, there are *explicit* negative authorizations, which temporarily inactivate base permissions (node privileges, not grant edges) that satisfy an explicit predicate p. (We can achieve the same effect by ANDing a term *(not p)* to the execution predicate for edges out of the root.) That model includes a large number of constructs, both for vendors to implement and for administrators to learn and use. Administrators must manage the additional privilege of imposing negative authorizations. The negative authorizations can be weak or strong, and there are axioms about overriding and about breaking ties. The model may be too complex to serve as the basis for a practical facility.

We believe that limitation predicates provide a simpler model that still meets the requirements identified in [Bert99]. Our model also improves on [Bert99] in two other areas – scoping of limitations, and views. Their negative authorizations are global, while our limitation predicates apply only along paths in the authorization graph. This scoping greatly reduces the chance of inadvertent or malicious denial of service. For views, [Bert99 section 2.3] adopts a very strong semantics for negative authorization – that absolutely no data visible in the view be accessible. Observing that implementation will be very costly, they then specify that there should be no limitations on views. By settling for less drastic controls, we are able to provide several useful capabilities (as described in Section 3).

Another important predecessor to our work is [Sand99], which proposes “prerequisites” (analogous to our limitation predicates) for onward privileges. The model limits only onward privileges, not execution privileges, and administrators must manage grants for the right to revoke each privilege. [Glad97] also has an effective notion of prerequisites, but has no direct support for granting privileges onward.

The Oracle 8i product supports “policy functions”, which allow administrator-supplied code to add conjuncts to a query’s Where clause. This mechanism is powerful, but difficult to understand. For example, it can be quite difficult to determine: “Based on the current permissions, who can access table T?”. There does not appear to be an analogous facility for gr_preds.

[Bert98] considers a special case of limitation predicates, namely those that specify time intervals when the privilege can be exercised. Although our predicate mechanism can handle such temporal predicates, we were not concerned with providing a special syntax and evauation mechanism, as done in [Bert98].

Finally, limitation predicates can capture much of the spirit of Originator Control (ORCON) restrictions. [McCo90] discusses Originator Control restrictions, such as “You can must get the originator’s prior permission before you ship a document outside some group G”. (Rather than controls on the products that a user produces, we assume commercial-style discretionary controls on granting and using privileges. We thus assume that authorized users do not simply make their own copy of the information, and ship it.)

To model ORCON, we assume that a user s passes the information to s2 by Granting s2 the right to read the original document (or a DBMS-controlled replicate located closer to s2). Within this style, limitation predicates are adequate. If s has no right to pass on the information, then the grant to s should have no grant option. If s has a right to pass information within a group FRIENDS but not outside, the grant to s carries the gr_pred that “grantee ε FRIENDS”. We conjecture that other ORAC policies in [McCo90] can be similarly approximated.

6. SUMMARY

This paper represents an initial theory that we believe deserves follow-up. The main contributions of the work are to state principles for a limitation model, and then to provide semantics that satisfy these principles. We also extended limitation semantics to permissions on views. (Previous work in non-database models naturally did not consider this issue.)

Our approach makes several advances over previous proposals.

- *Model Economy:* The model integrates Grant and Execute privileges, consistently. It cleanly extends SQL. An earlier (longer) version of this work showed that it was modular, cleanly separable from reactivation.
- *Easy administration:* The model naturally extends SQL concepts, and accepts all SQL grants. There is no need to administer a separate “Limit” privilege.
- *Limitations respect lines of authority:* Only a grantor or a responsible ancestor has the authority to limit or revoke a user’s privilege.
- *Flexibility in limitations:* Designers can give any degree of power to limitation predicates. For a pure security system (unconnected to a DBMS), one could have queries only over security information plus request parameters (e.g., time, trusted path). For a DBMS, one could allow any SQL predicate.
- *Views:* Limitations on underlying tables apply in a natural way to views.

Further work is necessary, of course. The top priority (beyond our resources) would be to implement these features, both as proof of concept and to gather users' reactions. Traditional research questions include:

- *Extend the theory* to support
  - a fuller treatment of limitations on views. A prerequisite for doing this well is to rethink the basic treatment of view ownership.
  - privilege inheritance via roles and groups.
  - a clean theory of reactivation and dynamic reevaluation of predicates.
- *Efficient implementation.*
- *Beyond access controls, allow limitation policies that modify execution of operations.*

The pragmatic questions are equally important.

- Would users see limitations as an important mechanism, and would they do the administrative work to impose them? How much generality is needed?
- How does this model compare with what is implemented in security policy managers for enterprises or for digital libraries [Glad97].
- What tools are needed to make it all usable?

7. REFERENCES

[Bert98] E. Bertino, C. Bettini, E. Ferrari, P. Samarati, "An access control model supporting periodicity constraints and temporal reasoning," *ACM Trans. Database Systems*, Vol. 23, No. 3, Sept. 1998, pp. 231 – 285.

[Bert99] E. Bertino, S. Jajodia, P. Samarati, "A Flexible Authorization Mechanism for Relational Data Management Systems," *ACM Trans. Information Systems*, Vol. 17, No. 2, April 1999, pp. 101-140.

[Cast95] S. Castano, M. Fugini, G. Martella, P. Samarati, *Database Security*, ACM Press/Addison Wesley, 1995.

[Glad97] H. Gladney, "Access Control for Large Collections," *ACM Trans. Information Systems*, Vol. 15, No. 2, April 1997, pp. 154-194.

[ISO99] ISO X3H2, *SQL 99 Standard*, section 4.35.

[McCo90] C. McCollum, J. Messing, L. Notargiacomo, "Beyond the Pale of MAC and DAC – Defining new forms of access control," *IEEE Symp. on Security and Privacy*, 1990, pp. 190-200.

[Ros00] A. Rosenthal, E. Sciore, "View Security as the Basis for Data Warehouse Security", CAiSE Workshop on Design and Management of Data Warehouses, Stockholm, 2000. Also available at http://www.mitre.org/resources/centers/it/staffpages/arnie/

[Sand99] R. Sandhu, V. Bhamidipati, Q. Munawer, "The ARBAC97 Model for Role-Based Administration of Roles," *ACM Trans. Information and System Security,* Vol. 2, No. 1, Feb. 1999, p 105-135.