Extending SQL's Grant and Revoke Operations, to Limit and Reactivate Privileges[1]

Arnon Rosenthal
The MITRE Corporation
arnie@mitre.org

Edward Sciore
Boston College and MITRE
sciore@bc.edu

Abstract

We propose two extensions to the SQL grant/revoke security model. In SQL, grants are unconditional, so the grantor must simply trust the recipient's discretion. We allow a grantor to impose limitations on how the received privilege may be used. Second, we provide a new means of selectively reactivating permissions that have been revoked. 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.

1 Introduction

The SQL security model has had few extensions in the past 20 years, except for the recent addition of role-based access controls. We propose some extensions that give important extra power at relatively low cost, in the hope of attracting vendors and users. Our goal is for these extensions to be simple for current SQL administrators to understand, simple to implement, and separable (so that one can be implemented without the other).

This paper addresses the following specific problems:

- *The need for a privilege-limitation mechanism.* When one delegates the power to grant privileges to others, especially in a distributed system, one needs to provide guidance and limits on how that power may be exercised. At the same time, the limitation mechanism should respect lines of authority, and introduce minimal new complexity.
- *The need for flexible privilege revocation and optional reactivation.* There are several possible semantics, differing in how orphans are defined and treated.
- *The need to include views as part of the model.* They should behave much like ordinary tables.

1.1 Requirements for Technology Transfer

Database security research has recently had distressingly little influence on DBMS vendors. To change this situation, we believe it is critical to have a high ratio of (value added) /complexity. Although the published research contains many interesting requirements and clever constructs, it is now time to craft models that are more suited to technology transfer.

[1] IFIP Working Conference on Database Security, 2000, Amsterdam.

Two key factors are modularity and backwards compatibility. Modularity allows vendors to deliver extensions gradually, with each increment providing useful functionality; the vendor can gain revenue and see if the market is accepting the advances. Users can absorb the new features gradually, and administration policies can be localized to each module. Backwards compatibility implies that the new features be a small delta from those they already have. In particular, the limitation and reactivation features should respect SQL's semantics of cascading deletions, at least as one of the supported options.

There are security models and products that offer greater scope and flexibility than ours. Our contributions are modularity, simplicity of each module, and (since we aim at databases) SQL compatibility.

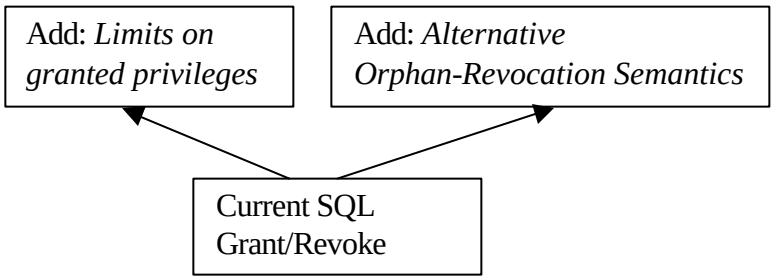


Figure 1: The extensions proposed in this paper

1.2 Paper Contents

This paper separates two capabilities that previous models tied together – limits on privileges granted, and the ability to *temporarily* revoke a privilege. Our envisioned implementation, and our presentation, splits these capabilities roughly into two modules (see Figure 1). The first module, the *Active grants module*, extends SQL grant/revoke semantics by allowing grants to have *limitation predicates* that restrict how the permission may be used. Its responsibility is to maintain an authorization graph: it adds grants to the graph if the grantor has the necessary authority; and when that authority is lost, it detects orphans and revokes them.

The second module, the *Inactive grants module*, provides several modes for invalidating and reactivating orphans' privileges. These include standard SQL (complete deletion), automated reactivation, or fine-grained control of the choice between the previous two. This module, which is absent from current DBMSs, intelligently maintains the set of orphaned permissions that are candidates for reactivation. It monitors each Grant operation, and determines if the operation causes any orphaned grants to be reactivated.

