An Authorization Mechanism for a Relational Database System

PATRICIA P. GRIFFITHS AND BRADFORD W. WADE
IBM Research Laboratory

A multiuser database system must selectively permit users to share data, while retaining the ability to restrict data access. There must be a mechanism to provide protection and security, permitting information to be accessed only by properly authorized users. Further, when tables or restricted views of tables are created and destroyed dynamically, the granting, authentication, and revocation of authorization to use them must also be dynamic. Each of these issues and their solutions in the context of the relational database management system System R are discussed.

When a database user creates a table, he is fully and solely authorized to perform upon it actions such as read, insert, update, and delete. He may explicitly grant to any other user any or all of his privileges on the table. In addition he may specify that that user is authorized to further grant these privileges to still other users. The result is a directed graph of granted privileges originating from the table creator.

At some later time a user A may revoke some or all of the privileges which he previously granted to another user B. This action usually revokes the entire subgraph of the grants originating from A's grant to B. It may be, however, that B will still possess the revoked privileges by means of a grant from another user C, and therefore some or all of B's grants should not be revoked. This problem is discussed in detail, and an algorithm for detecting exactly which of B's grants should be revoked is presented.

Key Words and Phrases: database systems, protection in databases, privacy, security, access control, authorization, data dependent authorization, revocation of authorization
CR Categories: 4.30, 4.33, 4.35

INTRODUCTION

A multiuser database system must permit users to selectively share data while retaining the ability to restrict data access. There must be a mechanism to provide protection and security, permitting information to be accessed only by properly authorized users. Further, when tables or restricted views of tables are created and destroyed dynamically, the granting, checking, and revocation of authorization to use them must also be dynamic.

In current database management systems the ability to grant authorization to perform actions on objects resides with a central "database administrator" or with

Copyright © 1976, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.
A version of this paper was presented at the ACM SIGMOD International Conference on Management of Data, Washington, D.C., June 2–4, 1976.
Authors' address: IBM Research Division, 5600 Cottle Road, San Jose, CA 95193.

the creator of the object. Many of the systems rely on password schemes, which are vulnerable to guessing. In addition many of them do not permit data dependent access control. In this paper we address the problems of dynamically authorizing data independent and data dependent operations and of revoking such authorization, in an environment in which more than one user may grant privileges on the same object.

We assume that the reader is familiar with the relational database approach as described by Codd [5–9], Date [10], and Boyce and Chamberlin [2, 3]. Although we will discuss the issues of authorization and their solutions in the context of System R [1], we believe that many of our solutions are applicable to problems common to any multiuser database system in which authorization is dynamically granted, checked, and revoked. The work described here is an extension of the ideas presented in [4].

The basic objects in the System R database are relations, which are sets of n-tuples. Each n-tuple in a relation has n columns; every column is named. An example of a relation is

EMPLOYEE (NAME, SALARY, MANAGER, DEPARTMENT)

The EMPLOYEE relation has a tuple (row) for each employee, giving his name, salary, manager, and department.

System R permits the creation and manipulation of relations through the facilities of the high level relational language SEQUEL [3]. SEQUEL is a nonprocedural, set oriented data definition, manipulation, and control language.

There are two types of relations: *base relations*, which are physically stored, and *views* [4]. A view is a virtual relation which is a dynamic window on the database. In response to a query the tuples of a view are materialized from the base relation(s) on which it is defined. An update to a view is performed (if possible) by updating the underlying base relation.

Any SEQUEL query whose result is a relation may be used to define a view. Therefore, a view may be:

- a row and column subset of a relation. For example, to define a view consisting of the names and salaries of the employees who work in the toy department, one can issue the SEQUEL statement

```
DEFINE VIEW  VEMP AS:
      SELECT  NAME, SALARY
      FROM    EMPLOYEE
      WHERE   DEPARTMENT = 'TOY'
```

- a summary of the information in a relation. For example, to define a view consisting of the average salary of each department, one can issue the SEQUEL statement

```
DEFINE VIEW AVGSAL AS:
      SELECT  AVG(SALARY)
      FROM    EMPLOYEE
      GROUP BY DEPARTMENT
```

- a join [5] of the information in two or more relations. For example, if the DEPT relation contains the number of the floor on which the department is located, one may define a view of employee names together with the floors on which they work:

```
DEFINE VIEW LOCEMP AS:
      SELECT  NAME, FLOOR
      FROM    EMPLOYEE, DEPT
      WHERE   EMPLOYEE.DEPARTMENT = DEPT.DEPARTMENT
```

It is to be emphasized that views are dynamic windows and not static copies. As the information stored in a base relation changes, the information visible through views defined on that relation changes with it.

