Database Security Overview

Murat Kantarcioglu

Topics

- The access control model of System R
- Extensions to the System R model
- Views and content-based access control
- Multi-level relational data model

Access Control in Commercial DBMSs

- All commercial systems adopt DAC
- Current discretionary authorization models for relational DBMS are based on the System R authorization model [Griffiths and Wade76]
- It is based on ownership administration with administration delegation

The System R Authorization Model

- Objects to be protected are tables and views
- Privileges include: *select*, *update*, *insert*, *delete*, *drop*, *index* (only for tables), *alter* (only for tables)
- Groups are supported, whereas roles are not
- Privileges can be granted with the GRANT OPTION

The System R - Delegation

- Privilege delegation is supported through the *grant option*: if a privilege is granted with the grant option, the user receiving it can not only exercise the privilege, but can also grant it to other users
- a user can only grant a privilege on a given relation if he/she is the table owner or if he/she has received the privilege with grant option

Grant operation

GRANT *PrivilegeList*| ALL[PRIVILEGES]
ON *Relation* | *View*
TO *UserList* | PUBLIC
[WITH GRANT OPTION]

- it is possible to grant privileges on both relations and views
- privileges apply to entire relations (or views)
- for the update privilege, one needs to specify the columns to which it applies

Grant operation - example

```
Bob:  GRANT select, insert ON Employee TO Ann
        WITH GRANT OPTION;
Bob:    GRANT select ON Employee TO  Jim
        WITH GRANT OPTION;
Ann:    GRANT select, insert ON Employee TO Jim;
```

- Jim has the select privilege (received from both Bob and Ann) and the insert privilege (received from Ann)
- Jim can grant to other users the select privilege (because it has received it with grant option); however, he cannot grant the insert privilege

Grant operation

- The authorization catalogs keep track for each users of the privileges the user possesses and of the ones that the user can delegate
- whenever a user u executes a Grant operation, the system intersects the delegable privileges of u with the set of privileges specified in the command
- if the intersection is empty, the command is not executed

Grant operation - example

Bob: GRANT select, insert ON Employee TO Jim WITH GRANT OPTION;

Bob: GRANT select ON Employee TO Ann WITH GRANT OPTION;

Bob: GRANT insert ON Employee TO Ann;

Jim: GRANT update ON Employee TO Tim WITH GRANT OPTION;

Ann: GRANT select, insert ON Employee TO Tim;

Grant operation - example

- The first three GRANT commands are fully executed (Bob is the owner of the table)
- The fourth command is not executed, because Jim does not have the update privilege on the table
- The fifth command is partially executed; Ann has the select and insert but she does not have the grant option for the insert --> Tim only receives the select privilege

Revoke operation

REVOKE *PrivilegeList*| ALL[PRIVILEGES]

ON *Relation* | *View*

FROM *UserList* | PUBLIC

- a user can only revoke the privileges he/she has granted; it is not possible to only revoke the grant option
- upon execution of a revoke operation, the user from whom the privileges have been revoked looses these privileges, unless has them from some source *independent* from that that has executed the revoke

Revoke operation - example

Bob: GRANT select ON Employee TO Jim WITH GRANT OPTION;

Bob: GRANT select ON Employee TO Ann WITH GRANT OPTION;

Jim: GRANT select ON Employee TO Tim;

Ann: GRANT select ON Employee TO Tim;

Jim: REVOKE select ON Employee FROM Tim;

- Tim continues to hold the select privilege on table Employee after the revoke operation, since he has independently obtained such privilege from Ann.

Revoke operations

- Recursive revocation: whenever a user revokes an authorization on a table from another user, all the authorizations that the revokee had granted because of the revoked authorization are removed
- The revocation is iteratively applied to all the subjects that received the access authorization from the revokee

Recursive revoke

Let G_1, …., G_n be a sequence of grant operations with a single privilege on the same relations, such that i,k = 1,…., n, if i<k, then G_i is executed before G_k. Let R_i be the revoke operation for the privilege granted with operation G_i.