Section 2 examines the active grants module, and Section 3 examines the inactive grants module. Section 4 extends the theory to views. Section 5 surveys previous work, both recent efforts and IBM's classic System R research.

2 Granting with Limitation Predicates

In SQL, a user who has a privilege is able to use it in any circumstance. Attaching a limitation predicate to each grant can refine this "all or nothing" mechanism. For example, a limitation predicate may restrict usage to certain days or hours, to members or nonmembers of certain groups, to users above a certain rank, to tables above a threshold size, or to requests that are received along a tamper-proof path (or which exceed a threshold certainty that they are authentic).

2.1 Principles for a Limitation Model

Our privilege limitation model is motivated by three 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.
- Limitation predicates should be independent of the treatment of reactivation, in both semantics and implementation.

The first principle implies that "revoking a privilege" and "imposing additional limits on an existing grant" are facets of the same concept. For example, revoking a privilege should be equivalent to imposing a limitation predicate of *false*. 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. In addition, a user should be able to modify not only the grants he made, but also any grant for which he is responsible.[2] For example, the owner 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.[3]

The chains of authority also imply that a subject can limit *only* those grants for which it is responsible. If user y has received grantable privilege θ 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.

---

[2] We say that user S_1 is *responsible* for a grant of privilege θ to user S_2, if all authorization chains from the owner of θ to S_2 pass through S_1. That is, if S_1 revoked all of her grants of θ, then S_2 would lose all privileges to θ.

[3] For further power, beyond our model, a responsible subject might delegate the right to impose limitations, e.g., for the DBA to delegate such rights to emergency response officers. Such a mechanism might be used for broad policies, but would not be the concern of most object owners.

The third principle allows users to learn, and vendors to implement, just one of the extensions.

2.2 Subjects, Privileges, and Grants

A *subject* is a user, group, or role; the SQL standard uses the term *authorization identifier*. We treat each subject as atomic; inheritance among subjects (e.g., from a group to its members) appears straightforward but is left for future research.

A *base privilege* is the right to perform some operation on some object. Example base privileges include selecting from (i.e. reading) a particular table, or executing a stored procedure. For each base privilege there is a corresponding *onward privilege*. An onward privilege is the right to grant either the base privilege or the onward privilege to a grantee. For example, *select(T)* is the base privilege of being able to select from table T; the corresponding onward privilege is *grantselect(T)*. We denote an arbitrary privilege with the Greek letter θ. Subscripts distinguish a base privilege θ_b from the corresponding onward privilege θ_g; together they are called a *privilege pair*.

Privileges are associated with subjects through one or more *grant commands*. Our model extends the standard SQL model in two ways:

- An onward privilege can be granted without the corresponding base privilege.
- A grant can have one or two *limitation predicates* specified with it.

Before we plunge into formal definitions, these bullets require some motivation.

Allowing a subject to receive an onward privilege without the corresponding base privilege is desirable for implementing some cases of separation of duty. For example, a software installer might be given an onward privilege with the limitation that the installer can grant only the corresponding base privilege, and only to subjects in Accounting (to which the installer does not belong). Organizations that wish to conform to standard SQL can enforce the protocol that each onward grant be accompanied by the corresponding base grant, and similarly with revocation. In this way, predicate pairs will always be maintained.

Associating predicates with grants allows the grantor to limit subsequent use of the power he grants. Two sorts of limitations can be imposed on a grant – limits on exercising the base privilege, and (for onward privileges) limits on granting the privilege onward. In the example of the above paragraph, the installer would be given the onward privilege with an onward predicate of *false* (i.e., no onward grants allowed), and a base predicate of *$GRANTEE is in Accounting*. These predicates are cumulative, in the sense that the limitations received by the grantor propagate to the grantee. That is, each grant G has an associated chain (or chains) of prior onward grants that justify the granting of G, and the grantee of G can exercise the privilege obtained along a chain only if all predicates in the chain are satisfied. One may also revisit justifications for existing grants, in light of changes to the database state (see Section 2.3).