The view mechanism is our means of granting access to row and column subsets, granting "statistical" access, and granting access to other transformations of relations. In the first part of the paper we describe our authorization mechanism as it applies to all objects in the system, regardless of whether they are views or relations. We refer to both views and relations by the collective name "tables." Later we describe the extensions necessary to accommodate a dynamic view mechanism.

Our approach to authorization is based on an access list scheme [13] which permits revocation. We are concerned here with problems relating to access control within an ideal framework which assumes that the following issues are resolved:

(1) User authentication: We presume that a user has been correctly identified by the procedure which has logged him onto the system. In most current systems this means that a user has neither borrowed, forged, nor stolen the USERID (an identifier by which a user is known) of another.

(2) Personnel and program reliability: We do not try to protect against authorized users or programs who release information to unauthorized users by copying it into relations for which those users are authorized.

AUTHORIZATION IN SEQUEL; THE GRANT COMMAND

In System R there is no central database administrator in the usual sense of the term. Any database user may be authorized to create a new table. When he does he is fully and solely authorized to perform actions upon it. (If the table is a view, his authorization may be restricted by the authorization he possesses on the underlying tables.) If he wishes to share his table with other users he may use the GRANT command of the SEQUEL language to give various privileges on that table to various users. Typically a table creator grants a selected set of other users access to his table immediately after he has created it or when he passes that table as a parameter to routines performed by other users. Among the privileges that may be granted on a table are:

- READ: the ability to use this relation in a query. This ability permits a user to read tuples from the relation, to define views based on the relation, etc.
- INSERT: the ability to insert new rows (tuples) into the table.
- DELETE: the ability to delete rows from the table
- UPDATE: the ability to modify existing data in the table. This ability may optionally be restricted to a subset of the columns of the table.
- DROP: the ability to delete the entire table from the system.

The GRANT command of SEQUEL has the form:

```
GRANT { ALL RIGHTS                 } ON ⟨table⟩ TO ⟨user-list⟩ [WITH GRANT OPTION]
      { ⟨privileges⟩               }
      { ALL BUT ⟨privileges⟩       }
```

The grantor may grant all privileges on a table; alternately, he may grant a specific set of privileges or all privileges except those named. ⟨user-list⟩ is the USERID of the grantee, or a list of such grantees. It may also be the keyword PUBLIC, in which case all database users are granted the privileges on that table.

The user may grant a set of privileges with the GRANT option. The GRANT option permits the grantee to further grant his acquired rights to other users. This is analogous to the copy flag of Lampson [13] and of Graham and Denning [11]. For example, let A be the creator of the EMPLOYEE relation and assume that he issues the command

```
GRANT READ, INSERT ON EMPLOYEE TO B
```

After the grant, B possesses the read and insert privileges on the EMPLOYEE relation. If B attempts to grant these privileges on EMPLOYEE to any user, the database system will refuse the command, for B has not been given the GRANT option.

If B has been given the grant option in addition to the READ and INSERT privileges, then he may make such a grant. Assume that the sequence of grants is:

```
A: GRANT READ, INSERT ON EMPLOYEE TO B WITH GRANT OPTION
A: GRANT READ         ON EMPLOYEE TO X WITH GRANT OPTION
B: GRANT READ, INSERT ON EMPLOYEE TO X
```

Both grants by A succeed, since A is the creator of EMPLOYEE. The grant by B also succeeds, since he has been given the GRANT option.

What rights may the user X grant in the above example? Clearly he possesses READ and INSERT privileges on EMPLOYEE, but his INSERT privilege is not grantable. X's source of INSERT privilege is B, and B did not give X the right to further grant it. In fact B may be completely unaware that X has been given the grant option. We therefore subdivide a grantee's privileges into two classes: grantable and not grantable. A user may not grant a privilege P that he has been given unless he was given P with the grant option.

IMPLEMENTATION

Before discussing revocation of granted privileges, we will indicate how the protection mechanism described above might be implemented. The implementation of this restricted authorization subsystem forms the basis of the complete implementation described later.

System R maintains two relations for the use of the authorization subsystem, SYSAUTH and SYSCOLAUTH. The relation SYSAUTH has the following columns:

USERID: indicates the user who is authorized to perform these actions on this table.