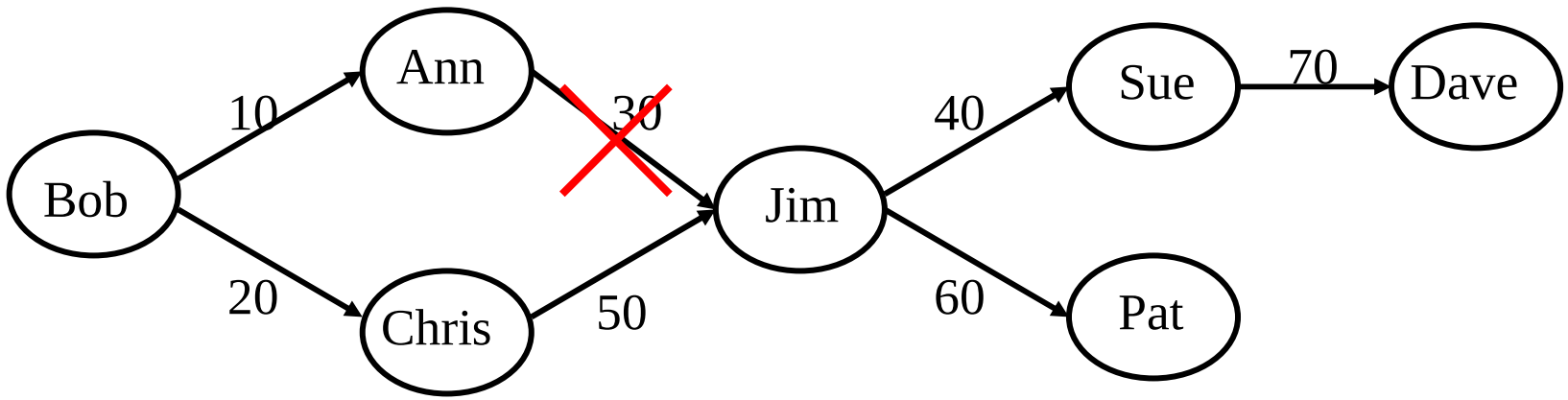
The semantics of the recursive revoke requires that the state of the authorization system after the execution of the sequence

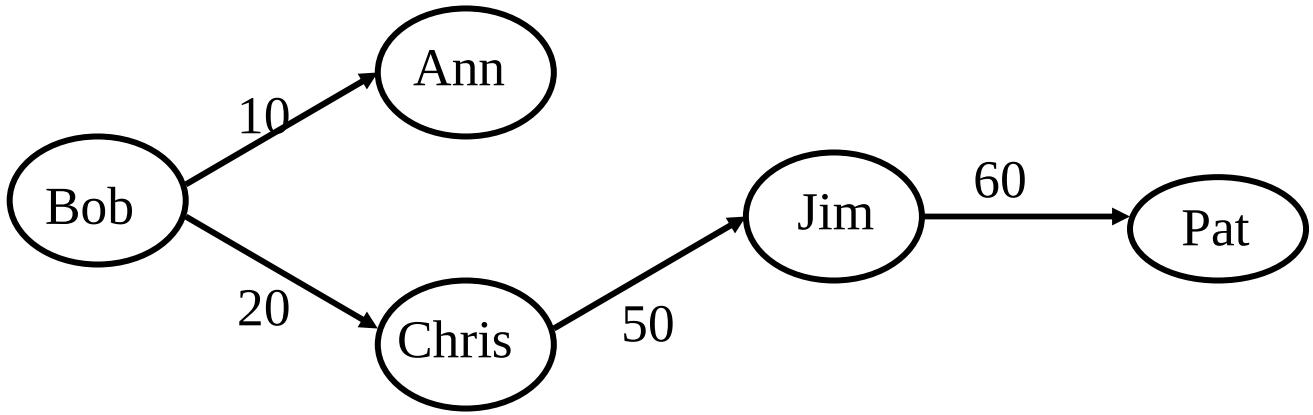
G_1, …., G_n , R_i

be identical to the state that one would have after the execution of the sequence