For example, the owner of table T may grant privilege θ_g on T to a project leader, with the limitation that the privilege can be used only along a tamper-proof path. The project leader might then grant the privilege with the limitation that it can be passed onward only to members of group Engineers. Those recipients of θ_g are then subject to both limitations.

The accumulation of limitations along predicate chains allows each grantor to worry only about his own concerns – they need not repeat restrictions imposed by their ancestors. It also makes it possible to rapidly add an extra limitation (e.g., *$GRANTEE is in Employee)* when circumstances change. If an object's owner adds this limitation, it will apply immediately to all descendent permissions.

Section 2.2.1 describes the form of limitation predicates and gives further examples. Section 2.2.2 formally defines the types and semantics of grants in our model. Section 2.2.3 further discusses the intuition behind the formal definitions.

2.2.1 Limitation Predicates

A *predicate* is a Boolean function without side effects. The inputs to this function can be properties of the actual grant request (such as the grantor, grantee, or time of grant), or information in the current database state (including the security state, such as previous grants and group membership). Like a stored procedure, a limitation predicate should execute with the grantor's privileges.

Predicates apply to a particular request, using the environment and database state at the time of request submission (or occasionally, the time grants are revalidated). We do not propose a particular syntax for predicates. For a DBMS, however, SQL-like expressions (with embedded environment variables and functions) seem to offer a roughly appropriate syntax.

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

($TIME between 8am and 6pm) or ($USER in NightShift)

A predicate limiting future onward grants may reference the input variable $GRANTEE, which denotes the subject to which a future grant might be directed. In such grants, $GRANTOR is a synonym for the $USER issuing the grant. For example:

Example. A predicate that evaluates to false whenever there is a grant between a particular pair of users:

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

Other likely environment predicates include $LOCATION (from which the request was submitted), and $AUTHENTICITY_LEVEL which describes one's certainty that the requestor was authentic and the request not tampered with. Interesting portions of the database state include group and role memberships, global status (is the system in "emergency mode"), and local status (is the particular patient in emergency mode?).

The default time for binding values to predicate arguments is the time the request is submitted (for θ_g, the time of grantonward). Such arguments are *fixed*; this same value is used if the grant is revalidated later. Thus in the first example above, a grant by a day worker remains valid when the clock passes 6pm, and midnight grants by a night worker are not invalidated when the worker is transferred off the night shift.

But sometimes a Grant should be revalidated in terms of current database state (e.g., when "Status = emergency" no longer holds). Support for dynamic assignment requires extensions to the syntax and (more seriously, as discussed in Section 2.3) the semantics. It might reasonably be omitted from a first implementation of limitation predicates.

2.2.2 The Semantics of Grant Commands

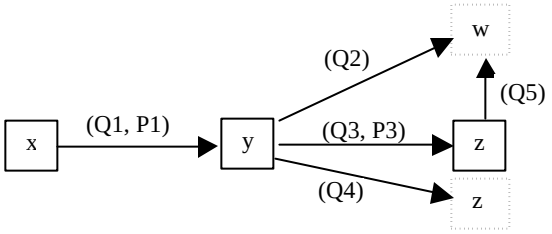
Formally, a *grant* of base predicate θ_b is the 4-tuple (grantor, grantee, θ_b, b-pred). A grant of onward privilege θ_g is the 5-tuple (grantor, grantee, θ_g, b-pred, g-pred), where b-pred and g-pred are limitation predicates. In order to present the semantics of grant commands, we need the following definitions.

An *authorization graph* is a graph that represents the set of grants for *one* privilege-pair (θ_b, θ_g). For each subject, there are two nodes, a *base* node (showed dotted) and an *onward* node. For each grant of either θ_b or θ_g there is a directed edge, from the grantor's onward node to the grantee's base or onward node, respectively. The edge is labeled with the specified predicates of the grant. The graph has a distinguished root node, which is an onward node, corresponding to the owner of the privilege-pair.