TNAME: specifies the table.

TYPE: is 'R' if this table is a base relation, 'V' if it is a view.

A column for each of the privileges READ, INSERT, . . . that may be granted on a table, excluding update, containing a 'Y' or an 'N' to indicate whether the user has that privilege on the named table.

UPDATE: indicates authorization for column update.

GRANTOPT: indicates whether the privileges in this row are grantable to other users.

For each table on which a user is authorized to perform some action, there are up to two tuples in SYSAUTH: one for grantable, and the other for nongrantable privileges. A tuple for that (USERID, table) pair is constructed only when the user has at least one privilege on the given table.

The value of the UPDATE column of a tuple in SYSAUTH is either 'ALL' (all columns may be updated), 'NONE' (no update privileges), or 'SOME'. If UPDATE is 'SOME', then the relation SYSCOLAUTH indicates precisely those columns on which the user holds the UPDATE privilege: For each updatable column, a (user, table, column, grantor, grantopt) tuple is inserted into SYSCOLAUTH.

When a user issues a GRANT command, the authorization module is consulted to determine whether the grant is authorized. The set of grantable privileges possessed by the grantor is intersected with the set of privileges named in the grant, and ths rights which are actually granted are those in this intersection.

The effect of a GRANT command is to insert a new tuple into the SYSAUTH relation, or to appropriately modify an existing tuple. For example, if the grant is made with the GRANT option, and there is already a tuple in SYSAUTH for this (grantee, table) combination, with GRANTOPT = 'Y', then this tuple is modified by setting to 'Y' those columns which correspond to the granted privileges.

REVOCATION

Any user who has granted a privilege may subsequently withdraw it, by issuing the REVOKE command. The format of the REVOKE command is:

$$\text{REVOKE} \begin{Bmatrix} \text{ALL RIGHTS ON} \\ \langle\text{privileges}\rangle \text{ ON} \end{Bmatrix} \langle\text{table}\rangle \text{ FROM } \langle\text{user-list}\rangle$$

Privileges on the named table are denied to the revokee, unless the revokee has another (independent) source of the privilege.

The need to REVOKE a previously granted privilege significantly complicates the authorization mechanism. No longer are two tuples sufficient to represent a user's rights on a table. We must retain the identity of the grantor along with the information previously discussed, because we allow a grantor to revoke only those privileges which he previously granted. This requires up to two tuples in SYSAUTH for each distinct (grantee, table, grantor) combination: one for grantable, and one for nongrantable privileges. Furthermore, upon revocation, the system must (recursively) revoke authorization from those to whom the grantee granted the table.

We will discuss the problem of revocation with respect to a given table. Let the sequence of grants of a specific privilege on a given table by any user before any REVOKE commands be represented by

$$G_1, G_2, \ldots, G_{i-1}, G_i, G_{i+1}, \ldots, G_n,$$

where each G_i is the grant of a single privilege. (We represent grants or revocations of several privileges as a sequence of individual grants or revocations.) If $i < j$, then grant G_i occurred at an earlier time than did G_j. Now suppose that grant G_i is

revoked. The sequence becomes

$$G_1, \ldots, G_{i-1}, G_i, G_{i+1}, \ldots, G_n, R_i.$$

We formally define the semantics of the revocation of G_i to be as if G_i had never occurred. This implies that the state of the authorization relations (SYSAUTH and SYSCOLAUTH) after the above sequence of grants should be identical to their state after the sequence

$$G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n.$$

The state is the same as if the grant of the revoked privilege had never been made.[1] For example, consider the sequence

```
A: GRANT READ, INSERT, UPDATE ON EMPLOYEE TO X
B: GRANT READ, UPDATE         ON EMPLOYEE TO X
A: REVOKE INSERT, UPDATE      ON EMPLOYEE FROM X
```

After the revoke X retains READ and UPDATE privileges on EMPLOYEE.

The general rule is that if the revokee possesses other grants of the revoked privilege from an independent source, then he retains these privileges. In the next section we discuss what we mean by "independent source."

RECURSIVE REVOCATION

Consider the following sequence of grants (assume that A is the creator of the EMPLOYEE relation):

```
A: GRANT ALL RIGHTS ON EMPLOYEE TO X WITH GRANT OPTION
X: GRANT ALL RIGHTS ON EMPLOYEE TO Y
A: REVOKE ALL RIGHTS ON EMPLOYEE FROM X
```

According to the semantics of REVOKE this sequence is equivalent to:

```
X: GRANT ALL RIGHTS ON EMPLOYEE TO Y
```

which fails, since X has no rights on EMPLOYEE. Therefore when a REVOKE command is issued, the system must not only modify the revokee's tuples in SYSAUTH and SYSCOLAUTH but it must also effect a revocation of the revokee's grants of these privileges. If the revoked privileges were not grantable, no further action need be taken.