G_1, …., G_{i-1}, G_{i+1} , …., G_n

Recursive Revocation

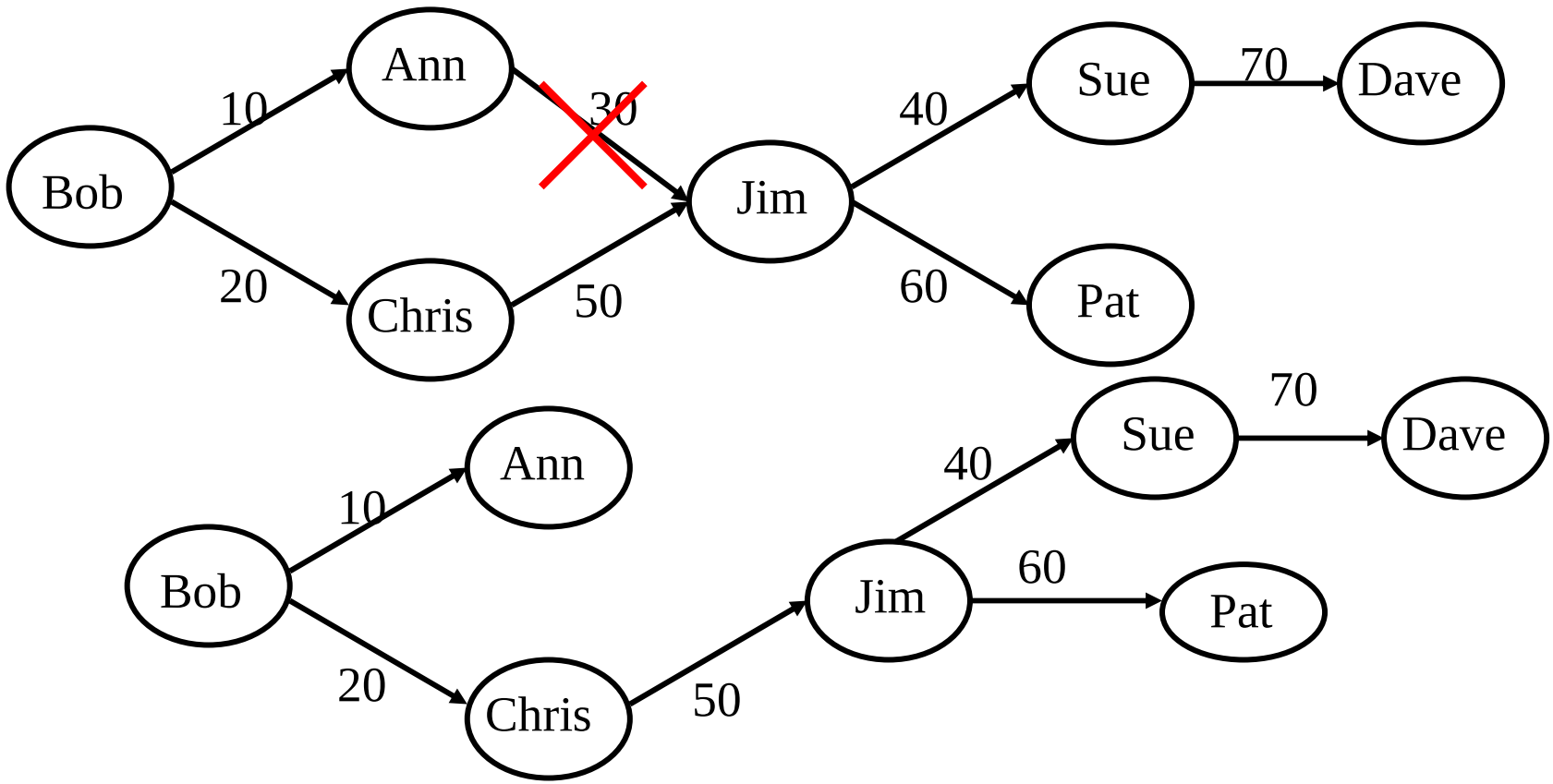




Recursive revocation

- Recursive revocation in the System R takes into account the timestamps denoting when each authorization has been granted
- Variations to this approach have been proposed that do not take into account the timestamps; the reason is to avoid cascades of revoke
- In such variations, the authorizations granted by the revokee are kept as long as the revokee has other authorizations for the same privilege (even if these authorizations have a larger timestamps with respect to the timestamps of the grant operations performed by the revokee)

Recursive revocation without timestamp



Noncascading Revoke

- Recursive revoke can be a very disruptive operation
- A recursive revoke entails:
  - revoking all authorizations the revokee granted, for which no other supporting authorizations exist and, recursively, revoking all authorizations granted through them
  - Invalidating application programs and views

Noncascading Revoke

- The noncascading revoke allows a user to revoke a privilege on a table from another user without entailing automatic revocation of the authorizations for the privilege on the table the latter may have granted
- Instead of deleting the authorizations the revokee may have granted by using the privilege received by the revoker, all these authorizations are restated as if they had been granted by the revoker