For example, let x be the owner of privilege-pair (θ_b, θ_g), and consider the following grants:

(x, y, θ_g, Q1, P1);
(y, w, θ_b, Q2);
(y, z, θ_g, Q3, P3);
(y, z, θ_b, Q4)
(z, w, θ_b, Q5)

The authorization graph for these grants (omitting nodes with no incident edges) is:



Note that only an onward node can have outward edges. Consider any directed path from the root node. Its edges (except the last one) will therefore denote a series of grants of the onward privilege, and its final edge will denote the grant of either the base privilege or that onward privilege. This observation motivates the following definition:

Let n be a node in an authorization graph. An *authorization chain* for n is a directed path from the distinguished root node to n. Each authorization chain has one or two *chain predicates,* defined as follows:

- The b-pred chain predicate is the conjunction of all b-pred predicates in the chain;
- The g-pred chain predicate is the conjunction of all g-pred predicates in the chain, and is defined only if n is an onward node;
- If the chain contains the single root node, then its chain predicates are both *true*.

For example in the above authorization graph, there are two chains for subject z: a chain for z's onward node, whose b-pred chain predicate is $Q1 \wedge Q3$, and whose g-pred chain predicate is $P1 \wedge P3$; and a chain for z's base node, whose b-pred chain predicate is $Q1 \wedge Q4$. Similarly, there are two chains for w's base node: one with b-pred chain predicate $Q1 \wedge Q2$, and the other with b-pred chain predicate $Q1 \wedge Q3 \wedge Q5$.

We can now define formally the semantics of a grant command. Consider an authorization graph for privilege-pair (θ_b, θ_g). A request by subject x to exercise θ_b is allowed if there is some authorization chain for x's base node, whose b-pred chain predicate evaluates to *true* with predicate arguments from that request. Subject x is allowed to exercise the onward privilege θ_g (i.e., perform a grant command) if there is some authorization chain for x's onward node, whose g-pred chain predicate evaluates to *true* from that request.

2.2.3 Discussion

The authorization chain is the basic unit of evaluation. A b-pred chain predicate should be tested each time the underlying base privilege is used, using the state at the time of the request. A g-pred chain predicate should be tested in two circumstances: First, during the execution of an onward grant, in order to test that the grantor has requisite authority; and second, if it includes dynamic variables, when one wishes to reevaluate whether a grant is still justified (as discussed in the next section).

An alternative to explicit b-predicates would be to define a view that contains the b-predicate as a Where clause conjunct. One could then pass permissions on the view. We do not recommend this approach. It seems likely to cause a proliferation of views with overlapping predicates. Furthermore, when a request receives an empty response, users are not informed whether the difficulty is due to access limitations or to an ordinary selection predicate.

Efficient implementation remains an open research problem. One should beware of having predicates whose inputs require database access, particularly on base predicates (which require evaluation each time the privilege is used). Instead, one would want to use data from the usage request, or database data that could be cached. For further efficiency, if many chain predicate terms reference static values from prior grants, one might want to pre-evaluate terms using those values.

For greater control, a system might allow the owner's privileges to be initialized to some value other than *true*. For example, one could treat the owner's privileges as delegated from the DBA, so the DBA can restrict the owner's privileges. For example, when preparing to move toward distributed processing, one might impose the default limitation that *grantModify* exclude grantees from coalition partners. Such control can be achieved by making the DBA-node be the distinguished node in the authorization graph.

Grants can be expressed in an extension of SQL.

Example. The following grant allows joe to read from the SalaryInfo table during working hours:

grant select **on** SalaryInfo **to** joe **bpred** ($TIME between 8am and 6pm)

Example. The following grant allows joe to perform grant commands on SalaryInfo, but only to accountants. Moreover, any base privilege resulting from such grants must be performed during working hours:

grant onward select **on** SalaryInfo **to** joe
gpred ($GRANTEE in Accountant)
bpred ($TIME between 8am and 6pm)

Example. The SQL predicate-pair-preserving semantics can be obtained by using the "grant option" syntax. The following grant has the same effect as the previous two:

grant select **on** SalaryInfo **to** joe
gpred ($GRANTEE in Accountant) **with grant option**
bpred ($TIME between 8am and 6pm)

2.3 Dynamics of Revocation Due to Predicate Evaluation

SQL revoke semantics requires that any Grant G that does not currently have a valid authorization chain is automatically revoked; G is called an *orphan*. In our model, an authorization chain can become invalid in two ways. First, as above, a predecessor of G may be revoked. Second, due to a new value of a dynamically-bound argument, the chain predicate for G might become false. If every limitation predicate in the chain uses fixed arguments, then the value of the chain predicate does not change, and the second mode need not be considered. This section considers this second, dynamic case.