The decision about exactly which privileges are to be revoked is not obvious. One might expect that if the revokee possesses other grants of the revoked right, then recursive revocation should not take place. The problem is that such an algorithm does not detect cycles in the chain of grants following the revokee. The revocation algorithm must distinguish between the two cases shown in Figure 1. Effectively the correct algorithm traces the grant chains from X back to the creator of the table. If every such path passes through the revoker, then X's privilege should be revoked. However, if there exists a path back to the creator which does not pass through the revoker, then X should retain the privilege after the REVOKE.

[1] This is why we defined a grant of privileges in excess of the available rights as the intersection, rather than rejecting the grant.

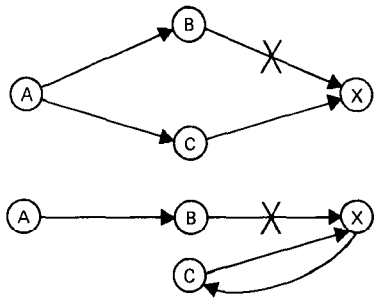

Fig. 1

In any case, the tuple in SYSAUTH representing the revoked grant will be modified or deleted. The tuples in SYSAUTH representing X's remaining privileges on the table, if any, will be consulted to decide exactly which of X's grants should be recursively revoked.

To decide whether to revoke recursively or not, without explicitly tracing the grant graph, we modify the SYSAUTH table slightly. Rather than just 'Y' or 'N', each authorization column contains a timestamp indicating the relative time of a grant. This timestamp may represent real time or it may be a system maintained counter; its important characteristics are that it is monotonically increasing and that no two GRANT commands are tagged with the same timestamp. (Privileges granted in the same command are tagged with the same timestamp.) An authorization column contains 0, signifying no possession of that right, or some positive value T, signifying that the right was granted at time T.

For example, assume that A, B, and C grant certain privileges on the EMPLOYEE table to X, who in turn grants them to Y, as shown in Figure 2. After this sequence of events, the relevant section of the SYSAUTH table looks like:

USERID	TABLE	GRANTOR	READ	INSERT	DELETE
X	EMPLOYEE	A	15	15	0
X	EMPLOYEE	B	20	0	20
Y	EMPLOYEE	X	25	25	25
X	EMPLOYEE	C	30	0	30

Suppose that at time $t=35$, B issues the command REVOKE ALL RIGHTS ON EMPLOYEE FROM X. Clearly the (X, EMPLOYEE, B) tuple must be deleted from SYSAUTH. In order to determine which of X's grants of EMPLOYEE must be revoked, we form a list of X's remaining incoming grants:

TABLE	READ	INSERT	DELETE
EMPLOYEE	{15, 30}	{15}	{30}

as well as a list of X's grants to others:

TABLE	READ	INSERT	DELETE
EMPLOYEE	{25}	{25}	{25}

The grant of the DELETE privilege by X at time $t=25$ must be revoked because his earliest remaining DELETE privilege was received at time $t=30$. But X's grants of READ and INSERT are allowed to remain because they are still "supported" by incoming grants which occurred earlier in time.

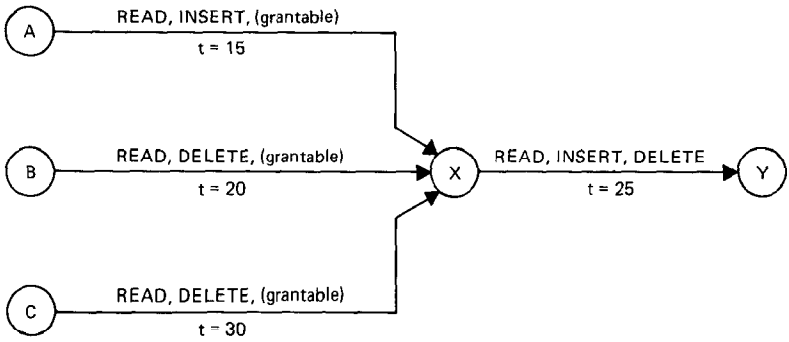

Fig. 2

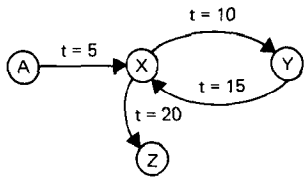

Fig. 3

The revocation algorithm is as follows:

REVOKE: **procedure**(*grantee, privilege, table, grantor*);
comment turn off the grantee's authorization for ⟨privilege⟩ obtained from ⟨grantor⟩;
set ⟨*privilege*⟩ = 0 *in the* (*grantee, table, grantor*) *tuple in* SYSAUTH;
comment find the minimum timestamp for the grantee's remaining grantable ⟨privilege⟩ on ⟨table⟩;
m ← *current timestamp*;
for each *grantor u such that* (*grantee, privilege, table, u, grantable*) *is in* SYSAUTH **do**
  if *privilege* ≠ 0 **and** *privilege* < *m*
    then m ← *privilege*;
comment *revoke grantee's grants of* ⟨*privilege*⟩ *on* ⟨*table*⟩ *which were made before time m*;

for each *user u such that* (*u, privilege, table, grantee*) *is in* SYSAUTH **do**
  if *privilege* < *m*
    then *REVOKE*(*u, privilege, table, grantee*);
return
end *REVOKE*

Example. Consider the graph of grants shown in Figure 3, assuming that all privileges are granted each time, and let the grant from A to X be revoked. Then,

- On the first pass, only the X-to-Y grant is revoked. The X-to-Z grant is *not* revoked, because of the incoming grant at $t = 15$.
- On the second pass, the Y-to-X grant is revoked.
- Finally, on the third pass, the X-to-Z grant is revoked.

If the same privilege is granted by the same grantor to the same user on the same table, then clearly the earlier timestamp should be maintained. The duplicate but later grant should have no effect on the SYSAUTH table. Otherwise, a REVOKE at a later time might cause the revocation of some earlier legitimate grants.

LABELED GRANTS

In the scheme described above grants are not distinguished from each other. Suppose user X has two grants of the same privilege on the same table, both with grant option, one from user A and one from user B, at times 5 and 10, respectively. When X further grants this privilege to a user Y, at time 15, the scheme above does not specify whether it is the privilege X obtained from A or the one X obtained from B. For the purpose of checking Y's authorization, there is no difference between the privilege from A and the privilege from B. In the event of revocation, however, a difference might be desirable. If A gave the privilege to X to give to Y (for example, A-X-Y might be a chain of management, or Y might be a consumer coroutine at the end of a producer-consumer pipeline), then upon revocation by A, A will expect that Y will lose that privilege. In the scheme presented above, the X-to-Y grant is not revoked as long as X has an incoming grant of that privilege (with grant option) at a time earlier than the grant to Y. Consequently the revocation of the X-to-Y grant is dependent upon the time of some other event (namely, the grant from B to X). This is unattractive but acceptable if A does not care whether Y retains the privilege in the presence of a B-to-X grant. In the timestamp scheme, A has *no way* to forbid Y to retain the privilege. From A's viewpoint, he has lost control over this privilege because of the occurrence of an asynchronous event (B's grant). In fact, A cannot even predict whether Y will keep the privilege, because he cannot examine X's other authorizations. We need to be able to give A, if he wishes, the opportunity to retain control over Y's possession of the privilege. This gives our access control scheme the ability to model revocation in capability based mechanisms.

In capability based authorization mechanisms which permit revocation, copies of capabilities are granted and revoked. In the typical capability scheme described by Redell [16], a tree of capability grants is created. When a user A grants a privilege to user B, he does so by copying his capability for that privilege, which is in turn a copy of the capability A received from some grantor. A grant of a privilege, then, is implicitly a grant of exactly one of the grantor's privileges; a grant is labeled with its source. Upon revocation in the system described by Redell, a subtree of grants originating at the revoker is revoked.

We would like to permit users to optionally label their grants to achieve a similar type of control over revocation. A labeled grant, like a capability, may be granted from user to user, resulting in a directed graph of such grants. The syntax for labeling a grant is:

```
GRANT ⟨privileges⟩ ON ⟨table⟩ TO ⟨users⟩
  WITH [GRANT OPTION AND] LABEL ⟨name⟩
```

Upon revocation of a grant made at time T with label L to user X, all grants from X with label L are (recursively) revoked. This recursive revocation is performed regardless of whether the revokee possesses the same privileges from any other source, unless, of course, the revokee possesses the same labeled grant from an independent source.