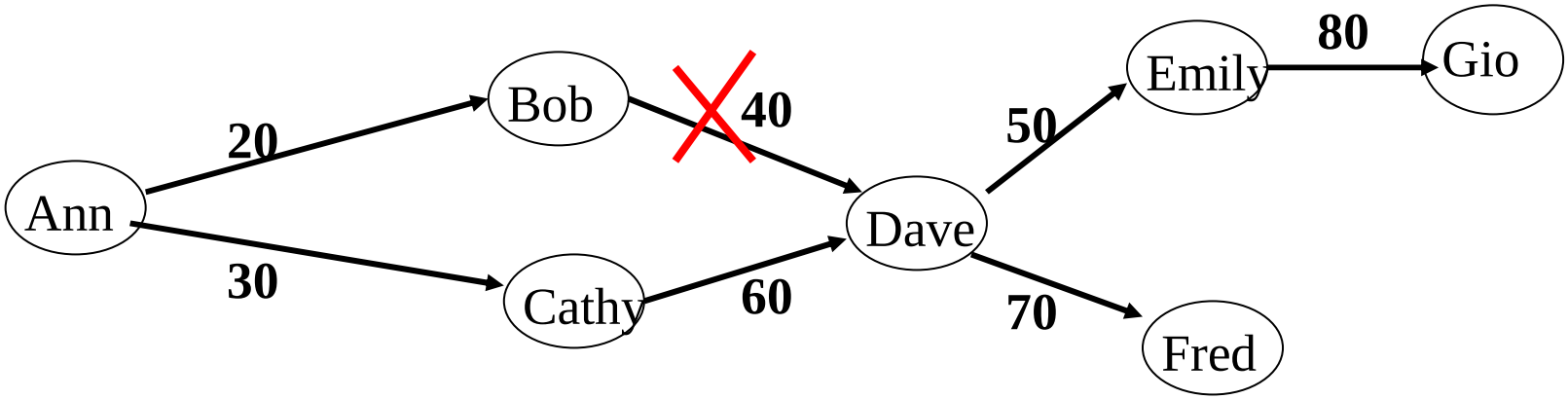
Noncascading Revoke

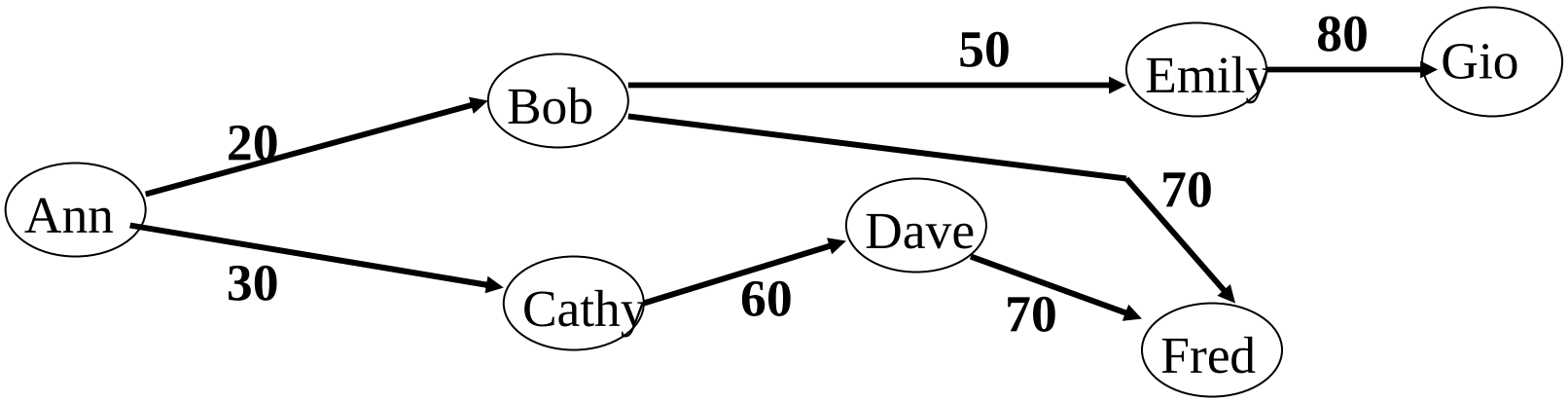
- The semantics of the revocation without cascade of privilege *p* on table *t* from user *y* by user *x* is:
  - To restate with *x* as grantor all authorizations that *y* granted by using the authorization being revoked