SQL semantics call for the authorization graphs to be maintained, up to date, whenever grants are revoked or granted. So all active grants must be checked at least at those times. However, the inputs of a limitation predicate can change with time (e.g., references to database values, group memberships, and other grants). The model must therefore specify when predicates need to be reevaluated: Never? Whenever a grant is revoked? Whenever a grant is made? Whenever the database changes?

However the decision is made, it will raise difficult issues of semantics. Will a temporary state change cause revocations to cascade? Will the behavior be predictable? Efficient? We propose some semantic alternatives below.

1. *Never reevaluate.* (Equivalently, support only "grant-submission-time" arguments for g-preds). Although this excludes some useful limitation predicates, it is still more general than current systems, and includes some (but not all) predicates mentioned in [Bert99, Sand99, Glad97]. For immediate implementation, it seems the most desirable.

2. *Have each grant specify exactly when its predicates should be re-evaluated.* This approach has considerable flexibility, but requires extra time from skilled administrators. Even so, the goal of predicable behavior is likely to prove elusive. Revalidation of one chain can revoke grants and hence break other chains. This would affect (and perhaps surprise) many users beyond the one who performed the triggering request.

3. *Reevaluate continuously*. All predicates are (conceptually) reevaluated after each database change. If the authorization chain that justified an onward grant has become false (and no other chains are true), that grant is revoked. Further grants for which this grant is responsible are also revoked. This approach has simple semantics, but may be infeasible to implement. A reasonable rigorous compromise is:

4. *Designate certain events as important,* and reevaluate only when such an event occurs. Candidates for "importance" include administrator request for reevaluation, revocation of all (or part) of a privilege, Granting additional permissions, or a timer that goes off when a time-limited predicate has expired. With this approach, we know that when an important operation was executed, it had necessary permissions in its current environment. Unimportant operations were justified as of the (difficult to predict) last time their predicates were reevaluated.

We emphasize that the difficulties here are not due to our use of an exotic limitation model. Any model that includes groups whose memberships change with time (e.g., SQL99, or ARBAC97 [Sand99]), or includes temporal predicates (e.g., [Bert98]), faces these difficulties. Current solutions include awkward restrictions; for example, DB2 forbids onward granting of privileges received via group membership. A better understanding of the dynamics could make membership revocation more consistent with Revoke, in all models. This seems a good topic for future research.

Finally, we point out that it makes sense for a system to provide tools to help the administrator, especially to avoid surprises. Crucial ones include

- What if? What orphans are created if I revoke a grant? For those orphans that I do not want revoked, help me determine appropriate additional grants that will provide the necessary authorization chains.
- Warning: The system has detected that the value of a predicate has changed. Show me the orphans that will be created, and give me time to make appropriate adjustments, if necessary.
- What gets reactivated if I execute a grant? Help me control this.

- Give me a history of all previous grants, to search or browse, so that I can delete grants that are no longer of interest (a sunset clause), and edit and re-authorize grants as desired.

3 Reactivation of Orphaned Grants

When a grant is orphaned, current DBMSs completely delete it. However, sometimes one wishes to consider a revocation as temporary. We deal with temporary loss of authority by the grantor. (Section 5 will compare this with negative authorizations). In this section, we propose a reactivation capability for orphaned grants. This capability can be implemented as a module, applicable to either standard SQL or to SQL with limitation predicates.

Section 3.1 examines the issues, assuming a standard SQL system with no limitation predicates. Section 3.2 considers the interaction with limitation predicates.