Consider the following set of grants:

(t=5) A: GRANT ALL RIGHTS ON EMPLOYEE TO X WITH GRANT OPTION AND LABEL L1

(t=10) B: GRANT ALL RIGHTS ON EMPLOYEE TO X WITH GRANT OPTION
(t=15) X: GRANT ALL RIGHTS ON EMPLOYEE TO Y WITH LABEL L1

If A revokes the L1 grant from X, the L1 grant made by X to Y is revoked, regardless of the fact that X still possesses all rights on EMPLOYEE from B.

Hence, labeled grants may coexist with ordinary unlabeled grants. Labeled grants are explicitly granted and revoked according to the timestamps tagging the grants, regardless of the existence of other privileges. All other interactions among grants, labeled or unlabeled, follow the algorithms described in the previous sections.

AUTHORIZATION CHECKING

Because the granting and revocation of privileges are performed dynamically, a user's authorization changes over time. Therefore authorization cannot be checked at compile time. This implies that a user's authorization for a table must be checked before each action on the table. One would expect, however, that accessing tables is a frequent occurrence, while revocation is rare. Consequently the ability to revoke would seem to be a very expensive facility, because it forces these frequent (and usually unnecessary) checks. Our authorization algorithm attempts to minimize the number of times that the system must recompute a user's authorization for a table.

A user in System R interacts with the database in a series of transactions. Each transaction consists of one or more commands to the user interface of the system, such as an INSERT statement or a SEQUEL query [3]. A transaction in System R is the basic unit of integrity, consistency, recovery, and authorization. During a single transaction, authorization to access a particular table is computed only once from the SYSAUTH and SYSCOLAUTH relations.

Within a transaction, when a user first attempts to perform an operation on a table, the system requests the authorization module to determine whether this user is authorized to perform this action on this table. If it is determined that the user is the table's creator (by looking at another system relation, SYSCATALOG), the authorization module returns 'YES'. Otherwise, it consults SYSAUTH to determine whether this user has *any* privileges on this table, and if so, whether he is authorized to perform this particular action. If the reply is 'YES', the system performs the operation; if it is 'NO', it returns an error code to the user. In addition, the authorization module stores a summary of the user's privileges on that table within an authorization cache in virtual memory. On subsequent accesses to that table within the same transaction, this cache is checked rather than the SYSAUTH relation. It follows that authorization to perform an action on a table, once obtained, is held until the end of the transaction. Revocations occurring after authorization has been checked have no effect until the transaction ends.

VIEWS

Up to now we have only been concerned with the granting and revoking of privileges on existing tables. However, System R provides the ability to define views [2, 4] on top of base relations and on top of other views. The view mechanism is also the chosen method to grant selective access to a table. In order to grant privileges on selected row and column subsets, or to grant "statistical" access, the user may define a view of the table and grant that view. The grantee may then query that view, even

though he does not necessarily have any access at all to the underlying table (and therefore cannot even issue the query that defines the view). He may also issue INSERT, DELETE, and UPDATE commands against that view, subject to his granted privileges on that view and to the semantics of view modification.

Other relational systems permit a similar type of data dependent access control. INGRES [17] performs query modification to achieve this. Whenever a user makes a query on a table, the predicate representing his data dependent authorization on that table is conjoined to the query. The MacAIMS system [15] uses filters to implement data dependent authorization.

AUTHORIZATION ON A VIEW

When a user creates a base relation, he is fully and solely authorized to perform actions on that relation. Similarly, when a user defines a view he is solely authorized to perform actions upon it. However, he is not *fully* authorized, for two distinct reasons:

(1) View semantics: certain operations may not be performed on any view; for instance, the creation of an index. Other operations may or may not be allowed, depending upon the view itself. For instance, "statistical" views are not updatable.

(2) The definer's authorization on the underlying tables: clearly a user with read-only access to the EMPLOYEE table should have read-only access to any view which he defines on top of it. We restrict a user's authorization on a view to be only those privileges held on the underlying table(s). If the view involves more than one underlying table, the user's privileges are constrained by the intersection of the privileges which he holds on the underlying tables. Furthermore a privilege P on the defined view is grantable if and only if the definer holds a grantable privilege P on all underlying tables. Recall that the update privilege may be restricted to a subset of the columns of a table; we may then consider the update privilege to be held column by column. Therefore the update privilege is held on a column of a view only if it is held on that column in all of the underlying tables in which it occurs.