Noncascading Revoke

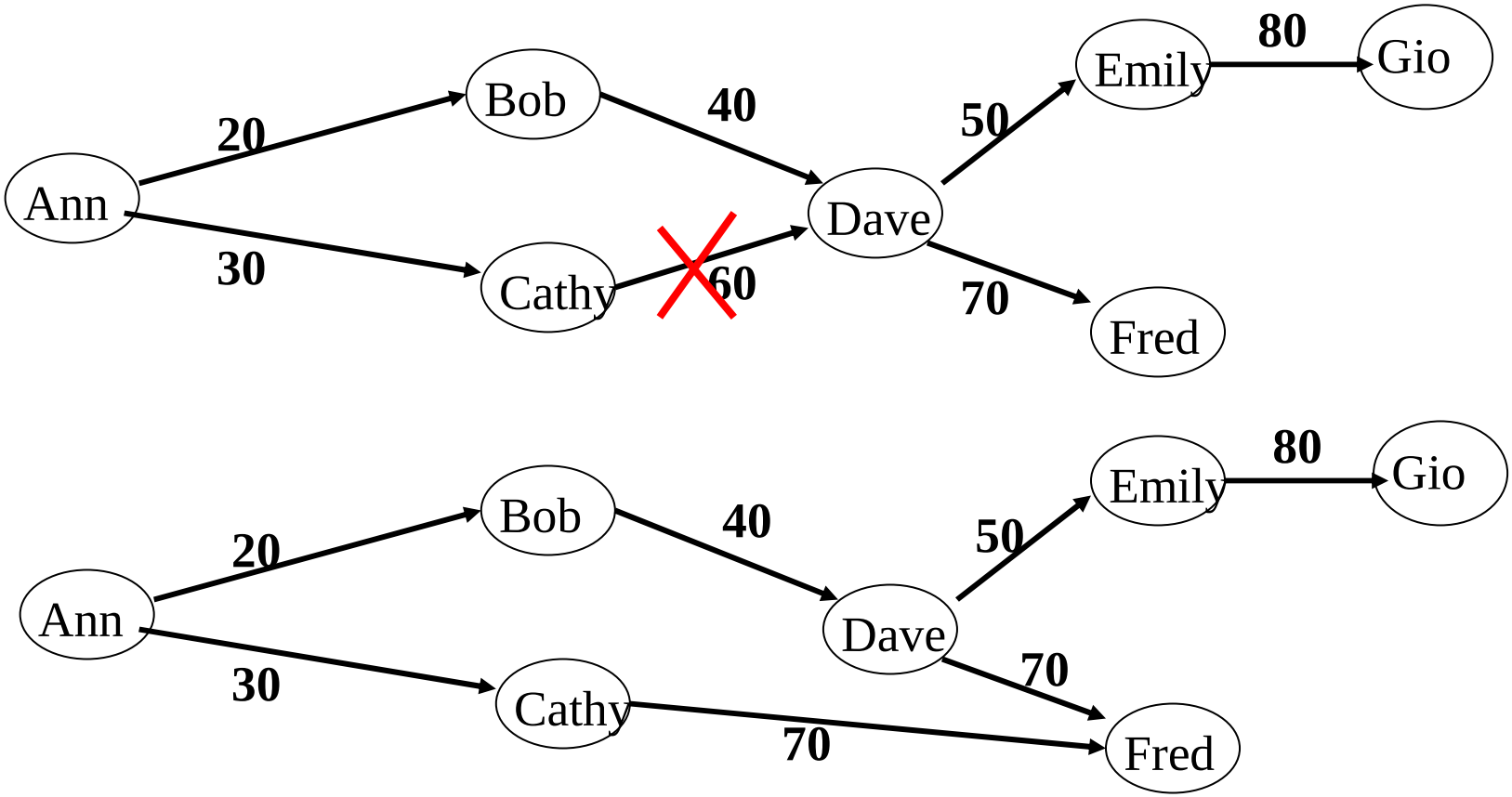
- Note that, since *y* may have received the grant option for the privilege on the table from some other users different from *x*, not all authorizations he/she granted will be given by *x*
- Specifically, *x* will be considered as grantor only of the authorizations *y* granted after recieving the privilege with the grant option from *x*; *y* will still be considered as grantor of all authorizations he/she granted that are supported by other authorizations not granted by *x*

Noncascading Revoke





Noncascading Revoke



Noncascading Revoke

- Note in the previous example that the authorization granted by Dave to Emily has not been specified with Cathy as grantor because it was granted before Dave received the privilege from Cathy

Views and content-based authorization

- Views are a mechanism commonly used to support content-based access control in RDBMS
- Content-based access authorizations should be specified in terms of predicates
- Only the tuples of a relation verifying a given predicate are considered as the protected objects of the authorization

Views and content-based authorization

- The approach to support content-based access control in RDBMS can be summarized as follows:
  - Define a view containing the predicates to select the tuples to be returned to a given subject S
  - Grant S the select privilge on the view, and <u>not</u> on the underlying table