3.1 Revocation without Limitation Predicates

In this section we assume that the limitation predicates in all grants are *true*. We examine three semantic approaches – pure deletion (i.e., the null case, where one provides no reactivation facility), pure reactivation, and their synthesis, controllable reactivation. As elsewhere in the paper, we focus on semantics rather than implementation.

Approach 1. *Orphaned grants are permanently deleted.*

SQL DBMSs follow these semantics, and provide no reactivation mechanism. Hence, orphans need not be maintained.

Approach 2. *Pure reactivation.*

When a grant becomes orphaned, it is revoked and placed in a set called *Inactive*. Let G be an *Inactive* grant of θ from S_1 to S_2. If a grant is issued that restores S_1's privilege θ_g, then G is reactivated – that is, a Grant command is automatically processed for G, and G is deleted from *Inactive*. If θ is an onward privilege, then, reactivation can cascade.

Note that in this approach, the set *Inactive* is always disjoint from the set of active grants (which we call *Active*). *Inactive* grants cannot participate in authorization chains, or justify onward grants.

The semantics above require that reactivations always occur, automatically. This is not a serious candidate for replacing a DBMS's current revocation semantics – SQL's semantics are sometimes preferable, and more important, we cannot abandon the installed base. We are thus led to consider ways to offer both options. We describe below an approach for achieving this, at the granularity of individual grants.

Approach 3. *Controlled reactivation.*

Each operation says how it wants to treat reactivation. A Revoke operation specifies whether to permanently delete orphans, or to save them in Inactive. A Grant operation (for an onward privilege) specifies whether reactivation of *Inactive* grants should occur, and whether cascading should occur.

If all revokes say "delete permanently", we get approach 1. If all grants and all revokes say "yes to reactivation" we get approach 2. When a revocation stems from a change to a dynamic argument of a limitation predicate, it seems appropriate to reactivate automatically. (This gives behavior equivalent to never having detected the need for revocation). The tradeoffs among the approaches are unclear, and require real world experience. In particular, will users understand the effects or want the burden. Can tools and defaults sufficiently mitigate the problem?

3.2 Revocation with Limitation Predicates

Section 3.1 assumes that "revoke" and "reactivate" applies to an *entire* grant. If the Active grants module supports limitation predicates, then modification of a limitation predicate can also trigger orphaning and subsequent reactivation.

In the following, we consider only base privileges, which have a single limitation predicate. Onward privileges, with their two predicates, are treated analogously.

Consider an existing grant $G = (S_1, S_2, \theta, p)$, and suppose it is modified so that its local predicate is replaced by q. We work in two steps. First we grant $G' = (S_1, S_2, \theta, q)$; then we revoke G. To maintain the invariant that *Active* and *Inactive* are disjoint, we place only $(S_1, S_2, \theta, p\text{-}q)$ into *Inactive*. Also, when revokes are placed into *Inactive*, we take the disjunction of all grants of the same privilege, between the same subjects. So *Inactive* will have only one grant from S_1 to S_2 for θ.

Next, suppose a new grant $G' = (S_1, S_2, \theta, p')$ is issued, and the corresponding grant in *Inactive* is (S_1, S_2, θ, q). If p' overlaps q, then we can split the *Inactive* grant into two grants with local predicates q-p' and $q \cap p'$. The first portion is unaffected, and remains in *Inactive*; the second portion now overlaps the active grant, and is removed.

Finally, we consider reactivation. Suppose the active module has processed a grant that again gives θ_g to S_1. We now act as if S_1 again granted θ to S_2. As a conservative choice, we bind limitation predicate arguments using values fixed as of *now*, and reevaluate the g-pred chain predicate. It it succeeds, then the grant (S_1, S_2, θ, p) will be granted, and deleted from *Inactive*.

4 Interactions with Views

Unlike operating systems, information retrieval, and middleware, databases have a rich theory for derived objects, namely views. This section applies our limitation-predicate semantics to views. Our goal is to preserve two elegant properties of relational views. First, from the grantees' viewpoint, it is irrelevant whether the table they received is a base table or a view. Thus, grant/revoke from a view must behave as if the view were an

ordinary table. Second, one should be able to grant privileges on the view without granting access to the entire underlying tables.