Therefore at view definition time an entry is made in SYSAUTH which indicates the definer's privileges on the view, as computed from considerations (1) and (2). The timestamp associated with each privilege is the time of view definition. As in the case of base relations, the privileges for views are separated into two tuples, representing grantable and nongrantable rights.

GRANTING VIEWS

Views may be granted to other users, and the set of grantable privileges is computed in the same manner as for base relations: namely, it is the union of all grantable privileges from all grantors.

REVOKING AND DROPPING VIEWS

While the granting of views proceeds very similarly to the granting of base relations, the presence of a view mechanism makes revocation more difficult. A view may be built on top of granted tables; then this view may be granted and used to construct other views. A view may be dropped (its definition deleted) at any time. As a result, views built on top of the dropped view must also be automatically dropped. Therefore at REVOKE time we need not only to recursively revoke grants of a privilege

on an object, but also to drop or reauthorize views which have been built using that object. (A view is reauthorized by computing its remaining set of privileges.)

To introduce the view mechanism into our formal model of the semantics of revocation, consider a sequence of grants and view definitions, followed by a revocation or a drop. The semantics of this sequence is defined to be the same as if the corresponding grant or view definition had never been made. A revocation or a drop affects both grants and view definitions. All grants of dropped views are automatically revoked. All view definitions based on dropped views, or on tables for which the definer has no remaining privileges, are deleted.

To illustrate the implementation of REVOKE in a system with a view mechanism, and the implementation of DROP in the presence of a grant mechanism, we will present our REVOKE and DROP algorithms as they apply to a restricted version of our authorization mechanism. Then we will indicate how to extend the algorithms to apply to the general case. Consider a system which is restricted in the following manner: individual privileges cannot be granted; on any given table, each user has either all privileges (and can grant them) or no privileges. A user may possess only one grant of any given table.

In such a restricted system the set of grants and view definitions ultimately based on any given table may be described by a tree. The nodes of the tree are (user, table) pairs, representing the fact that the user has all privileges on the table; the arcs of the tree represent grants and view definitions. A grant of table-1 from user-1 to user-2 is represented by an arc connecting a (user-1, table-1) node to a (user-2, table-1) node. A view definition by user-1 of table-1 using table-2, . . . , table-n is represented by connections to a (user-1, table-1) node from each of the nodes (user-1, table-2), . . . , (user-1, table-n).

The commands REVOKE and DROP cut a grant arc or a set of view definition arcs, respectively. They also prune the entire subtree originating at the removed edge(s). To implement REVOKE and DROP, we need two recursive procedures with quite similar structures. Each procedure performs the atomic act of cutting an arc, and then calls itself and the other recursively in order to prune the subtree. Each procedure relies on the system relation SYSUSAGE (UNDERLYING_TABLE, VIEW, DEFINER), which records the fact that the user DEFINER used the UNDERLYING_TABLE in defining the VIEW.

```
REVOKE:  procedure(grantee, table, grantor);
         delete the (grantee, table, grantor) tuple in SYSAUTH;
         for each u such that (u, table, grantee) is in SYSAUTH do
           REVOKE(u, table, grantee);
         for each view such that (table, view, grantee) is in SYSUSAGE do
           DROP(view);
         return;
         end REVOKE;

DROP:    procedure(view);
         delete the view definition from the system;
         for each u1 and u2 such that (u1, view, u2) is in SYSAUTH do
         REVOKE(u1, view, u2);
         for each v and u such that (view, v, u) is in SYSUSAGE do
         DROP(v);
         return;
         end DROP;
```

To extend this simplified model to encompass the full power of our authorization mechanism, we proceed as follows: To properly process revocations from users who have received several grants of the same table, and to detect circular grants, we issue timestamps as before. After a DROP or a REVOKE, those grants are revoked which grant a dropped view, or which are not supported by any remaining earlier incoming grants; those views are dropped which are built using a dropped view, or which are built on a granted table which was not received prior to the time of the view definition.

Selective granting and revocation of individual privileges is handled as before; affected views are reauthorized and dropped only if their definer no longer holds any privileges on the underlying table.

OTHER AUTHORIZATION ISSUES

The reader may have observed that a user who has been granted a view has access to the tuples of that view, even though he may not be able to execute the query which defines the view (because he may not be able to access the underlying tables). Therefore it is not sufficient to transform a query on a view into a query on the base relations and execute that; instead, a view must be evaluated in the authorization context of its definer. This is what Jones [12] speaks of as the "protection environment."