Views and content-based authorization

- Example: suppose we want authorize user Ann to access only the employees whose salary is lower than 20000 – steps:
  - CREATE VIEW Vemp AS
    SELECT * FROM Employee
    WHERE Salary < 20000;
  - GRANT Select ON Vemp TO Ann;

Views and content-based authorization

- Queries against views are transformed through the *view composition* in queries against base tables
- The view composition operation combines in AND the predicates specified in the query on the view with the predicates which are part of the view definition

Views and content-based authorization

Ann: SELECT * FROM Vemp
WHERE Job = ‘Programmer’;

Query after view composition:

SELECT * FROM Employee
WHERE Salary < 20000 AND
Job = ‘Programmer’;

Steps in Query Processing

- Parsing
- Catalog lookup
- Authorization checking
- View Composition
- Query optimization

Note that authorization is performed before view composition; therefore, authorization checking is against the views used in the query and not against the base tables used in these views

Views and content-based authorization

- Views can also be useful to grant select privileges on specific columns: we only need to define a view as projection on the columns on which we want to give privileges
- Views can also be used to grant privileges on simple statistics calculated on data (such as AVG, SUMM,..)

Authorizations on views

- The user creating a view is called the *view definer*
- The privileges that the view definer gets on the view depend from:
  - The view semantics, that is, its definition in terms of the base relation(s)
  - The authorizations that the definers has on the base table

Authorizations on views

- The view definer does not receive privileges corresponding to operations that cannot be executed on the view
- For example, `alter` and `index` do not apply to views

Authorizations on views

- Consider the following view

```
Bob: CREATE VIEW V1 (Emp#,
     Total_Sal)
   AS SELECT Emp#, Salary + Bonus
   FROM Employee WHERE
   Job ='Programmer';
```

The update operation is not defined on column Total_Sal of the view; therefore, Bob will not receive the update authorization on such column

Authorizations on views

- Basically, to determine the privileges that the view definer has on the view, the system needs to intersect the set of privileges that the view definer has on the base tables with the set of privileges corresponding to the operations that can be performed on the view

Authorizations on views - example

- Consider relation Employee and assume Bob is the creator of Employee
- Consider the following sequence of commands:
  - Bob: GRANT Select, Insert, Update ON Employee to Tim;
  - Tim: CREATE VIEW V1 AS SELECT Emp#, Salary FROM Employee;
  - Tim: CREATE VIEW V2 (Emp#, Annual_Salary) AS SELECT Emp#, Salary*12 FROM Employee;

Authorizations on views - example

- Tim can exercise on V1 all privileges he has on relation Employee, that is, Select, Insert, Update
- By contrast, Tim can exercise on V2 only the privileges of Select and Update on column Emp#;

Authorizations on views

- It is possible to grant authorizations on a view: the privileges that a user can grant are those that he/she owns with grant option on the base tables
- Example: user Tim cannot grant any authorization on views V1 and V2 he has defined, because he does not have the authorizations with grant option on the base table

Authorizations on views - example

- Consider the following sequence of commands:
  - Bob: GRANT Select ON Employee TO Tim WITH GRANT OPTION;
  - Bob: GRANT Update, Insert ON Employee TO Tim;
  - Tim: CREATE VIEW V4 AS SELECT Emp#, Salary FROM Employee;

  Authorizations of Tim on V4:

  - Select with Grant Option;
  - Update, Insert without Grant Option;

GRANT Command in SQL 99 Standard

GRANT privileges ON object TO users [WITH GRANT OPTION]

- The following privileges can be specified:
  - SELECT: Can read all columns (including those added later via ALTER TABLE command).
  - INSERT(col-name): Can insert tuples with non-null or non-default values in this column.

    INSERT means same right with respect to all columns. .
  - REFERENCES (col-name): Can define foreign keys (in other tables) that refer to this column.
  - DELETE: Can delete tuples
  - Many vendors support some other priviliges.

- If a user has a privilege with the GRANT OPTION, can pass privilege on to other users (with or without passing on the GRANT OPTION).
- Only owner can execute CREATE, ALTER, and DROP.

GRANT and REVOKE of Privileges in SQL 99

- GRANT INSERT, SELECT ON Emp TO Ann
  - Ann can query Emp or insert tuples into it.
- GRANT DELETE ON Emp TO Jim WITH GRANT OPTION
  - Jim can delete tuples, and also authorize others to do so.
- GRANT UPDATE (*salary*) ON Emp TO Dustin
  - Dustin can update (only) the *salary* field of Emp tuples.
- REVOKE: When a user executes a REVOKE command with CASCADE key word, the effect is to withdraw the named privileges from all users who currently hold these privileges **solely** through a GRANT command that was previously executed by the same user who is now executing the REVOKE command. If these users received the privileges with the grant option and passed it along, those recipients in turn lose their privileges as a consequence of the revoke command, unless they received through an additional grant command. This is basically the recursive revocation without timestamp.