We begin with two examples. First, suppose a subject S has the privilege *grantSelect(T)* on a table T, subject to a limitation predicate. If S defines a view V over T, the limitation predicate must also apply to *grantSelect(V)* – otherwise, S could escape the limitation on T by defining the trivial view "select * from T". Hence for views, initial predicates on the root node of the authorization graph for *grantSelect(V)* are initialized to the owner's limitations.

Second, consider the view V defined by "Select A_1, A_2 from T". In conventional SQL, a view owner may want to grant the privilege *select(V)* to users who do not have authority on the base table T. But suppose the owner 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?

The problem is solved by exploiting lines of authority. We treat a view not as an object to be owned, but as a set of derived data, for which certain permissions are justifiable. The solution below is for the general case, where a view query can have multiple input tables. Our approach generalizes the treatment in Oracle SQL, where the owner's privileges on a view are the intersection of the owner's privileges on the input tables.[4]

Let θ denote the privilege for an operation op on a view V. Our solution is:

- The owner's limitation predicates for θ_b and θ_g are the conjunctions (respectively) of the b-pred and g-pred chain predicates for the view's input tables T_i.
- *Any user* may grant additional permissions on a view, as justified by their permissions on all the underlying tables. Consequently, a user who does not suffer from the definer's limitations can authorize wider use.

While the treatment is not complete, we contend that the basic difficulty lies with SQL, which effectively requires that a view owner be trusted by owners of all the underlying tables. This trust may be unachievable for views that span organizations. Fortunately, if one trusts the DBMS, one can sometimes collaborate without a universally trusted access control administrator. Suppose one can define a view V that "sanitizes" information from multiple inputs (e.g., by computing statewide totals of Patient statistics). In [Ros00], constructs are proposed by which possessor of θ_g can grant to another subject the right to use θ_b to compute V, and the right to grantonward that privilege.

5 Comparison with Previous Work

We compare with several recent, ambitious approaches. We then examine how the original System R looks, through our lens of reactivation. Each of the references below represents an extensive study, with ideas on many subjects, with different goals. We

[4] As stated, this policy needs further research for updates to multi-table views. For example, update of a view that joins across a foreign key may require update access to one table, and read access to the parent.

consider only the part of their work that seems relevant to our two main issues, limitations and reactivation.

Our research began with our attempt to obtain a simpler model that met the requirements identified in [Bert99], which is the culmination of a series of papers that offer powerful alternatives to SQL. We agree with most of their analyses, but think that the foundation is too complex. A model that begins complex in theory will become unworkably complex in practice.

[Bert99] proposes two rather independent ways to lessen privileges. First, there is SQL-like cascading Revoke, with neither explicit predicates nor reactivation. Second, there are *explicit* negative authorizations, which temporarily inactivate grants satisfying an explicit predicate. These explicit predicates give great flexibility in choosing what to deactivate – which can be useful, or dangerous.[5] We next address our concerns with the model.

First, the model included a large number of constructs, for vendors to implement and administrators to learn and use. Deactivation requires explicit action (while our more limited form was automatic, when supporting chains are lost). The negative authorizations can be weak or strong, and there are axioms about overriding and about breaking ties. (We wonder about the ease of administering strengths, and whether two levels will be adequate). Most significant, there appeared no good way to get both reactivation and cascading; currently, deactivation does not cascade, so orphaned grants remain in force.

We are able to improve 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 (Section 4).

Another important predecessor is [Sand99], which proposes "prerequisites" (analogous to our limitation predicates) for onward privileges, with implicit reactivation. One is guaranteed only that a privilege was grantable at the time the grant was made, i.e., all predicate arguments are taken as fixed. Revocation does not cascade to ordinary privileges whose grantor lost rights. Also, because the model is not coupled to a query environment, limitation predicates could reference only security administrative information (particularly grants of role membership). Finally, administrators must manage grants for the right to revoke each privilege.