Associated with each view is a "recipe" for materializing tuples of the view, given in terms of operations on the base relations. The recipe is generated automatically by the system from the definition of the view. The granting of authorization on a view is equivalent to authorizing the use of the associated recipe. Therefore once it has been established that a user is authorized to use a view, the recipe is made available, and no further authorization need be performed.

When a user defines a view, a check is made to determine whether he has the necessary authorization on the underlying tables. If he does, then the view definition is accepted and a "recipe" is created for it.

CONCLUSION

We have presented a mechanism which permits the users of a shared database to maintain private data, and which permits them to share a set of privileges on their data with a selected group of other users, or with all users. Subsets of a user's data, derived data, and other transformations of data may be shared by defining a view and sharing that view. Privileges on an object, once granted, may be withdrawn. We have defined the semantics of revocation within a shared database and we have presented a recursive algorithm which effects these semantics. Upon revocation of a privilege from a user, the algorithm revokes his grants of that privilege which were made before his oldest remaining receipt of the privilege. Consequently, privileges legitimately obtained from other sources, and his grants of those privileges, are retained; privileges obtained circularly via a collusion of users are revoked. Examples of the techniques described are presented within the context of System R, a relational database system now under development. Moreover, we feel that these techniques are applicable to any database management system which performs authorization dynamically.

ACKNOWLEDGMENTS

The authors thank Donald D. Chamberlin and Jim Gray of the IBM San Jose Research Laboratory, and David Redell of Project MAC, for many fruitful discussions on the subject of protection and privacy.

REFERENCES

Note. References [14 and 18] are not cited in the text.

1. ASTRAHAN, M.M., ET AL. System R: Relational approach to database management. *ACM Trans. on Database Systems 1*, 2 (June 1976), pp. 97–137.
2. BOYCE, R.F., AND CHAMBERLIN, D.D. Using a structured English query language as a data definition facility. Res. Rep. RJ1318, IBM Research Laboratory, San Jose, Calif., Dec. 10, 1973.
3. CHAMBERLIN, D.D., AND BOYCE, R.F. SEQUEL: A structured English query language. Proc. ACM-SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, Mich., May 1–3, 1974, pp. 249–264.
4. CHAMBERLIN, D.D., GRAY, J.N., AND TRAIGER, I.L. Views, authorization, and locking in a relational data base system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 425–430.
5. CODD, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
6. CODD, E.F. A data base sublanguage founded on the relational calculus. Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif., Nov. 11–12, 1971, pp. 35–68.
7. CODD, E.F. Further normalization of the data base relational model. In *Courant Computer Science Symposium, Vol. 6: Data Base Systems*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 33–64.
8. CODD, E.F. Relational completeness of data base sublanguages. In *Courant Computer Science Symposium, Vol. 6: Data Base Systems*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 65–98.
9. CODD, E.F. Recent investigations in relational data base systems. Proc. IFIP Congr. 1974, North-Holland Pub. Co., Amsterdam, pp. 1017–1021.
10. DATE, C.J. *An Introduction to Data Base Systems*. Addison-Wesley, Reading, Mass., 1975.
11. GRAHAM, G.S., AND DENNING, P.J. Protection—principles and practice. Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 417–429.
12. JONES, A.K. Protection in programmed systems. Ph.D. th., Carnegie-Mellon U., Pittsburgh, Pa., 1973.
13. LAMPSON, B.W. Protection. Proc. Fifth Annual Princeton Conf., Princeton U., Princeton, N.J., March 1971, pp. 437–443.
14. MINSKY, N. Protection of data-bases and the process of user data-base interaction. Tech. Rep. SOSAP-TR-11, Rutgers U., New Brunswick, N.J., Sept. 1974.
15. OWENS, R.C., JR. Primary access control in large-scale time-shared decision systems. Tech. Rep. TR-89, Project MAC, M.I.T., Cambridge, Mass., July 1971.
16. REDELL, D.D. Naming and protection in extendible operating systems. Ph.D. th., U. of California, Berkeley, Calif., Sept. 1974.
17. STONEBRAKER, M.R., AND WONG, E. Access control in a relational data base management system by query modification. Memo No. ERL-M438, Electronics Research Lab., U. of California, Berkeley, Calif., May 1974.
18. SUMMERS, R.C., COLEMAN, C.D., AND FERNANDEZ, E.B. A programming language approach to secure data base access. Tech. Rep. G320-2662, IBM Los Angeles Scientific Center, May 1974.