GRANT/REVOKE on Views (SQL 99)

- If the creator of a view loses the SELECT privilege on an underlying table, the view is dropped!
- If the creator of a view loses a privilege held with the grant option on an underlying table, (s)he loses the privilege on the view as well; so do users who were granted that privilege on the view!

Role-Based Authorization (SQL-99)

- In SQL-92, privileges are actually assigned to authorization ids, which can denote a single user or a group of users.
- In SQL:1999 (and in many current systems), privileges are assigned to roles.
  - Roles can then be granted to users and to other roles.
  - Reflects how real organizations work.
  - Illustrates how standards often catch up with “de facto” standards embodied in popular systems.
- It looks like many commercial systems have slightly different implementations.

Access Control in Commercial DBMSs

- Most of the commercial DBMSs also support RBAC features (Informix, Sybase, Oracle)
- However, in most cases they only supports flat RBAC

RBAC – SQL Commands

- CREATE ROLE role-name IDENTIFIED BY passw |NOT IDENTIFIED;

  example:

  CREATE ROLE teller IDENTIFIED BY cashflow;

- DROP ROLE role-name;

RBAC – SQL Commands

- GRANT role TO user | role | PUBLIC [WITH ADMIN OPTION];

  to perform the grant of a role, a user must have the privilege for the role with the ADMIN option, or the system privilege GRANT ANY ROLE

  The ADMIN option allows the receiver to modify or drop the role
- Example:

  GRANT teller TO Bob;

RBAC – SQL Commands

- The grant command for authorization granting can have roles as subjects

  example:

  ```
  GRANT select ON Employee TO teller;
  ```

RBAC – SQL Commands

- SET ROLE role-name IDENTIFIED BY passwd;

  The set command is used enable and disable roles during sessions

  Example: SET ROLE teller IDENTIFIED by cashflow;
- SET ROLE ALL [EXCEPT role-name]

  it can only be used for roles not requiring passwords

  SET ROLE ALL; SET ROLE ALL EXCEPT banker;
- SET ROLE NONE;

  It disables roles for the current session

Multilevel Relational Model

- The multilevel relational (MLR for short) model results from the application of the BLP model to relational databases
- Several issues
  - Granularity: to which element do we apply the classification?
  - Integrity constraints

MLR Model - notation

Standard relational model – each relation is characterized by two components

- A state-invariant *relation scheme*
  - R(A1,…., An) where Ai is an attribute over some domain Di
- A state-dependent *relation* over R composed of distinct tuples of the form (a1,…, an), where each ai is a value in domain Di

MLR Model – keys and FD

- Functional dependencies
  - Let *R* be a relation and let *X* and *Y* be attribute sets, both subsets of the attribute set of R

    we say that *X* **functionally determines** *Y* if and only if not two tuples may exist in the same relation over *R* with the same value for *X* but different values for *Y*
- Primary Keys *(entity integrity property)*
  - the primary key uniquely identifies each tuple in the relation
  - A primary key cannot contain attributes with null values
  - A relation cannot contain two tuples with the same value for the primary key

MLR Model

- Given a relation, an access class can be associated with:
  - The entire relation
  - Each tuple in the relation
    - This is the common choice in commercial systems
  - Each attribute value of each tuple in the relation
    - In the remainder we consider this case

Multilevel (ML) relations

A ML relation is characterized by two components

- A state-invariant *relation scheme*

  R(A1,C1,…., An,Cn, TC) where:

  - Ai is an attribute over some domain Di
  - Ci is a classification attribute for Ai; its domain is the set of access classes that can be associated with values of Ai
  - TC is the classification attribute of the tuple

- A set of state-dependent *relation instances Rc* over R for each access class in the access class lattice. Each instance *Rc* is composed of distinct tuples of the form (a1,c1,…, an,cn, tc), where:
  - ai is a value in domain Di
  - ci is the access class for ai
  - tc is the access class of the tuple determined as the least upper bound of all ci in the tuple
  - Classification attributes cannot assume null values

ML relations - example

Name	C_{Name}	Dept#	$C_{Dept\#}$	Salary	$C_{Dept\#}$	TC
Bob	Low	Dept1	Low	100K	Low	Low
Ann	High	Dept2	High	200K	High	High
Sam	Low	Dept1	Low	150K	High	High