[5] We hope eventually to extend our work by adding a controlled form of this explicit deactivation. One possible approach would be scripts that impose additional limitations and yet remember the old form so that it can be rolled back.

[Glad97] also has an effective notion of prerequisites. However, unlike database models, there is no direct support for granting privileges onward – instead, the model makes extensive use of roles and groups. In the absence of grantonward privileges, privileges do not become orphaned, and Revoke becomes trivial.

The Oracle 8i product supports "policy functions", which allow administrator-supplied code to add conjuncts to a query's Where clause. This mechanism is quite powerful and lets one implement many useful restrictions. But it is reminiscent of self-modifying code in a less favorable way – the difficulty of understanding consequences of using it. For example, it can be quite difficult to determine: "Based on the current permissions, who can access table T?". If anything, it magnifies the disadvantages of using views for b-predicates as discussed in Section 2.2.3.

[Bert98] considers a special case of limitaiton 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].

[Grif76, Fagin79] present and refine the grant/revoke semantics of IBM's original System R. Their key axiom was that every usable privilege needs an authorization chain (as in SQL), but in addition the timestamps must be in increasing order. Since a Grant cannot ever be supported by a later grant, reactivation is not an option. [Fagin79] proved a very elegant result for his semantics – the behavior is as if the revoked permissions had never been granted. The current authorization chains determine the user privileges currently justified by a grant/revoke history.[6]

For the SQL standard, the current permissions are defined procedurally, by explicitly deleting orphans from the Active set. Cascading Revoke is thus semantically necessary. (System R's implementation is no simpler, since they too remove orphaned grants, for efficiency.). Furthermore, administrators prefer to look forward based just on today's privileges, so they do not directly benefit from System R's elegant interpretation of histories. Finally, SQL's semantics are amenable to reactivation by later Grants.

Finally, we consider the use of roles, which can give significant help in organizing privileges and reducing explicit grants. For example, it is simpler to grant a privilege directly to a group, rather than granting to a user with a limitation predicate that the user be in the group. However, we suspect that one still will need a more primitive grant mechanism, for imposing explicit limitations on what can be passed onward from a role, and who can be admitted to membership.

6 Summary

This paper represents an initial theory that we believe deserves follow-up. The main contributions of the work are to:

[6] In fact, the privileges are the same regardless of whether one deletes orphaned Grants. However, deletion was done after a revoke in the process of determining the revised privileges.

- Modularize the issues of limitation versus reactivation, so as to allow separate progress (both theoretical and implementation) on each.
- State principles for a limitation model, and provide one set of semantics that satisfy these principles.
- Examine approaches to reactivation semantics.
- Extend the 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.

- *Modularity:* From the base of a SQL DBMS, one can add either limitation or reactivation or both. This lets a vendor improve functionality with a smaller investment.
- *Economy:* Limitation is a generalization of revoke; we do not need separate semantics for negative authorizations. The model has relatively few constructs, and administrators have relatively little metadata to supply.
- *Limitations respect lines of authority:* Only the grantor (or a responsible ancestor, or perhaps an explicit delegatee) can impose a limitation, or revoke a 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. A grantor who is not subject to the limitations can grant further privileges.

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. Second, it would be desirable to allow limitations that modify execution of operations, not just whether they can execute. (This can be achieved by creating stored procedures and views, but such approaches seem clumsy and opaque). Next, the theory should be extended to models where privileges are inherited through role and group membership. Finally, there are the algorithmic challenges of efficient implementation.

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.

[Date94] C. Date with H. Darwen, *A Guide to the SQL Standard, third edition* (SQL2), Addison Wesley, 1994.

[Fagin79] R. Fagin, Ón an authorization mechanism", *ACM Trans. Database Systems,* Vol. 3, No. 3, 1979, pp. 310-375.

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

[Grif76] P. Griffiths, B. Wade, "An authorization mechanism for a relational database system", *ACM Trans. Database Systems,* Vol. 1, No. 3, 1976, pp. 242-255.

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

[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.