ML relations - instances

- A given relation may thus have instances at different access classes
- The relation instance at class c contains all data that are visible to subjects at level c
- That is, it contains all data whose access classes are dominated by c
- All elements with access classes higher than c, or incomparable, are masked by null values
- Sometimes, to avoid signaling channels, fictitious values (called *cover story values*) can be used

ML relations - instances

Name	C_{Name}	Dept#	$C_{Dept\#}$	Salary	$C_{Dept\#}$	TC
Bob	Low	Dept1	Low	100K	Low	Low
Sam	Low	Dept1	Low	null	Low	Low

Low instance

ML relations - instances

Name	C_{Name}	Dept#	$C_{Dept\#}$	Salary	$C_{Dept\#}$	TC
Bob	Low	Dept1	Low	100K	Low	Low
Ann	High	Dept2	High	200K	High	High
Sam	Low	Dept1	Low	150K	High	High

High instance

ML relations – correctness conditions

ML relations must satisfy the following conditions:

- for each tuple in a ML relation, the attributes of the primary key must have the same access class
- for each tuple in a ML relation, the access class associated with the non-key attributes must dominate the access class of the primary key

ML relations – keys and polyinstantiation

- In the standard relational model, each tuple is uniquely identified, by the values of its key attributes
- When access class are introduced, there may be the need for the simultaneous presence of multiple tuples with the same value for the key attributes but with different classification, which is phenomenon known as *polyinstantiation*

ML relations – polyinstantiation

- Polyinstantiation occurs in the following two situations:
  - When a low user inserts data in a field which already contains data at higher or incomparable level – *invisible polyinstantiation*
  - When a high user inserts data in a field which already contains data at a lower level – *visible polyinstantiation*

ML relations – invisible polyinstantiation

Suppose a low user asks to insert a tuple with the same primary key as an existing tuple at a higher level; the DBMS has three choices:

1) Notify the user that a tuple with the same primary key exists at higher level and reject the insertion
2) Replace the existing tuple at higher level with the new tuple being inserted at low level
3) Insert the new tuple at low level without modifying the existing tuple at the higher level (i.e. polyinstantiate the entity)

Choice 1 introduces a signaling channel

Choice 2 allows the low user to overwrite data not visible to him and thus compromising integrity

Choice 3 is a reasonable choice; as consequence it introduces a polyinstantiated entity

ML relations – invisible polyinstantiation Example

Name	C_{Name}	Dept#	$C_{Dept\#}$	Salary	$C_{Dept\#}$	TC
Bob	Low	Dept1	Low	100K	Low	Low
Ann	High	Dept2	High	200K	High	High
Sam	Low	Dept1	Low	150K	High	High

Assume a low user issue the following insert operation

INSERT INTO Employee
VALUES (Ann, Dept1, 100k)

ML relations – invisible polyinstantiation Example

Name	C_{Name}	Dept#	$C_{Dept\#}$	Salary	$C_{Dept\#}$	TC
Bob	Low	Dept1	Low	100K	Low	Low
Ann	High	Dept2	High	200K	High	High
Sam	Low	Dept1	Low	150K	High	High
Ann	Low	Dept1	Low	100K	Low	Low

The tuples with primary key "Ann" are *polyinstantied*

ML relations – visible polyinstantiation

Suppose a high user asks to insert a tuple with the same primary key as an existing tuple at lower level; the DBMS has three choices:

1) Notify the user that a tuple with the same primary key exists and reject the insertion
2) Replace the existing tuple at lower level with the new tuple being inserted at the high level
3) Insert the new tuple at high level without modifying the existing tuple at the lower level (i.e. polyinstantiate the entity)

Choice 1 does not introduce a signaling channel; however, rejecting the insertion my result in a DoS problem

Choice 2 would result in removing a tuple at lower level and thus introduce a signaling channel

Choice 3 is a reasonable choice; as consequence it introduces a polyinstantiated entity

ML relations – polyinstantiation

- The introduction of data classification in relational DBMS introduces polyinstantiation
- Several approaches have been developed to handle this problem
  - Approaches that allows polyinstantion
    - Sandhu&Jajodia, SeaView Model by Denning et al.
      - These approaches define the key of a multilevel relation to be a combination of the original key attributes and their classifications
    - Belief-based model by Smith and Winslett
  - Approaches that prevent polyinstantion
    - Require that all keys be classified at the lowest possible access class
    - Partition the domain of the primary key among the various access classes so that each value has a unique possible classification